AN ARTIFICIAL NEURAL NETWORK APPROACH TO FINDING THE KEY LENGTH OF THE VIGENÈRE CIPHER

CHRISTIAN MILLICHAP AND YEEKA YAU

ABSTRACT. In this article, we create an artificial neural network (ANN) that combines both classical and modern techniques for determining the key length of a Vigenère cipher. We provide experimental evidence supporting the accuracy of our model for a wide range of parameters. We also discuss the creation and features of this ANN along with a comparative analysis between our ANN, the index of coincidence, and the twist-based algorithms.

1. INTRODUCTION

Artificial Neural Networks (ANNs) have recently seen a plethora of applications throughout academia. The development of open-source software libraries, such as TensorFlow and Keras, have made it possible to quickly train and test ANNs for a variety of purposes. In particular, there has been a steady stream of applications to cryptanalysis related questions in historical cryptology in the last 10 years; see [10] and [8] for examples of ANNs created for cipher detection among sets of classical ciphers. Here, we build an ANN to attack another important cryptanalysis question in historical cryptology: how can we accurately predict the key length of a Vigenère cipher?

The Vigenère cipher is one of the most well-studied historical ciphers. The development of this cipher can be traced back to the mid 1400s and involved several cryptologists. We refer the reader to Chapter 4 of [6] for a detailed description of the history of the Vigenère cipher. For this cipher, a keyword of length k is used to designate a sequence of k shifts that are repeatedly used in order for encryption. Once a cryptanalyst knows the key length is k, then they can partition the ciphertext into k cosets, each of which contains letters that have been encrypted with the same shift (or the same alphabet for an arbitrary polyalphabetic substitution cipher). Assuming these cosets are sufficiently large, it is a straightforward task to finish breaking this cipher via frequency analysis; see [2, Chapter 2] for an example that implements this technique. This all motivates the importance of having accurate and efficient algorithms for first finding the key length.

There are several algorithms that have been developed to predict this key length, which we review in detail in Section 2. Historically, the Babbage–Kasiski test [7] and the index of coincidence [15] are the two most well known techniques, which have been applied for over a hundred years now. More recently, a series of papers have introduced the twist-based algorithms as new approaches to predict key length. The original twist algorithm was introduced by Barr–Simoson in [1]. Park–Kim–Cho–Yum provided an improved version, called the twist$^+$ algorithm, in [12]. Further modifications were made by Millichap–Yau–Pate–Carns in [9] to build the twist^{++} algorithm. For a wide variety of key lengths and text lengths, each of these algorithms has its own strengths and weaknesses. Therefore, it

is natural to build an ANN for key length prediction that uses these algorithms in hopes of highlighting their individual strengths, while mitigating their weaknesses.

Our work here provides an accurate model for predicting the key length of a Vigenère cipher for a large range of key lengths and text lengths. This ANN implements both classical tools (index of coincidence in a variety of formats) and much more recent algorithms (twist-based algorithms) along with a few other features. Our exact model features are highlighted in Section 3.3. Accuracy comparisons with the index of coincidence and the twist-based algorithms are given in Table 2, which clearly highlight the superior accuracy of our ANN for a variety of text lengths.

Our paper is organized as follows. In Section 2, we review a variety of well-established methods from the literature for predicting the key length of a Vigenère cipher and provide an analysis of their strengths and weaknesses. All of these tools were tested as potential features for our ANN. In Section 3, we discuss the creation, training, and evaluation of an ANN for finding the key length of the Vigenère cipher.

2. Key Length Attacks

In this section, we review a variety of algorithms that predict the key length of a Vigenère cipher. Strengths and weaknesses of these algorithms will also be discussed.

2.1. The Babbage–Kasiski Test.

For the Babbage–Kasiski test, one looks for repetitions of n-grams ($n \geq 3$) in the ciphertext. Usually, such repetitions represent the same plaintext with the same portion of the keyword used to encrypt that plaintext; see [13] for a analysis of accidental repetitions. Thus, the distance between these repetitions should be a multiple of the key length. By finding several such repetitions and calculating the greatest common divisor of these distances that show up most frequently, one can then formulate a conjecture for the key length. This test has the advantage of being independent of the underlying language and alphabet.

There are several implementation challenges that can occur when applying the Babbage–Kasiski Test. If a ciphertext does not contain any repeated trigrams or only contains a few, then the Babbage–Kasiski test might not prove helpful. In addition, the Babbage–Kasiski test could easily direct one towards a multiple of the key length or a divisor of the key length, rather than the actual key length. Furthermore, there might be multiple values that frequently show up as distinct greatest common divisor of distances between repetitions. How should one decide which is the best conjecture for the key length or an ordering for key length conjectures?

2.2. The Index of Coincidence.

The Index of Coincidence (IC) calculates the probability that two randomly chosen distinct letters from a text are the same. Mathematically,

$$IC(\mathcal{M}) = \frac{\sum_{i=1}^{26} f_i(f_i-1)}{N(N-1)}, \tag{1}$$

where N is the length of a text $\mathcal{M}$ and f_i represents the frequency of the i^{th} letter of the alphabet in this ciphertext. When the index of coincidence is applied to a Vigenére ciphertext $\mathcal{M}$, then one can estimate the key length k via

$$k \approx \frac{0.028N}{IC(\mathcal{M})(N-1) - 0.038N + 0.066}. \tag{2}$$

Furthermore, if one has a conjectured key length of m, then one can partition $\mathcal{M}$ into m cosets, where coset $\mathcal{M}_i$ contains the ciphertext letters encrypted by the i^{th} key letter, for $1 \leq i \leq m$. From here, one can apply the IC to each $\mathcal{M}_i$. If these values approximate the IC for the underlying language of the plaintext, then there is a good chance m is the actual key length since the IC of a set of letters coming from a shift is the same as the IC of the corresponding plaintext letters. We refer the reader to Section 2.3 of [2] for details on (2) and an example of applying the IC to cosets of a ciphertext.

