Analyzing Classical Ciphers with Machine Learning

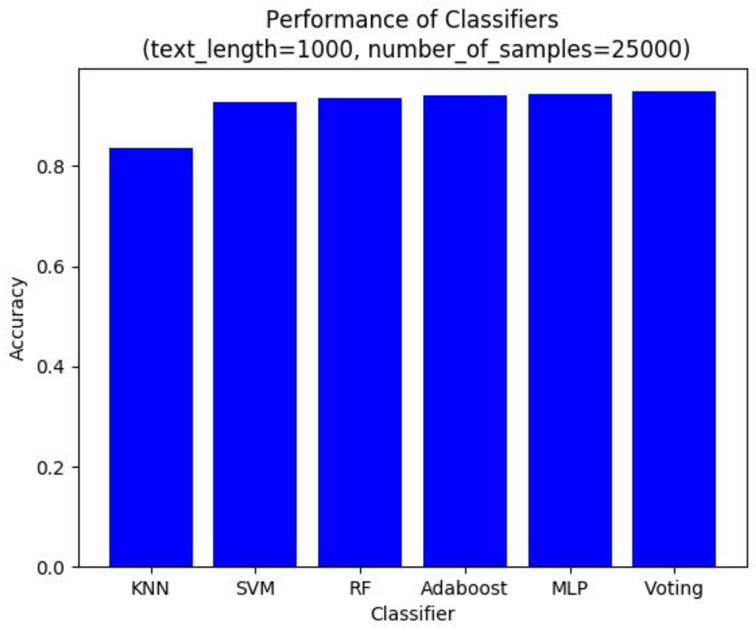
Aaron Smith, Zizhen Huang

12/3/2019

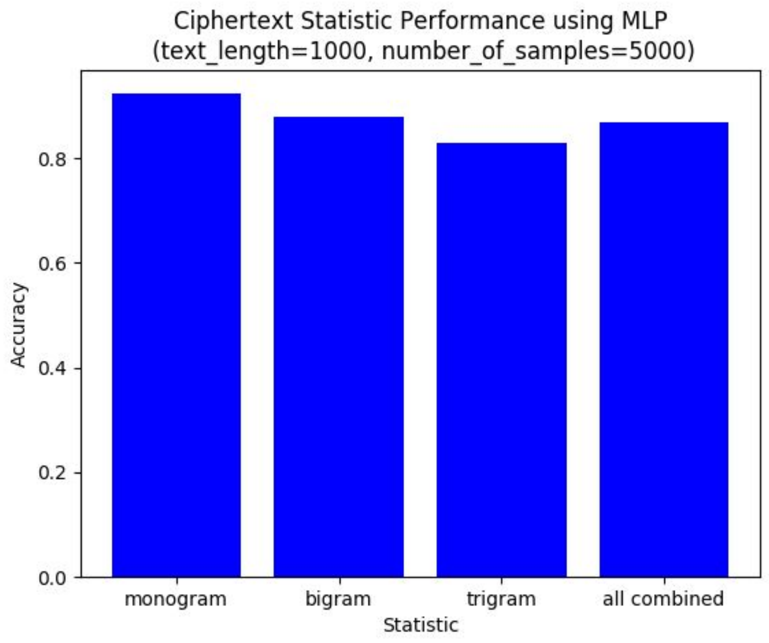
For our project, we set out to use machine learning to train a computer to learn something about classical ciphers. Our goals were to be able to classify a particular ciphertext according to its respective cipher, and to try to decrypt (or find the key of) a ciphertext using machine learning. Our experiments used five different ciphers, ciphertexts of various lengths, and a handful of machine learning techniques.

To collect our data, we programmed each cipher in Python in order to generate our ciphertext samples. The ciphers we considered included: shift, columnar transposition, vigenere, playfair, and hill cipher. Each cipher used a randomized key, and each plaintext was found from a random line in the brown corpus. Text was filtered so that there were only 26 symbols in our plaintexts and ciphertexts: only lowercase letters, no special characters or spaces. For each cipher, we generated five files of ciphertext samples. Each file has 10,000 ciphertext samples, and each file has ciphertext samples of a particular length. The lengths we considered were: 100, 200, 300, 500, and 1000 characters long. Thus, all in all, 25 txt files were created each containing 10,000 ciphertext samples.

