National College of Ireland

# RSAnalyzer: Cryptanalyzing RSA using Supervised Machine Learning Methods

MSc Research Project

M.Sc. Cybersecurity Top-up


Gary Crowe

Student ID: 23182652

School of Computing

National College of Ireland


Supervisor: Mark Monaghan

**National College of Ireland**

**MSc Project Submission Sheet**

**School of Computing**

| | | | |
|---|---|---|---|
| **Student Name:** | Gary Crowe | | |
| **Student ID:** | 231825652 | | |
| **Programme:** | M.Sc. Cybersecurity Top-up | **Year:** | 1st |
| **Module:** | MSc Research Practicum Part 2 | | |
| **Supervisor:** | Mark Monaghan | | |
| **Submission Due Date:** | 11/08/2025 | | |
| **Project Title:** | RSAnalyzer – Cryptanalyzing RSA with Supervised Machine Learning Methods. | | |
| **Word Count:** | ~6500 | **Page Count**. | 20 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | Gary Crowe |
| **Date:** | 10/08/2025 |

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST**

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies) | □ |
| **Attach a Moodle submission receipt of the online project submission,** to each project (including multiple copies). | □ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | □ |

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# RSAnalyzer: Cryptanalyzing RSA using Supervised Machine Learning Methods

Gary Crowe

23182652

**Dedication and acknowledgements**

*To my brother Darren, for being at the right place at the right time and because of that now I have my degree, and I know everything.*

# Contents

1 Introduction....8
2 Related Work....9
2.1 Machine Learning in Cryptanalysis: State of the Art....9
2.2 The RSA Cryptosystem....9
2.3 Potential Attacks on RSA....9
2.3 Research Niche....10
3 Research Methodology....10
3.1 Data Acquisition/Generation....10
3.2 Evaluation Metrics....10
4 Design Specification....11
5 Implementation....13
5.1 Dataset Creation....14
5.2 Machine Learning Model Creation....18
5.3 RSAnalyzer....21
Analyze....21
Encrypt....22
6 Evaluation and Discussions....22
6.1 Experimental Setup....23
6.2 Decision Tree Classifier....24
6.3 Random Forest Classifier....24
6.4 Support Vector Machine Classifier....25
6.5 Neural Network Classifier....26
6.6 Misclassification of Weakness....27
7 Conclusion....27
References....27

**Abstract**

This work aimed to explore the cross-section of cryptanalysis and supervised machine learning, to determine whether machine learning techniques can be used to detect weaknesses in asymmetric cryptosystems, RSA in this case. RSA was chosen for this research as it is the most widely used asymmetric cryptosystem in the world. Its position as a commonly used cryptosystem in e-commerce transactions highlight the need to use all tools available to analyse and break it.

This research aimed to show supervised machine learning methods can be used to break RSA ciphertext not only in theory but also in practice by using the findings of this research to create a software artefact called RSAnalyzer. The aim of this tool is taking a ciphertext and public key as input, and output the original plaintext based on the weakness found in the ciphertext.

The detection of the weakness will be performed by machine learning models trained to detect one of 4 weakness in RSA's parameter selection process. Evaluation of the supervised models showed a strong indication that supervised machine learning techniques can reliably detect weaknesses in RSA.

# 1 Introduction

RSA is the most widely used asymmetric cryptosystem in the world. Modern e-commerce could not exist without asymmetric ciphers, like RSA. It is important that a system that is such wide use should be examined with every tool available to a potential threat actor.

It seems that recent years that many of the research in using machine learning in cryptanalysis has been to focus on quantum cryptography, which is a very important area to research as the field of quantum computing is still developing. Even when more traditional cryptosystems are cryptanalyzed with machine learning they seemed focused on symmetric ciphers rather than asymmetric ones.

The research question of this thesis is: *Can supervised machine learning models detect known weaknesses or RSA?*

To this end this work aims to create a software artefact called "RSAnalyzer". This program once complete will be able to take an inputted ciphertext, detect what weakness it has and then exploit that weakness to retrieve the original plaintext.

As this work is a proof of concept, the implementation of RSA will be a standard textbook implementation with no modes of operation. This work will also only focus on known weakness of RSA rather than trying to discover new ones. The foundation laid out here is intended to be built upon in future work.