While it is simple to calculate the IC and the approximation for k via Equation (2), this tool does have its weaknesses. For sufficiently long keys, estimated key lengths are sensitive to a small perturbation in the IC. Thus, the IC becomes less reliable as key length increases, which can also be seen experimentally in Figure 2 of [5] and Figure 3 of [12]. In addition, the key length estimate given above is dependent on your keyword containing k distinct letters. If many letters are repeated in the keyword, then the IC will most likely underestimate k. Finally, the IC is dependent on your underlying language since different languages have different letter frequency distributions.

2.3. **The Twist-Based Algorithms.** Before reviewing the twist-based algorithms, we first introduce some necessary notation and definitions. Suppose we are given a text $\mathcal{M}$ of length N. We first form a **sample signature** for $\mathcal{M}$, which is $C =< c_1, c_2, \ldots, c_{26} >$, where $c_i = \frac{f_i}{N}$ with f_i representing the number of frequencies of the i^{th} most common letter in $\mathcal{M}$. In other words, C is the ordered set of relative frequencies for text $\mathcal{M}$. Then we can compute the **twist** of a sample signature:

$$\diamondsuit C = \sum_{i=14}^{26} c_i - \sum_{i=1}^{13} c_i.$$

When a sufficiently long text $\mathcal{M}$ is a plaintext or a ciphertext encrypted with a monoalphabetic substitution cipher, then $\diamondsuit C$ should reflect the behavior of the underlying language (English in all cases considered in this paper) and be relatively large based on the variance of frequency distributions in the underlying language. However, if $\mathcal{M}$ is a random text that would lack this variation in frequency distribution, then we should expect $\diamondsuit C$ to be quite small.

Now, suppose $\mathcal{M}$ is a ciphertext of length N that was encrypted using the Vigenère Cipher. Further, suppose we conjecture a key length of $m \in \mathbb{N}$. Then we can partition $\mathcal{M}$ into m cosets $\{\mathcal{M}_j\}_{j=1}^{m}$, where $\mathcal{M}_j$ contains all the letters encrypted with the j^{th} letter of the (conjectured) key of length N. Let C_j represent the sample signature for $\mathcal{M}_j$, and let

$$\diamondsuit C_j = \sum_{i=14}^{26} c_{i,j} - \sum_{i=1}^{13} c_{i,j}$$

be the corresponding twist of each such sample signature. If our key length conjecture is correct, then each $\diamondsuit C_j$ should be relatively large, since each such coset should approximately model the frequencies of the underlying language. This all motivates the twist algorithm definition introduced by Barr-Simoson [1].

Definition 2.1. Let $\mathcal{M}$ be a text of length N. The **twist algorithm** finds $m \in \mathbb{N}^+$ that maximizes the **twist index**

$$T(\mathcal{M},m) = \Big(\frac{100}{m}\sum_{j=1}^{m} \diamondsuit C_j\Big).$$

While the twist algorithm provided a good first step towards a new key length attack, it does have some significant flaws. The biggest issue is the fact that the twist index is increasing as a function of multiples of the actual key length k. This fact is proven for certain cases and verified experimentally for all other cases in [9]. As a result, the twist index will always make an incorrect prediction, assuming nontrivial multiples of the key length are part of the domain of $T(\mathcal{M},m)$ for a fixed $\mathcal{M}$. This flaw inspired Park–Kim–Cho–Yum to design the twist$^+$ algorithm in [12].

Definition 2.2. Let $\mathcal{M}$ be a text of length N. The **twist$^+$ algorithm** finds $m \in \mathbb{N}^+$ that maximizes the **twist$^+$ index**

$$T^+(\mathcal{M},m) = T(\mathcal{M},m) - \frac{1}{m-1}\sum_{\mu=1}^{m-1} T(\mathcal{M},\mu).$$

Park–Kim–Cho–Yum provide experimental evidence highlighting the twist$^+$ algorithm as far more successful than both the index of coincidence and the twist algorithm for a variety of parameters; see Figure 3 in [12]. However, they also highlight the fact that the twist$^+$ algorithm does become less effective when short key lengths were used on relatively short texts. Furthermore, it is unclear what domain of m-values are considered for maximizing both the twist index and the twist$^+$ index in these definitions. In particular, if one increases this domain, then the twist$^+$ algorithm decreases in success, as highlighted in Figure 1 and Figure 2 of [9]. Similar to the twist algorithm, there is an issue with the twist$^+$ algorithm predicting a multiple of the key length, though under more specialized parameters. In hopes of constructing a twist-based algorithm that won't predict multiples of the key length and maintain a high level of accuracy even for large domains of m-values, Millichap-Yau–Pate–Carns introduced the twist^{++} algorithm in[9]. This algorithm finds the m-value that maximizes a local change in twist index.

Definition 2.3. Let $\mathcal{M}$ be a text of length N. The **twist^{++} algorithm** finds $m \in \mathbb{N}^+$ that maximizes the **twist^{++} index**

$$T^{++}(\mathcal{M},m) = T(\mathcal{M},m) - \frac{1}{2}\Big(T(\mathcal{M},m-1) + T(\mathcal{M},m+1)\Big),$$

where $m \in S \subseteq \{2,\ldots,q\}$ and $N = 12q + r$ for quotient q and remainder r.