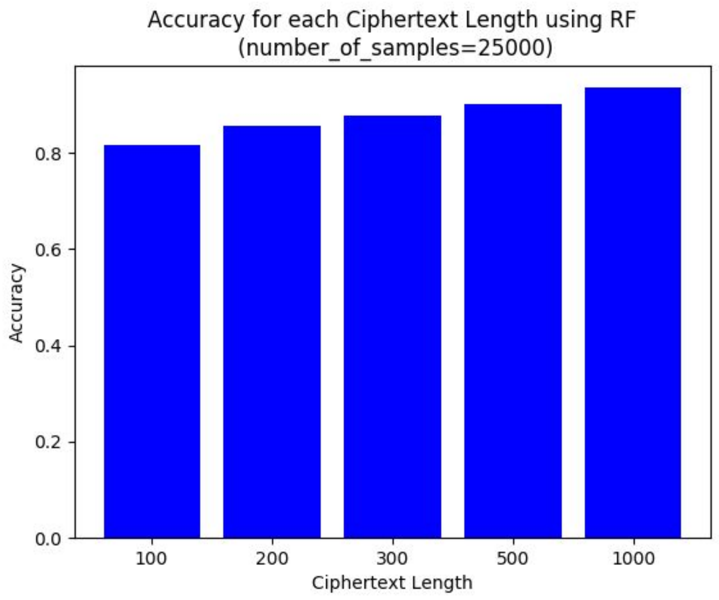
After generating our data, we moved on to classifying ciphertexts using various machine learning techniques. Each technique considered the same two parameters: text_length which is how many characters were in the ciphertext, and number_of_samples which is the total number of ciphertext samples that were used for training. We tested the following techniques (all from the scikit-learn package): K-Nearest Neighbor (KNN), Support Vector Machine (SVM), Random Forest (RF), Adaboost, Neural Network (MLP), and a Voting Classifier. The accuracy of each technique was pretty similar (with the exception of KNN).



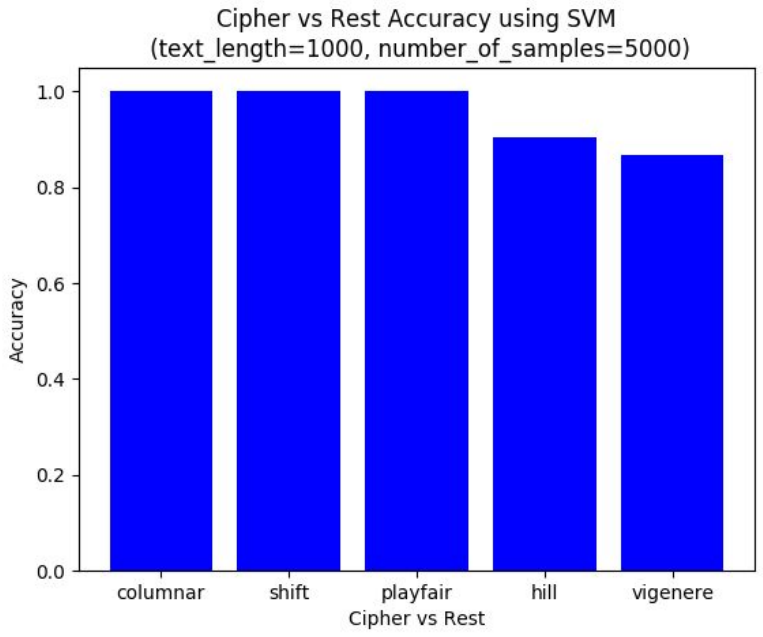
For all our tests (except this one), we trained our models using ciphertext monogram counts (i.e. the number of times each ciphertext character appears in the ciphertext). Thus, for each ciphertext, a vector of length 26 is created. Thousands of such vectors are then used for training, and thousands more for scoring. We also considered taking into account other ciphertext statistics for training and scoring. For this test, we considered bigram and trigram counts. Bigram counts are the number of times that pairs of ciphertext characters appear, while trigram counts are the number of times that three ciphertext characters appear together in a particular order. Thus, for bigram counts a vector of length 26 * 26 is created, and for trigram counts a vector of length 26 * 26 * 26. We then tested the accuracy of each statistic by testing them on a Neural Network Classifier. Each run through used the exact same parameters, just different training and scoring vectors. In addition, each test was run multiple times and the best accuracy was recorded. We then tried combining all three statistics into each of our training/scoring vectors to observe the results. We found that monogram statistics were not only the simplest method but also performed the best.