This work is divided up into several sections which include, Related Work, Research Methodology, Design, Implementation, Evaluation and Conclusion and Future Work.

Related work lays down current state of the literature involving the intersection of machine learning and cryptanalysis. It also details the weaknesses which be explored throughout this work.

Research Methodology details the metrics used to evaluate the supervised machine learning models used by RSAnalyzer

Design discussed the overall structure of RSAnalyzer from an algorithmic perspective and provides several pieces of pseudocode for major functionality.

Implementation discusses the tools and technologies used to implement the pseudocode from the Design section into a working piece of software.

The Evaluation section then applies the metrics discussed in the Research Methodology section to several machine learning models.

Finally, the Conclusion and Future work section discusses what was learned from this research, its successes and limitations as well as what future research in this area would aim to do.

# 2 Related Work

## 2.1 Machine Learning in Cryptanalysis: State of the Art

In February 2024, a paper was released detailing the state of art of the use of Neural Networks in Cryptanalysis (Ajeet, et al., 2024). This paper focuses on advancements in cryptanalysis of symmetric ciphers using Stochastic Optimisation Methods and Neural Distinguishers. The paper deals with two sub-classes of symmetric cipher, block and stream ciphers.

Many of the papers found on the topic of machine learning in cryptanalysis such as (Tolba, et al., 2022) make mention of asymmetric approaches but then go on to mostly discuss symmetric cryptosystems. This lack of exploration of machine learning methods in the cryptanalysis of asymmetric ciphers, has made finding relevant peer reviewed literature difficult. There have been some papers found that do explore the public-key cryptosystems. (Yaseen & Sahasrabuddhe, 1999) discusses the use of a genetic algorithm for the cryptanalysis of the Chor-Rivest cryptosystem for a conference paper back in 1999.

While searching for relevant papers for this research, it became evident that much of the research community seems to be focused on researching quantum cryptography (Ye, et al., 2022). While developing and analysing cryptosystems for quantum computers is important, at time of writing quantum computers are not in widespread use. So, some attention should be paid to cryptosystems like RSA that are still in widespread use.

## 2.2 The RSA Cryptosystem

The RSA cryptosystem has two components (Stallings, 2017), an encryption component described by Equation 2.2.1 and a decryption component described by Equation 2.2.2

$$C = M^e \; mod \; n \quad \text{(Eq. 2.2.1)}$$

$$M = C^d \; mod \; n \quad \text{(Eq. 2.2.2)}$$

Where $n$ is the product of 2 large and distinct primes, $p$ and $q$ and

$$ed \; mod \; \varphi(n) = 1 \text{ (Eq. 2.2.3)}$$

where $\varphi(n)$ is the Euler Totient function of $n$.

## 2.3 Potential Attacks on RSA

RSA can have several weaknesses because of poor parameter selection (Stallings, 2017). Most of the attacks here focus on the factorisation of $n$ to obtain $p$ and $q$. Once this is done then computing $d$ is trivial as an attacker would already have $e$ and $n$ as part of the public key. Here some of the possible attacks on RSA because of this are discussed:

*p and q are not primes*

The fact that $p$ and $q$ are not primes makes factorisation relatively easy to do. All an attacker would have to do is keep guessing possible factors of $n$ until the correct values are found.

*p and q are equal*
This issue is easier than the previous vulnerability as all an attacker would have to do is take the square root of *n* to obtain *p* and *q*.

*p and q are not "large" primes*
In this scenario, all an attacker would have to do is attempt to factorize *n* using "small" primes until the correct answer is found

*p and q are "close"*
When two given primes are "close", this means that there is not a large difference between them. For example, say *p* is the *nth* prime and *q* is the *(n+1)th* prime. This weakness can be exploited using a method called Fermat Factorization. This method even allows for the factorization of *n* even when "large" values are used for p and q if they are "close" enough.

## 2.3 Research Niche

It is clear from the review the relevant literature that the idea of using machine learning techniques on asymmetric ciphers has gone relatively unexplored up until this point. None of the papers found discussed asymmetric or public-key cryptosystems, nor mentioned it as a future avenue of exploration. This is only speculation but there may be an assumption among researchers in this area that cryptanalyzing asymmetric ciphers using machine learning techniques is not currently feasible. If this is the case then, it is worth investigating this assumption and proving it one way or the other.