Note, this third definition highlights the need to specify a domain of potential key lengths to check. In particular, the value q is set as a maximal m-value one should consider since for $m > q$, we have $T(\mathcal{M},m) = 100$. Thus, twist (and twist$^+$ and twist^{++}) indices will not

provide any useful information for such values. Statistics from Figure 1 and Figure 2 in [9] show that the twist^{++} algorithm performs exceptionally well under a variety of parameters, including ones where the twist^{+} algorithm drops in accuracy. While the twist^{++} algorithm is the most accurate twist-based algorithm for most parameters (variety of key lengths and text lengths), there are still some specialized conditions under which the twist^{++} algorithm might predict the largest nontrivial divisor of the actual key length; see Section 3 of [9] for a discussion on this and examples.

2.4. **Other Tests.** Here, we briefly discuss a few other statistical and quantitative tools that can assist with key length attacks. These tools were also tested as potential features for our ANNs discussed in Section 3.

In [5], Matthews introduced two basic tools, H and Δ, that are functions of frequencies of individual letters in a ciphertext and both are highly correlated with the key length k. Given a text $\mathcal{M}$, the function $H(\mathcal{M})$ sums the percentage frequencies of the seven most common letters in $\mathcal{M}$, and the function $\Delta(\mathcal{M})$ is the difference between the sum of the percentage frequencies of the seven most common letters in $\mathcal{M}$ and the sum of the percentage frequencies of the seven least common letters in $\mathcal{M}$. Matthews performed a regression analysis that showed a linear relationship between H and k and a linear relationship between Δ and k. While Matthews highlighted H and Δ as improvements over the IC, both of these tools were only applied to a specific set of key lengths ($k \in \{3,5,9,13,17,21\}$) and accuracy rates were still quite low.

One other tool that could assist with key length attacks is (information) entropy, which measures the amount of "information" in a text and was introduced by Claude Shannon in 1948 [14]. The first order entropy of a text $\mathcal{M}$ is defined as

$$H_1(\mathcal{M}) = -\sum f_i \log_2(f_i),$$

where f_i is the relative frequency of the i^{th} letter from the underlying alphabet. Higher order entropies can also be calculated by considering frequencies of digrams, trigrams, etc. With regards to key length attacks, information entropy could serve a similar role to the IC, as both are functions of the individual frequencies of a text and ciphertexts encrypted with varying lengths will have varying frequency distributions. We refer the reader to Chapter 11 of [2] for a further introduction to information entropy.

3. A Neural Networks Approach to Key Length

The discussion in the previous sections illustrates that existing techniques have various strengths and weaknesses depending on a variety of parameters. Motivated by our key question in the introduction, a neural network is a natural candidate as a method to combine the existing techniques into one key length finding structure such that the strengths of one technique could potentially compensate for weaknesses in another. For example, by the discussion in Section 2.3 using both twist^{+} and twist^{++} or using multiple tests to potentially pick the correct key length from it's multiples. In this section, we first give some brief background on Artificial Neural Networks (ANNs) in 3.1 and then discuss the specifics of our ANN in 3.2, 3.3 and 3.4.

3.1. **Background on Neural Networks.** We first give a brief overview of Feedforward Neural Networks (FFNN) and introduce some of the essential terminology for our work. We direct the reader to [11] and [3] for further background on ANNs.

A Feedforward Neural Network is a machine learning framework inspired by biological neural networks existing in animal brains. They are both the simplest structure and basis of many machine learning architectures in the class of "deep learning" algorithms.

A FFNN is often viewed as a directed, acyclic graph with a number of *layers*. Each vertex of the graph is called a *neuron* and is edge-connected to each neuron in the previous and subsequent layer. To each neuron n we associate two numbers: a_n, the *activation* of neuron n and a number b_n called a *bias*. To each edge we associate a number called a *weight*, which represents the strength of the connection from one neuron to another. The activation of a neuron n, a_n is a function of the activations a_i associated to the neurons in the previous layer connected to neuron n, the weights w_i associated to those connections and the bias b_n:

$$a_n = h(w_1 a_1 + w_2 a_2 + \ldots + w_k a_k + b_n) \tag{3}$$

where h is called an *activation function* and usually has range $[0,1]$. A number of activation functions are used in practice and we refer the reader to [3, Chapter 6] for further details about these functions. Figure 1 illustrates a FFNN with input layer consisting of four neurons (this is the left most layer), two hidden layers each with five neurons and an output layer consisting of three neurons.

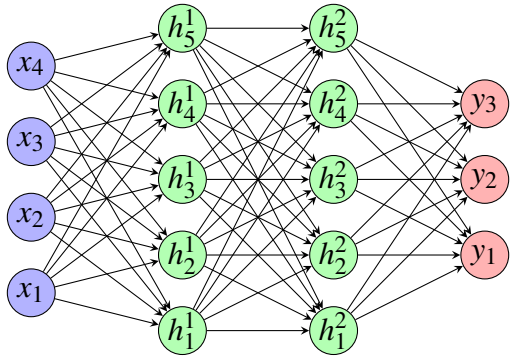

FIGURE 1. Example of a Feedforward Neural Network

A FFNN which acts as a classifier can then be seen as a function $f(x;\mathbf{W},\mathbf{B})$ with x an input vector of "features" and parameterized by it's weights $\mathbf{W}$ and biases $\mathbf{B}$. For a given a set of weights and biases, when the FFNN is provided an input vector of features, all activations are computed by the formula of Equation 3. The output of f is a vector y of activations of the neurons in the output layer, where the entry y_i is the probability that the FFNN believes the input x should be assigned to category i. Training the FFNN is the process of determining the optimal weights and biases for doing the job of correctly classifying the inputs for a particular problem, by exposing the network to correctly labelled training examples and evaluating the predictions of the network via a *cost function*. We refer the

reader to [3, Chapter 6]) for further details about training a neural network, including various cost functions. In practice, determining the architecture and weights **W** (training) of a FFNN is a process of experimenting with different features, activation functions, cost functions, number of layers and number of neurons in each layer.