We next looked at how the length of the ciphertext affects the accuracy of the classifier. We tested this by training a Random Forest Classifier on each length of ciphertext that we created. The results are what you'd expect, with greater length comes greater accuracy.



We then wanted to test if some types of ciphertexts are easier to classify than others. Scikit-learn's SVM by default supports multi-class classification (it's actually called Support Vector Classifier or SVC). Normally, an SVM only supports classification amongst two classes. To try to classify amongst five different classes using SVM, you could train five SVM's, one for each class. This is what we did, where each SVM was implemented in a one vs. rest scheme. As an example, to classify something as shift, we created an SVM with two classes, either shift or not shift. The shift classification was trained on shift ciphertexts, while the non-shift classification was trained on a combination of ciphertexts corresponding to all the other ciphers. We then repeated this for each cipher, obtaining five one vs. rest SVM's. The results show that if a ciphertext belongs to columnar transposition, shift, or the playfair ciphers, then there is close to a 100% chance of classifying it correctly (at least when the text_length = 1000). Hill and vigenere ciphers, on the other hand, are more difficult to correctly classify.

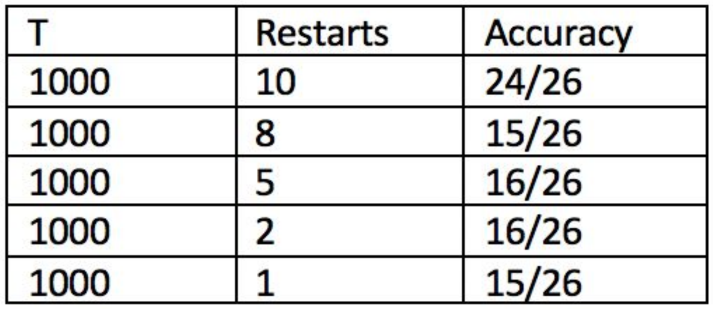


Why are some types of ciphertexts easier to classify than others? Perhaps because of how each cipher actually works. Columnar transposition cipher, for example, should be easy to classify because it doesn't actually change the plaintext monogram counts, rather it just rearranges the plaintext characters. This is unlike any of the other ciphers, and should be plain why it is easy to tell columnar transposition from the rest. What about the rest of them? The shift cipher is a monoalphabetic cipher, playfair is a bigram substitution cipher, hill is a polygraphic substitution cipher, and vigenere is a polyalphabetic substitution cipher. Thus, each cipher works differently, and yields a ciphertext that has particular characteristics. From our results, it would seem that hill and vigenere ciphers yield less distinct characteristics than the rest of the ciphers (at least when it comes to monogram counts).

For all of the techniques we used for classification, many tests were run. Parameters were changed to try to find the best results. As an example, we found that the poly kernel for SVM worked the best. A few more things to note. The adaboost classifier in the scikit-learn package doesn't support using different types of classifiers. So after testing it on multiple valid classifiers, we found that using adaboost on a collection of random forest classifiers yielded good results (though not much better than random forest on its own). Voting classifier is essentially what it sounds like, it takes a collection of different types of classifiers and each one is given a weighted vote. Whichever classification has the most votes wins. We did some tweaking with the classifiers and the weights and eventually got good results (though barely better than any one of the classifiers on their own). Perhaps given more time, it would have been beneficial to write our own custom multi-class adaboost algorithm and input all of our classifiers in order to train the best weights. However, even if we did do that, I doubt we would've seen a significant improvement over the results we've already had.

After we have prepared all the interesting ciphers, we decide to crack them, at least some of those easy ones.

We started with simple substitution cipher because it should be pretty easy to crack by using Hidden Markov Model. As what we expected, even with not enough plain English text data, we are able to successfully crack the shift key. This confirms that the value of N, the number of hidden states, matches the number of hidden features in the observation sequence we take in to train and analyze. For example, when N = 2. It means we want to split the ciphertext into vowels and consonants. But when it comes to complete substitution cipher, we understand that the number of hidden states corresponds to the 26 unique English letters.



T	Restarts	Accuracy
1000	10	24/26
1000	8	15/26
1000	5	16/26
1000	2	16/26
1000	1	15/26

Table 1

We started training our model with observation (cipher text) length 1000 and each associated with a different number of restarts. We initialized our initial matrix A as the English letter frequency. After the training, we found that the accuracy decreases as the number of restarts decreases.