# 3 Research Methodology

## 3.1 Data Acquisition/Generation

To work used a combination of publicly available and synthetic data. The publicly available comprises a list of roughly 466k words in the English language. This wordlist is available through a GitHub repository, and the dataset is in the public domain under an Unlicense license (dwyl, 2018).

The rest of the data was generated by the author of this report. This data included 4 separate lists of prime numbers, one for each weakness mentioned above, 4 separate lists of ciphertexts and public keys, again one for each weakness mentioned above and one list of ciphertexts and public keys generated by randomly selecting entries from the previously mentioned lists to form the dataset that the classification models were trained and tested on.

The synthetic data has some restrictions on that are important to mention. First, that the dataset only uses in English with 7 characters or less. The reason for this is that the larger the plaintext that needs to be encrypted the larger the primes needed to encrypt them and longer the time it will take to generate datasets and to decrypt ciphertext.

Second is that the plaintext is limited to English meaning that the implementation can restrict itself to using ASCII codes for all characters.

## 3.2 Evaluation Metrics

The classification models in used here were evaluated using several metrics, accuracy, precision, recall and f1 score (Müller, 2016).

The accuracy metric is the simplest way to evaluate any classification model. It is simply the number of predictions a model made that were correct. While accuracy is not recommended for heavily imbalanced dataset, the dataset used to train/test the classifier is almost uniform (this was/can be verified by a python script), so this is not a concern here.

The recall or true positive rate calculates the proportion of positives that were correctly classified as positive. So, in this context this is the number of ciphertexts correctly classified with their assigned weakness.

The precision or false positive rate is a measure of the proportion of negatives that were incorrectly classified as positive. In a multiclass scenario this means that, if the precision calculated per class, then this is proportion of other weakness being incorrectly classed as the class in question.

# 4 Design Specification

This section discusses the algorithms used to exploit the weakness of a given ciphertext to decrypt it. This section will also go over the overall process of how RSAnalyzer operates at a high level.

After the user has input a ciphertext and a corresponding public key, the user can choose between 4 machine learning models to classify their input, a Decision Tree model, a Random Forest Model, a Support Vector Machine model or a Neural Network model. Once classified, RSAnalyzer performs a different decryption method based on the weakness. For a pseudocode description of this phase of RSAnalyzer see Algorithm 4.1.

**ALGORITHM 4.1: RSANALYZER**

***Input:*** *A ciphertext C and the corresponding public key (e, n)*
***Output:*** *A plaintext P and the corresponding private key (d, p, q)*

1. ***C and (e, n)*** *← Classify weakness based on ciphertext and public key*
2. ***Select Machine Learning model for classification***
3. ***Model Classifies weakness of input***
4. ***If*** *weakness is n uses small primes*, ***then***
5.     ***Small prime decryption*** *(Algorithm 4.2)*
6. ***If*** *weakness is n uses the same primes,* ***then***
7.     ***Decrypt n with the same prime for p and q*** *(Algorithm 4.3)*
8. ***If*** *weakness is n uses non-primes,* ***then***
9.     ***Decrypt n with non-primes for p and q*** *(Algorithm 4.4)*
10. ***If*** *weakness is n uses close primes,* ***then***
11.     ***Decrypt n with close primes for p and q*** *(Algorithm 4.5)*
12. ***End***

Algorithm 4.1 details the overall functionality of the RSAnalyzer. The classification step (Step 1) uses machine learning to identify what weakness a given ciphertext has. Since the decryption method is dependent on what weakness a given ciphertext has, each decryption algorithm will be detailed separately. It is important to note that that RSAnalyzer is built with the assumption that the given ciphertext has one of the weaknesses it exploits.