In this work, we train a FFNN to classify the key length of a Vigenère ciphertext given an input vector of features calculated from the text. We discuss the exact features, activation functions, cost functions, number of hidden layers and neurons used and tested in this work in Section 3.3.

3.2. **Data Generation for Model.** The data for this project was obtained from the Project Gutenberg website [4], an online library of free books. Approximately 5,500 English text files were downloaded, a subset which were then systematically parsed and cleaned (removing numbers, punctuation and spaces). The cleaned text files were then split into non-overlapping texts of length 200-500. To ensure uniformity of training data, each length i, for $200 \leq i \leq 500$, approximately 1,300 samples were generated. Each sample was then encrypted with random keywords with lengths varying from 3 to 25 characters. The keywords include English words and phrases as well as random strings of letters. Keywords which are English words and phrases were randomly selected from the WordNet database in the Natural Language Toolkit (NLTK) package in Python.

A selection of features based on the key length attacks described in Section 2 were then computed for each sample and saved. We discuss the details of these features in Section 3.3. Our FFNN was trained on 332,605 samples and then tested on a test set (unseen by the model during the training process) of size 58,695. To do these computations, we used the Tensorflow and Keras libraries in Python.

3.3. **Creating a Model.** We considered a number of features for our ANN based on the techniques discussed in Section 2. The complete list of features we considered in this work are highlighted in Table 1.

Let us discuss our rationale for the consideration of these features. Naturally, the experimentally verified effectiveness of the twist^{+} and twist^{++} algorithms in finding the key lengths under various conditions warranted inclusion of features (6), (7) and (8) as features in the network. However, as discussed earlier, when the key length exceeded $N/12$ these tests could not be used. Thus, we included features (5) and (9) to give the network at least some information when this occurred.

Initial investigations also showed that when repeated trigrams or quadgrams where present, the Babbage-Kasiski test generally yielded useful information. In addition, the Babbage-Kasiski method does not rely on comparing the ciphertext with statistical properties of the English language and hence the inclusion of features (2), (10) and (11). Features (12)-(14) were considered as other possible pieces of information related to the key length as discussed in Section 2.4.

In order to investigate the relative impact of various subsets of the above features, a standard architecture for the models was initially chosen. We refer to this as the *base NN*. This architecture included an input layer of features coming from Table 1, two hidden layers of 128 neurons each and an output layer of 23 neurons corresponding with key lengths of 3 to 25 characters. We chose ReLu for the activation functions of the hidden layers, and

(1) Length of ciphertext
(2) Has repeated sequences
(3) Index of coincidence of the ciphertext
(4) Index of coincidence of English (constant, which is 0.066)
(5) The quotient q from $N = 12q + r$ where N is the length of the ciphertext.
(6) The twist indices: $T(\mathcal{M}, m)$ for $1 \leq m \leq 25$
(7) The twist$^+$ indices: $T^+(\mathcal{M}, m)$ for $2 \leq i \leq 25$
(8) The twist^{++} indices: $T^{++}(\mathcal{M}, m)$ for $2 \leq i \leq 25$
(9) The average index of coincidence for cosets m where $3 \leq m \leq 25$
(10) The 5 most common distances that occurred between repeated sequences of length 3 or 4.
(11) The number of times each of the top 5 most common distances appeared.
(12) Hi-7
(13) $\Delta - 7$
(14) First order entropy

TABLE 1. Features considered

Softmax for the output layer. Our optimizer and loss functions were chosen to be "Adam" (Adaptive Moment Estimation) and Categorical cross-entropy respectively. These functions were chosen based on fairly standard choices (see [3, Chapter 6]) for categorization problems. Our training process involved 10 epochs (passes through the training data) and validation test set sizes of 20% per epoch.

An iterative approach was used to engineer the features of our final neural network. We initiated our investigation of the features by starting with our *base NN* and including all features listed above in Table 1 in Model 1. Based on an analysis of the accuracy of the model during each epoch of the training process and the accuracy of the model on the unseen test set, we iterated through models by systematically removing each feature (or in some cases, a pair of features) from our original list. At each iteration, if the removal of a feature increased accuracy, further features were removed in the following iteration. When the accuracy rate decreased, we returned those features to the model and removed different features in the next iteration. The results of the feature engineering process are recorded in Table 2. The numbers appearing in the "Input features" column refers to the numbering of the features in Table 1.

The feature engineering process is further summarized in Table 3 below, which displays the effect of leaving out each feature (in the order in which they were removed) on the accuracy rate of the neural network. From Table 2 and Table 3 we see that the best performing model was Model 9, with the following features removed from our original complete feature list: first order entropy, quotient, the Babbage-Kasiski features ((10) and (11) from Table 1) and the average of the index of coincidences for m-cosets.