T	Restarts	Accuracy
500	1000	0
500	8	0
500	5	0
500	2	0
500	1	0

Table 2

T	Restarts	Accuracy
300	100	0
300	8	0
300	5	0
300	2	0
300	1	0

Table 3

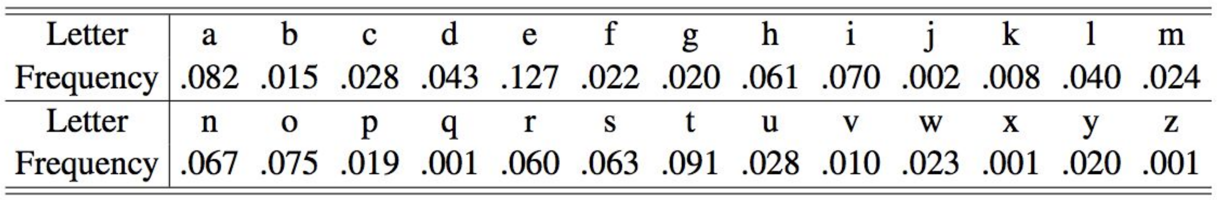
T	Restarts	Accuracy
200	100	0
200	8	0
200	5	0
200	2	0
200	1	0

Table 4

Surprisingly, when we change the ciphertext length to 500, 300 and 200, the accuracy drops to 0. This is quite confusing because it should at least have some very low accuracy. It definitely needs more tweaking. However, we did confirm that HMM works very well for simple substitution cipher.

Since, simple substitution is very easy for HMM to handle. We figured that maybe we should try it with a more complicated cipher: vigenere cipher. Maybe vigenere cipher is just really a formula that we could possibly view it algebraically.

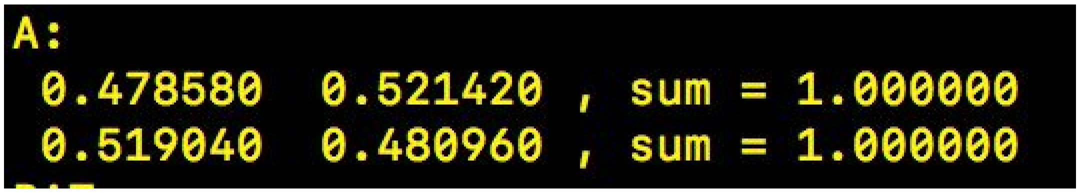
The tricky part of cracking a vigenere cipher is to determine the length of its key (what is vigenere cipher). Once, we found its key length, which we view it as the value of N. Thus, it becomes a simple substitution problem for us. Hope we are lucky.



Letter	a	b	c	d	e	f	g	h	i	j	k	l	m
Frequency	.082	.015	.028	.043	.127	.022	.020	.061	.070	.002	.008	.040	.024
Letter	n	o	p	q	r	s	t	u	v	w	x	y	z
Frequency	.067	.075	.019	.001	.060	.063	.091	.028	.010	.023	.001	.020	.001

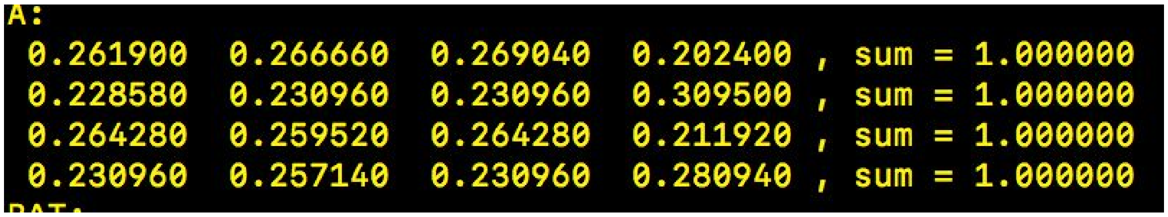
Table 5

We choose the key as BHZFFX and encrypted text data from our project. Now, to actually do the training we have some preparation, what we are expecting is that the final A matrix and the final B matrix will tell us something. To find the actual N value. We started with N = 2. The results does not really help us. See below:



```
A:
 0.478580  0.521420 , sum = 1.000000
 0.519040  0.480960 , sum = 1.000000
```