**ALGORITHM 4.2: *SMALL PRIME DECRYPTION***

***Input:*** *A ciphertext C and a public key (e, n)*
***Output:*** *The corresponding plaintext P*

1. ***Initialise Variables ←*** *assign p, q, phi, d, and m a value of 0*

**2** ***Factorise n into its two prime factors*** *← assign these factors to p and q respectively*
**3** ***Calculate Euler Totient Function of n*** *← assign output value to phi*
**4** ***Calculate modular inverse of e mod phi*** *← Assign output value to d*
**5** ***Decrypt C using RSA decryption formula*** *← assign output value to m*
**6** ***Initialise Variable*** *← assign b and P the value of the empty string*
**7** ***Convert m from decimal to binary*** *← assign output value to b*
**8** ***Convert b from binary to text*** *← assign this value to P*
**9** ***Return P***
**10** ***End***

In Algorithm 4.2 the factorisation step (Step 2) is done simply by iterating over all small primes $p_i$ and checking if n is divisible by $p_i$. Once a n is shown to be divisible by $p_i$, it becomes the value for p and factor resulting from dividing n by p becomes q. Since this weakness assumes that primes were used in calculating n, then all non-prime values can be ignored. For efficiency all small primes are precomputed and stored in a separate file.

Like Step 2, since it is assumed that n was the product of two primes. in Step 3 the Euler totient function is calculated using Equation 4.2.1 (Hardy, 2008).

$$(p-1) \times (q-1) \quad \text{(Eq. 4.2.1)}$$

---

**ALGORITHM 4.3: SAME *PRIME DECRYPTION***

***Input:*** *A ciphertext C and a public key (e, n)*
***Output:*** *The corresponding plaintext P*
**1** ***Initialise Variables*** *← assign p, q, phi, d, and m a value of 0*
**2** ***Calculate the square root of n*** *← assign output value to p*
**3** ***Calculate Euler Totient Function of n*** *← assign output value to phi*
**4** ***Calculate modular inverse of e mod phi*** *← Assign output value to d*
**5** ***Decrypt C using RSA decryption formula*** *← assign output value to m*
**6** ***Initialise Variable*** *← assign b and P the value of the empty string*
**7** ***Convert m from decimal to binary*** *← assign output value to b*
**8** ***Convert b from binary to text*** *← assign this value to P*
**9** ***Return P***
**10** ***End***

For Algorithm 4.3 the two primes used to construct n have the same value, then to find the value of these primes, only the square root of n needs to be calculated. This also changes how the Euler Totient function is calculated as the equation used to calculate it for Algorithm 4.3 assumes that p and q are distinct primes. To calculate the Euler Totient function of n where p and q are the same see Equation 4.4.1 or Equation 4.4.2 (Hardy, 2008).

$$p \times (p-1) \quad \text{(Eq. 4.3.1)}$$

$$q \times (q-1) \quad \text{(Eq. 4.3.2)}$$

---

**ALGORITHM 4.4: NON-*PRIME DECRYPTION***

***Input:*** *A ciphertext C and a public key (e, n)*
***Output:*** *The corresponding plaintext P*
**1** ***Initialise Variables*** *← assign p, q, phi, d, and m a value of 0*

2 ***Calculate Euler Totient Function of n*** ← *assign output value to phi*
3 ***Calculate modular inverse of e mod phi*** ← *Assign output value to d*
4 ***Decrypt C using RSA decryption formula*** ← *assign output value to m*
5 ***Initialise Variable*** ← *assign b and P the value of the empty string*
6 ***Convert m from decimal to binary*** ← *assign output value to b*
7 ***Convert b from binary to text*** ← *assign this value to P*
8 ***Return P***
9 ***End***

Since n is not made of primes here, this means that a given n could have many different factors. For example, consider all the non-prime factors of 100, {100,50,25,20,10,4,1}. How can it be determined what factors were used to produce the given n? The reason that n was factorised in the first place was so that then the Euler Totient function of n could be calculated easier using special cases of the function. So, the factorisation step was removed so which non-prime factors were used didn't have to be determined but the Euler Totient function of n could just be calculated directly using Equation 4.4.1 (Hardy, 2008).

$$\varphi(n) = n \prod_{p|n}(1 - \frac{1}{p}) \qquad \text{(Eq. 4.4.1)}$$

This approach voids having to figure out which factors were used to construct n but as a trade-off it does mean a more complex implementation is needed for Step 2.