Subsequently, we proceeded to further engineer Model 9 by adding back into the model some of the removed features in a different order to test whether different subsets of the removed features could increase model performance. From our experiments, we concluded

Model	Input features	Number of Features	NN Accuracy on unseen test data
Model 1	All features	114	88.8%
Model 2	All features except 12 and 13	112	88%
Model 3	All features except 14	113	88.2%
Model 4	All features except 2 and 14	112	87.6%
Model 5	All features except 14, 3 and 4	111	87.4%
Model 6	All features except 14, 5	112	87.7%
Model 7	All features except 14, 5, 10 and 11	102	88.3%
Model 8	All features except 14, 5, 10, 11 and 9	79	88.7%
Model 9	All features except 14, 5, 9, 10, 11 and 6	54	88.9%
Model 10	All features except 14, 5, 9, 10, 11, 6 and 7	30	87.6%
Model 11	All features except 14, 5, 9, 10, 11, 6 and 8	30	87.2%

TABLE 2. Model features and accuracy rates

that adding the average index of coincidences for the m-cosets back into the model increased the accuracy rate to 89.2%. None of the other removed features were able to further improve the performance of Model 9.

In the final step to create our model, we experimented with increasing the number of epochs to 20 during the training process and including an additional layer with 128 neurons in the network. This did not make a significant difference to the accuracy rate.

Our final model architecture is as follows: Input layer with 77 features, two hidden layers each with 128 neurons and an ouput layer of 23 neurons. The activation functions are ReLu for the hidden layers, and Softmax for the output layer. Our optimizer and loss functions are "Adam" (Adaptive Moment Estimation) and Categorical cross-entropy respectively. The final features included in the neural network are highlighted in Table 4.

3.4. **Evaluation of Model.** From Table 5 we see that our neural network is a vast improvement over the index of coincidence (using the formula from [2, Chapter 3]) and the

Model amended	Feature removed	Effect on accuracy rate after removal
Model 1	Hi-7 and $\Delta - 7$	-0.8%
Model 2	First order entropy	+0.2%
Model 3	Has repeated sequences	-0.6%
Model 4	Index of coincidence of the ciphertext and English	-0.2%
Model 5	The quotient q from $N = 12q + r$ where N is the length of the cipher text.	+0.3%
Model 6	10 & 11 (Babbage-Kasiski features)	+0.6%
Model 7	The average index of coincidence for cosets m where $3 \leq m \leq 25$	+0.4%
Model 8	Twist indices: $T(\mathcal{M}, m)$ for $1 \leq m \leq 25$	+0.2%
Model 9	Twist$^+$ indices: $T^+(\mathcal{M}, m)$ for $2 \leq i \leq 25$	-1.3%
Model 10	Twist^{++} indices: $T^{++}(\mathcal{M}, m)$ for $2 \leq i \leq 25$	-0.4%

TABLE 3. Feature importance

(1) Length of ciphertext
(2) Has repeated sequences
(3) Index of coincidence of the ciphertext
(4) Index of coincidence of English (constant, which is 0.066)
(5) The twist$^+$ indices: $T^+(\mathcal{M}, m)$ for $2 \leq i \leq 25$
(6) The twist^{++} indices: $T^{++}(\mathcal{M}, m)$ for $2 \leq i \leq 25$
(7) The average index of coincidence for cosets m where $3 \leq m \leq 25$
(8) Hi-7
(9) $\Delta - 7$

TABLE 4. Final model features

Text-Length/Method	IC	Twist Index	T^+	T^{++}	Neural Network
200-500 (Overall accuracy)	7.4%	22.4%	66.2%	63.3%	89.2%
200-299	6.8%	16.1%	47.3%	42.3%	75.1%
300-399	7.6%	20.3%	71.7%	65%	94.6%
400-500	7.7%	30.6%	79.4%	82.2%	97.9%

TABLE 5. Accuracy rates by key length finding method

twist-based algorithms. In particular, it is able to maintain high accuracy when the ratio of text length to key length is large.

From our original feature list in Table 4, (14) first order entropy, (5) the quotient q from $N = 12q + r$,(6) the twist indices: $T(\mathcal{M}, m)$ for $1 \leq m \leq 25$ and (10) and (11) the Babbage-Kasiski features, did not appear to add any value to the neural network, as evidenced by the slight increase in accuracy rate when those respective features were removed. It is perhaps somewhat surprising that the Babbage-Kasiski features were relatively unimportant given that 99% of the samples contained at least one repeated trigram. One possible explanation could be that to determine the key length from the Babbage-Kasiski data requires an extra layer of intelligent processing to sift through the possible multiples of the key length given in the data and then determine the actual key length from all the possible factors. The twist indices also did not provide useful information in this experiment, which is not surprising given it's overall accuracy rate and the fact that the twist$^+$ and twist^{++} indices already implicitly include this information.

As a rough measure of feature importance, we observed that the removal of the twist$^+$ data caused the biggest decrease in accuracy (-1.3%). We also experimented with removing the twist^{++} data at that step of the engineering process instead, and this yielded a decrease of -1.2% in accuracy. This suggests that both the twist$^+$ and twist^{++} provide strong predictors of key length.