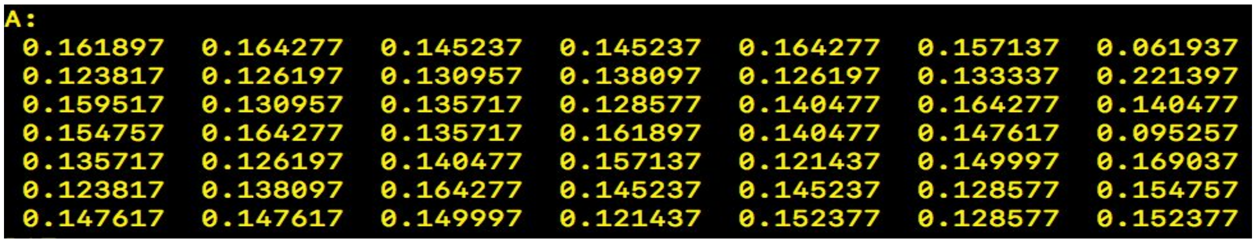
So, how about N = 4. Still nope. See below:



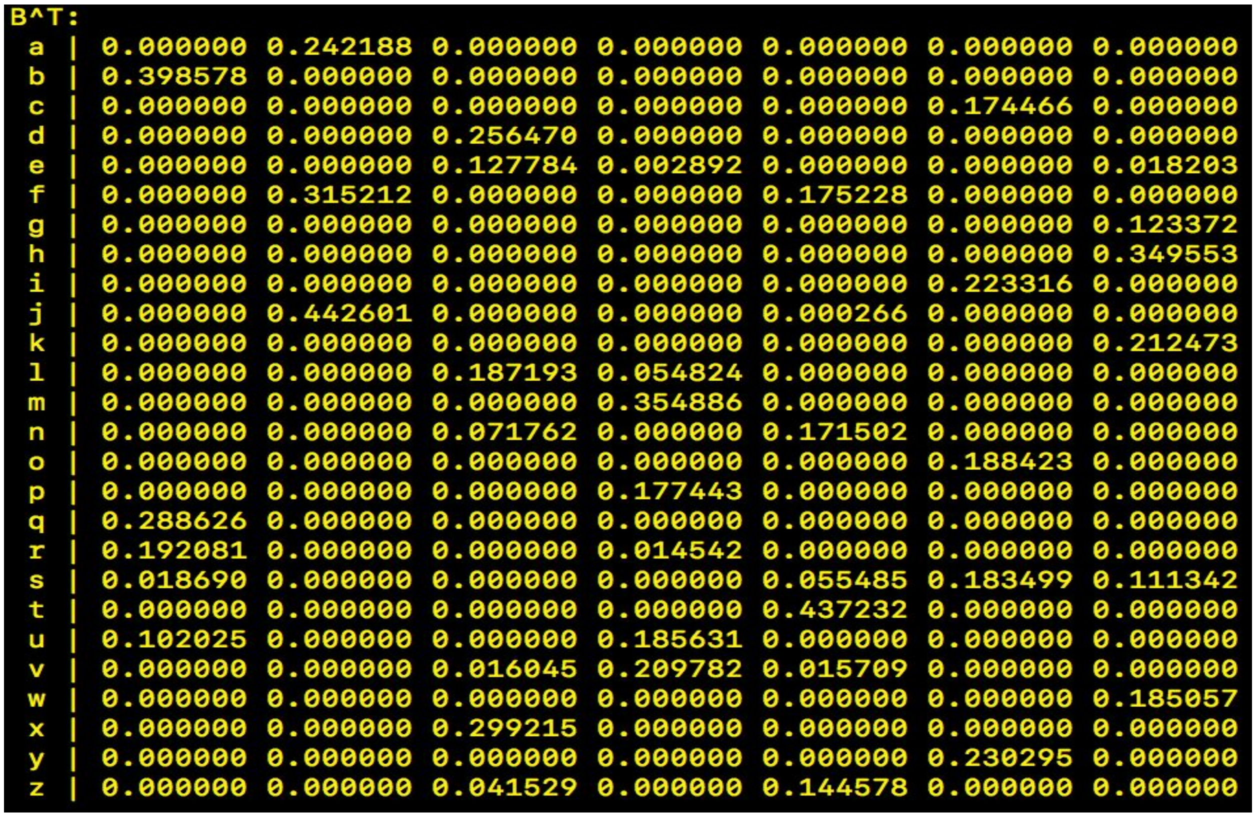
```
A:
 0.261900  0.266660  0.269040  0.202400 , sum = 1.000000
 0.228580  0.230960  0.230960  0.309500 , sum = 1.000000
 0.264280  0.259520  0.264280  0.211920 , sum = 1.000000
 0.230960  0.257140  0.230960  0.280940 , sum = 1.000000
```

Since our final A matrix does not converge, this means the transition probability does not give us enough information. So, we need a bigger N.