**ALGORITHM 4.5: CLOSE *PRIME DECRYPTION***

***Input:*** *A ciphertext C and a public key (e, n)*
***Output:*** *The corresponding plaintext P*
1 ***Initialise Variables*** ← *assign p, q, phi, d, and m a value of 0*
2 ***Factorise n using Fermat Factorisation*** ← *assign output values to p and q*
3 ***Calculate Euler Totient Function of n*** ← *assign output value to phi*
4 ***Calculate modular inverse of e mod phi*** ← *Assign output value to d*
5 ***Decrypt C using RSA decryption formula*** ← *assign output value to m*
6 ***Initialise Variable*** ← *assign b and P the value of the empty string*
7 ***Convert m from decimal to binary*** ← *assign output value to b*
8 ***Convert b from binary to text*** ← *assign this value to P*
9 ***Return P***
10 ***End***

In Algorithm 4.6 as the primes are "close", Fermat factorisation was used for the factorisation step (Step 2). For each ciphertext encrypted the close prime weakness, each pair of close primes is the $n^{th}$ and $(n+1)^{th}$ prime. This means that using Fermat Factorisation it will take a relatively few iterations.

# 5 Implementation

This section follows on from the previous section by discussing the tools used, data and code created during the development of RSAnalyzer. Table 5.1 details the tools used in the creation and testing of this project.

## 5.1 Dataset Creation

The first code developed for RSAnalyzer was the code to generate the data the models would be trained on. The process for generating the training/testing dataset is detailed in Figure 5.2. Since there were 4 weakness in prime number selection for the RSA algorithm, the approach taken was to split each of the weak primes into 4 separate files. A file for small primes used for P and, a file for the same prime for p and q, a file for non-primes used for p and q and a file for close primes being used for p and q.

**Table 5.1.1: Tools Used and Purpose**

| Tool Name | Version | Purpose |
| --- | --- | --- |
| Python 3 | 3.11.7 | The version of Python that most of the code for this project was written in. |
| Anaconda | 24.11.3 | Needed for extra libraries such as sci-kit-learn and others |
| Jupyter-notebook | 7.0.8 | Required to read/run the files the models were trained in. |
| VSCode | N/A | Any version of any IDE will do here if you can run the code. VSCode is just what I used for this project, and it is what some of the example screenshots are taken from. |

The code that generated these files were also split into 4 different scripts:

- *gen_small_primes.py*
- *gen_same_primes.py*
- *gen_non_primes.py*
- *gen_close_primes.py*