We now discuss the accuracy of the twist-based algorithms in relation to our neural network. In our investigations, the twist$^+$ was slightly more accurate than the twist^{++} for longer key lengths, while the twist^{++} was more accurate for shorter key lengths. However, it should be noted that our domain of m-values for the twist-based algorithms were restricted to the possible known values of the key length. Since $T(\mathcal{M}, m) \leq T(\mathcal{M}, \lambda m)$ for $\lambda \in \mathbb{N}$ (originally stated in [12], with a special case proved in [9] along with experimental data verifying this result for the other cases), if a larger range of m-values were considered then we would expect a decline in accuracy in the twist and twist$^+$ algorithms, but relatively steady accuracy for the twist^{++}. In general, the twist and twist$^+$ algorithms frequently predict a multiple of the actual key length when making an incorrect prediction, while the twist^{++} frequently predicts the largest common divisor of the actual key length when making a false prediction. Thus, some of the accuracy ratings in Figure 3, Figure 4, and Figure 5 for the twist and twist$^+$ algorithms could be overestimates for longer key lengths. In addition, this might mean our neural network's accuracy is slightly overestimated for longer key lengths since the twist$^+$ indices are a feature of this network. We refer the reader to [9] for a further discussion about the effect of domain size on the accuracy of the twist$^+$ and twist^{++} algorithms and most common scenarios for incorrect predictions for these algorithms.

For shorter texts $200 \leq N \leq 299$ it can be observed from Figure 3 that the twist-based algorithms perform very poorly for longer key lengths, but the neural network gives some chance of success. This is due to the fact that each coset in the twist algorithm contains too few letters, resulting in twist indices all being close to 100 (see [9] for further discussion). It is interesting to see that our ANN performs reasonably well under these conditions despite the fact that many of the key features of this ANN individually perform quite poorly under such conditions.