N = 7



```
A:
 0.161897  0.164277  0.145237  0.145237  0.164277  0.157137  0.061937
 0.123817  0.126197  0.130957  0.138097  0.126197  0.133337  0.221397
 0.159517  0.130957  0.135717  0.128577  0.140477  0.164277  0.140477
 0.154757  0.164277  0.135717  0.161897  0.140477  0.147617  0.095257
 0.135717  0.126197  0.140477  0.157137  0.121437  0.149997  0.169037
 0.123817  0.138097  0.164277  0.145237  0.145237  0.128577  0.154757
 0.147617  0.147617  0.149997  0.121437  0.152377  0.128577  0.152377
```



```
B^T:
 a | 0.000000 0.242188 0.000000 0.000000 0.000000 0.000000 0.000000
 b | 0.398578 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 c | 0.000000 0.000000 0.000000 0.000000 0.000000 0.174466 0.000000
 d | 0.000000 0.000000 0.256470 0.000000 0.000000 0.000000 0.000000
 e | 0.000000 0.000000 0.127784 0.002892 0.000000 0.000000 0.018203
 f | 0.000000 0.315212 0.000000 0.000000 0.175228 0.000000 0.000000
 g | 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.123372
 h | 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.349553
 i | 0.000000 0.000000 0.000000 0.000000 0.000000 0.223316 0.000000
 j | 0.000000 0.442601 0.000000 0.000000 0.000266 0.000000 0.000000
 k | 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.212473
 l | 0.000000 0.000000 0.187193 0.054824 0.000000 0.000000 0.000000
 m | 0.000000 0.000000 0.000000 0.354886 0.000000 0.000000 0.000000
 n | 0.000000 0.000000 0.071762 0.000000 0.171502 0.000000 0.000000
 o | 0.000000 0.000000 0.000000 0.000000 0.000000 0.188423 0.000000
 p | 0.000000 0.000000 0.000000 0.177443 0.000000 0.000000 0.000000
 q | 0.288626 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
 r | 0.192081 0.000000 0.000000 0.014542 0.000000 0.000000 0.000000
 s | 0.018690 0.000000 0.000000 0.000000 0.055485 0.183499 0.111342
 t | 0.000000 0.000000 0.000000 0.000000 0.437232 0.000000 0.000000
 u | 0.102025 0.000000 0.000000 0.185631 0.000000 0.000000 0.000000
 v | 0.000000 0.000000 0.016045 0.209782 0.015709 0.000000 0.000000
 w | 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.185057
 x | 0.000000 0.000000 0.299215 0.000000 0.000000 0.000000 0.000000
 y | 0.000000 0.000000 0.000000 0.000000 0.000000 0.230295 0.000000
 z | 0.000000 0.000000 0.041529 0.000000 0.144578 0.000000 0.000000
```

Unfortunately, the final A and B are not able to tell us anything useful even when we train the data with the correct N value 6. The problem is that the final A matrix never converge to become deterministic and the B matrix does not reflect the change of ratio comparing to the English frequency in table 1 above. Without these two, we can not determine the actual key length and key letters. Hence, that leaves us some more homework to work on.

Another approach we tried is to use recurrent neural network LSTM to see if we can do anything with the vigenere cipher. However, we had a hard time to implement tensorflow correctly on LSTM and the result does not give us any useful answer.

Resources used during the project:

For programming the ciphers:

- https://en.wikipedia.org/wiki/Playfair_cipher
- https://en.wikipedia.org/wiki/Substitution_cipher
- https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
- https://en.wikipedia.org/wiki/Hill_cipher

For reference:

- https://greydanus.github.io/2017/01/07/enigma-rnn/
- *Introduction to Machine Learning with Applications in Information Security* by Mark Stamp
- https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1408&context=etd_projects
- https://scikit-learn.org/stable/

Please note that this paper didn't quote anything or use any outside knowledge that would require a traditional works cited page.