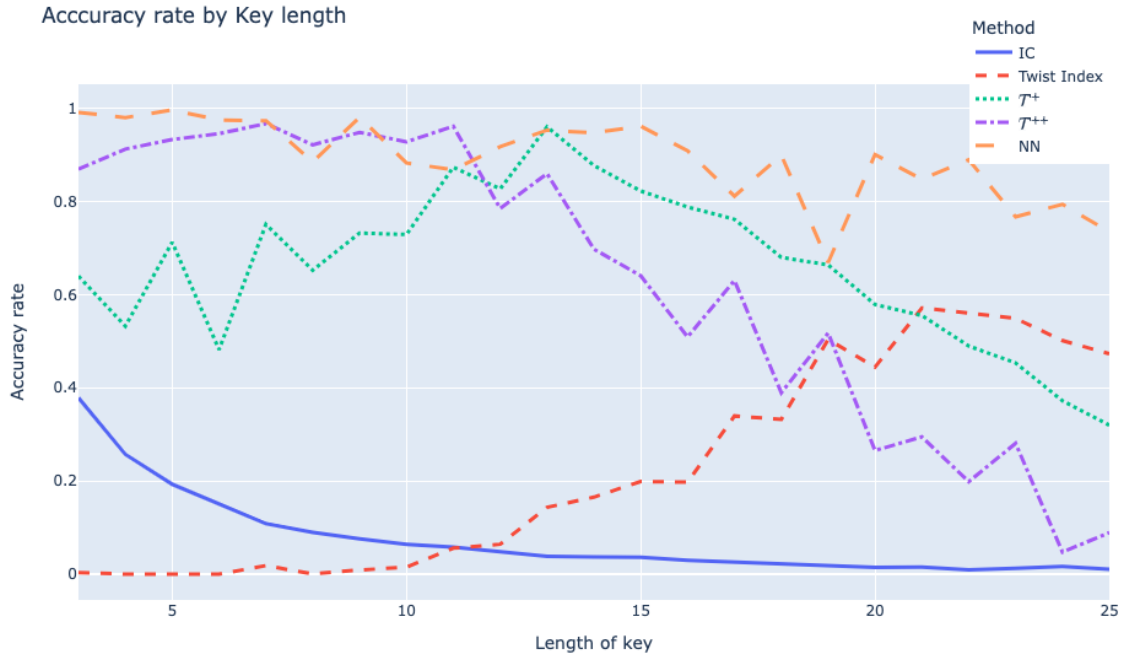

FIGURE 2. Accuracy rates by key length finding method

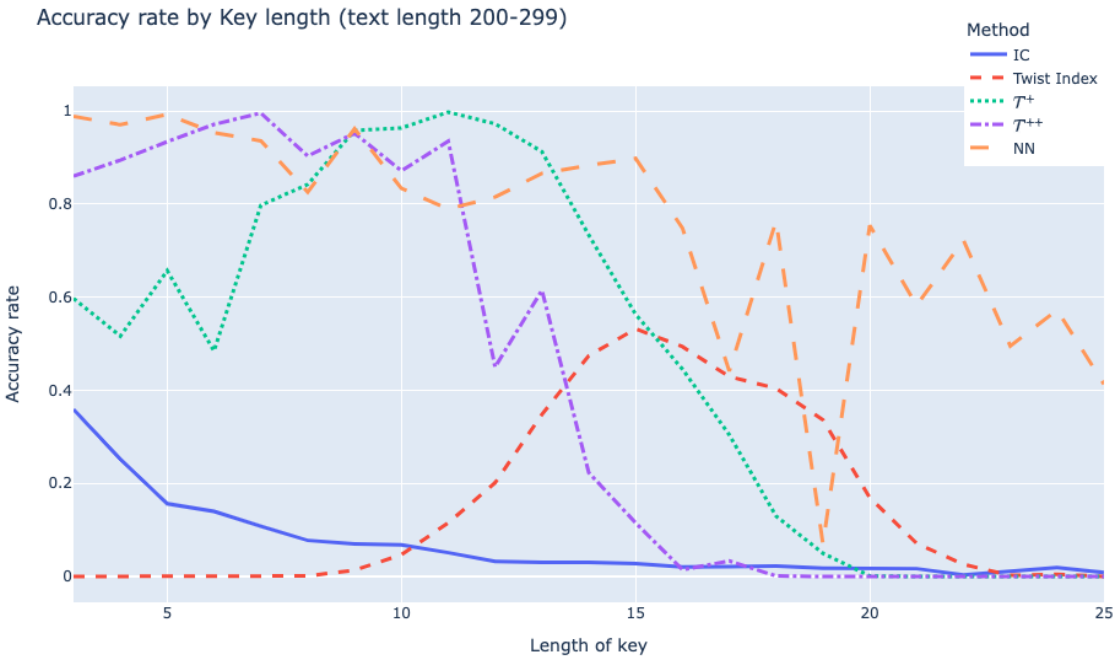

FIGURE 3. Accuracy rate by key length for $200 \leq N \leq 299$

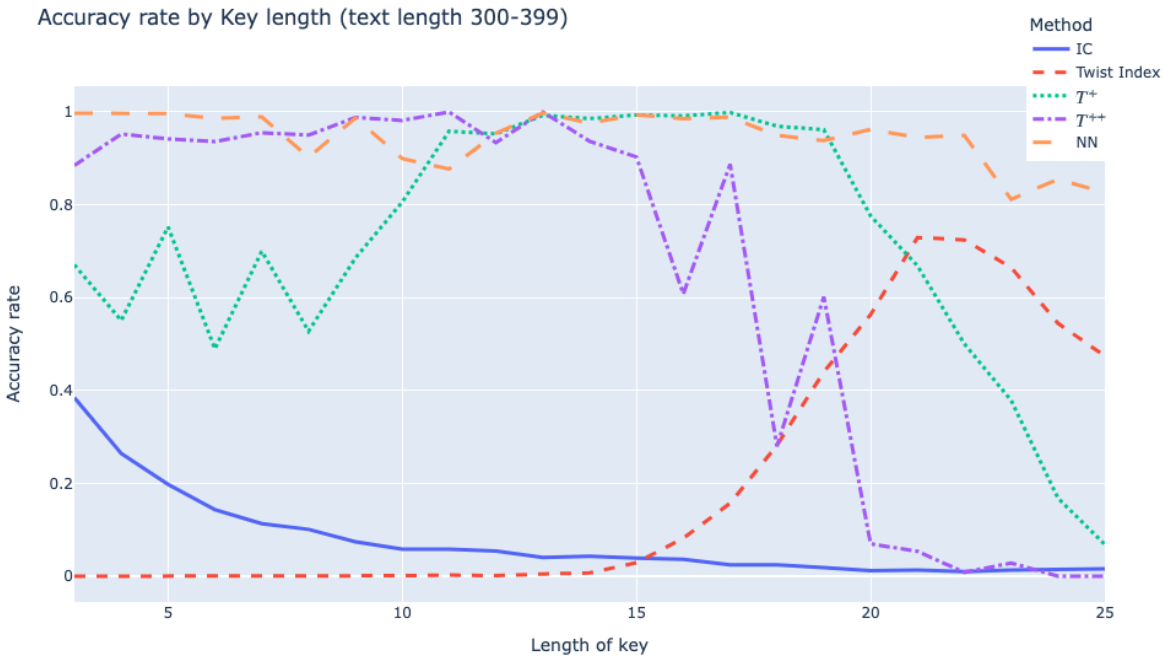

FIGURE 4. Accuracy rate by key length for $300 \leq N \leq 399$

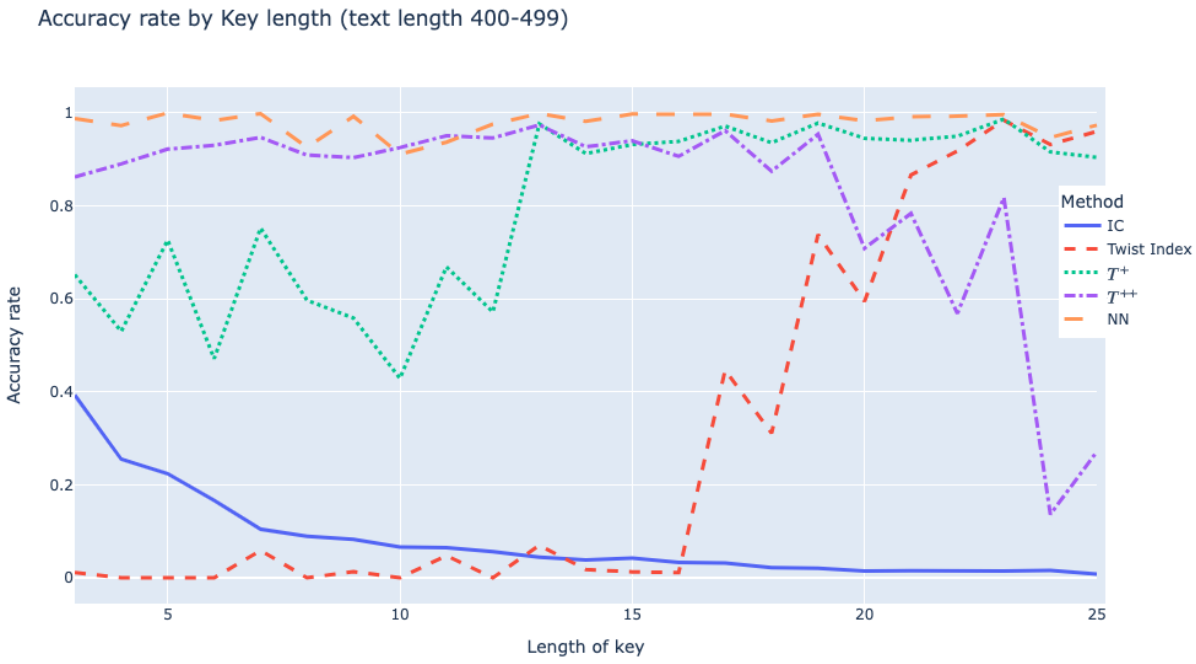

FIGURE 5. Accuracy rate by key length for $400 \leq N \leq 499$

For longer text lengths $400 \leq N \leq 500$ we see from Figure 5 that the twist$^+$ and twist^{++} algorithms are both quite accurate and complement each other in terms of key length, with twist^{++} performing better for shorter key lengths and twist$^+$ for longer key lengths. The neural network seems to be able to combine this information and is extraordinarily accurate in these circumstances with an accuracy rate of 97.9%.

3.5. **Summary.** In this project we investigated the possibility of combining both classical and recent techniques into one key-length finding algorithm via a neural network. We demonstrate that neural networks can be a powerful tool for predicting the key length of Vigenère encrypted text.

We observed that our original model with 114 features was a significant improvement in performance from the twist-based algorithms, and overall, the feature engineering experiment did not make a significant improvement to the success rate of the model, providing some indication that appropriate features were originally chosen.

We were able to feature engineer the network to reduce the number of features in half whilst slightly improving the accuracy of the network, to an overall success rate of 89.2%. In particular, the neural network is much more accurate than existing methods in predicting key length when the ratio of text length to key length is large. This project also revealed that the recent twist-based algorithms, the twist$^+$ and twist^{++} algorithms, provided some of the strongest indicators of key length.

4. Biographical Note

Christian Millichap is an Associate Professor of Mathematics at Furman University in Greenville, SC. His research interests are in geometric topology and knot theory. He has also enjoyed teaching a variety of classes in cryptology for high school students and undergraduates.

Yeeka Yau is an Assistant Professor of Mathematics at the University of North Carolina Asheville. His research interests are in Coxeter groups, combinatorial and geometric group theory, cryptology and machine learning.

5. Data Availability Statement

The data that support the findings of this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.10363051.

6. Acknowledgements

This work was financially supported by the Furman University Department of Mathematics via the Summer Mathematics Undergraduate Research Fellowships.

References

[1] Thomas H. Barr and Andrew J. Simoson, *Twisting the keyword length from a vigenère cipher*, Cryptologia **39** (2015), no. 4, 335–341, https://doi.org/10.1080/01611194.2014.988365.

[2] Craig P. Bauer, *Secret history—the story of cryptology*, second ed., Chapman & Hall/CRC Cryptography and Network Security, CRC Press, Boca Raton, FL, 2021.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, http://www.deeplearningbook.org.
[4] Michael S. Hart, *Project gutenberg*.
[5] Matthews R. A. J., *An empirical method for finding the key length of periodic ciphers*, Cryptologia **12** (1988), 220.
[6] D. Kahn, *The codebreakers: The comprehensive history of secret communication from ancient times to the internet.*, Scribner, New York, NY, 1996.
[7] F.W. Kasiski, *Geheimschriften und die dechiffrirkunst*, Mittler und Sohn (1863).
[8] Esslinger B. Lampesberger H. Hermann E. Leierzopf E., Kopal N., *A massive machine-learning approach for classical cipher type detection using feature engineering*, Proceedings of the 4th International Conference on Historical Cryptology, HistoCrypt 2021 (2021), no. 183, 111–120.
[9] Christian Millichap, Yeeka Yau, Alyssa Pate, and Morgan Carns, *Modifying twist algorithms for determining the key length of a vigenère cipher*, Cryptologia (2023), 1–16, https://doi.org/10.1080/01611194.2023.2275583.
[10] Kopal N., *Of ciphers and neurons detecting the type of ciphers using artificial neural networks*, Proceedings of the 3rd International Conference on HIstorical Cryptology, HistoCrypt 2020 (2020), no. 171, 77–86.
[11] Michael Nielsen, *Neural networks and deep learning*, 2019, August 9th 2023.
[12] Seongmin Park, Juneyeun Kim, Kookrae Cho, and Dae Hyun Yum, *Finding the key length of a vigenère cipher: How to improve the twist algorithm*, Cryptologia **44** (2020), no. 3, 197–204, https://doi.org/10.1080/01611194.2019.1657202.
[13] Klaus Pommerening, *Kasiski's test: Couldn't the repetitions be by accident?*, Cryptologia **30** (2006), no. 4, 346–352, https://doi.org/10.1080/01611190600803819.
[14] C. E. Shannon, *A mathematical theory of communication*, Bell System Tech. J. **27** (1948), 379–423, 623–656.
[15] Friedman W.F., *"the index of coincidence and its application in cryptography,"*, Riverbank Laboratories Deparmtent of Ciphers Publication **22** (1920).

DEPARTMENT OF MATHEMATICS, FURMAN UNIVERSITY, GREENVILLE, SC 29613
Email address: christian.millichap@furman.edu

DEPARTMENT OF MATHEMATICS & STATISTICS, UNC ASHEVILLE, ASHEVILLE, NC 28804
Email address: yyau@unca.edu