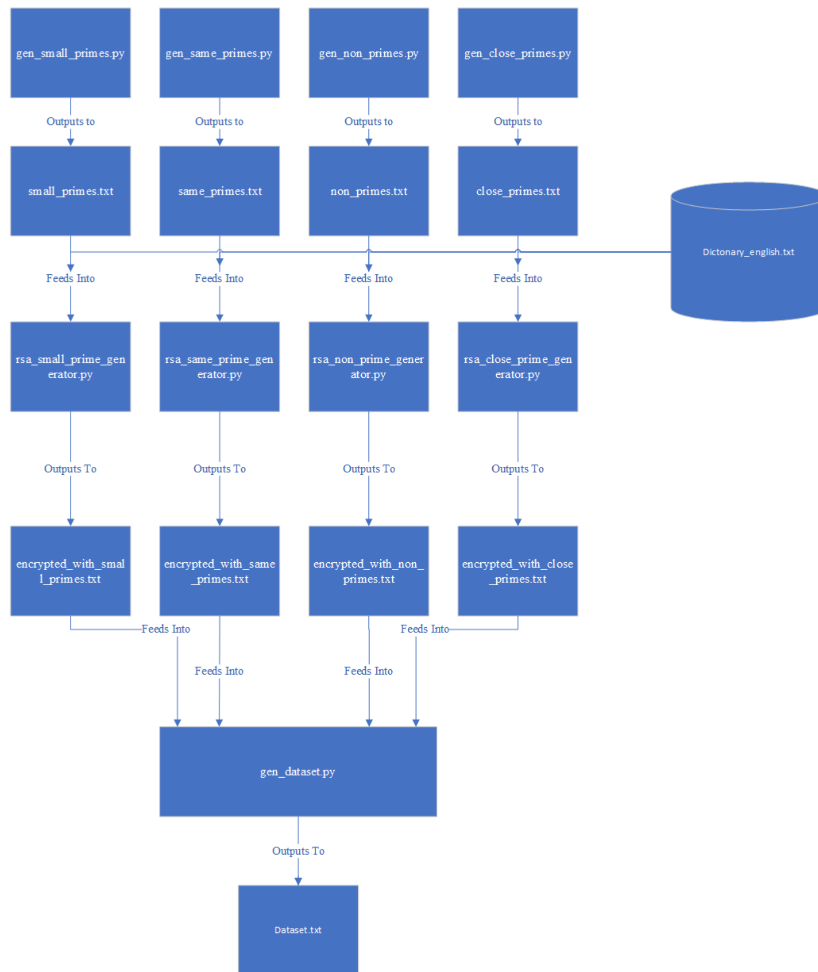


**Figure 5.1.2: Dataset generation process**

As mentioned above the output of these four scripts were sent to their relevant files. The reason for this approach is that in case some error occurred in the creation of the close primes for example only *gen_close_primes.py* would have to be fixed and run again rather than all 4. An example of one of the code these scripts can be seen in Figure 5.1.3.

A similar approach was taken for the generation of the test data of each weakness type. One script for each weakness type. These are:

- *rsa_small_prime_generator.py*
- *rsa_same_prime_generator.py*
- *rsa_non_prime_generator.py*
- *rsa_close_prime_generator.py*

The output of these four scripts was again sent to their own files following the separation of concerns approach being taken thus far. Figure 5.4 shows the main function of one these scripts as the whole program is long to be contain in a single screenshot.

```python
gen_small_primes.py > main
1   import sympy
2   import random
3   import datetime
4   import time as t
5
6   def main():
7       n_primes = 0
8       primes = []
9       not_prime = []
10
11      while n_primes < 10000000:
12          candidate = random.randint(2, 1000000000000000)
13
14          if sympy.isprime(candidate) and candidate not in primes and candidate not in not_prime:
15              primes.append(candidate)
16              n_primes += 1
17          else:
18              not_prime.append(candidate)
19
20
21      timestamp = datetime.datetime.now().strftime("%d-%m-%Y %H-%M-%S")
22      small_primes_file = 'Corpus/small_primes-' + timestamp + '.txt'
23      f = open(small_primes_file, 'a+')
24      primes.sort()
25
26      for prime in primes:
27          f.write(str(prime) + '\n')
28
29      f.close()
30
31  if __name__ == '__main__':
32      start_time = t.time()
33      main()
34      end_time = t.time() - start_time
35      print('[+] Time to execute: ' + str(end_time) + ' seconds')
```

**Figure 5.1.3 - gen_small_primes.py**

```python
1   import time as t
2   import math
3   import datetime
4   import secrets
5
6   def main():
7       dictonary_file = 'Corpus/Dictionary_english.txt'
8       f = open(dictonary_file, 'r')
9       words = f.read().splitlines()
10      small_words = [i for i in words if len(i) <= 8]
11      count = 0
12      sample_size = 2500
13      sampled_small_words = []
14      classifier = 1
15
16
17      for i in range(sample_size):
18          sampled_small_words.append(secrets.choice(small_words))
19
20      timestamp = datetime.datetime.now().strftime("%d-%m-%Y %H-%M-%S")
21      small_primes_file = 'Corpus/encrypted_with_small_primes-' + timestamp + '.txt'
22      f = open(small_primes_file, 'a+')
23
24      for word in sampled_small_words:
25          (public_key, private_key) = small_prime_rsa_generator()
26          binary_message = ''.join(format(ord(i), '08b') for i in word)
27          message = int(binary_message, 2)
28          ciphertext = rsa_encrypt(message, public_key[0], public_key[1])
29          f.write('{cipher} {e} {n} {classifier} \n'.format(cipher = ciphertext, e = public_key[0], n = public_key[1], classifier = classifier))
30          count += 1
31
32          if count % 1 == 0:
33              print('[+] Number of words encrypted with small primes: {count}'.format(count=count) )
34
35      f.close()
```

**Figure 5.1.4 - rsa_small_prime_generator.py (main function)**

As can be seen in Figure 5.1.4, line 7 refers to a text file called *Dictionary_english.txt*. This file contains ~466K words in the English language and words from this file were used as plaintext in the dataset that was used to train the machine learning models. It is also important to note that not all the words in this file are under consideration to be used as plaintext for the training/test data. The longest a word can be to be is 8 characters (this was restricted to 7 characters later though as will be discussed later).

The reason for this was that the words converted to binary and then to decimal. Before being encrypted the word must be turned into some decimal representation. The approach that was taken was to convert each character into its ASCII representation and then concatenate these binary strings into one string. For instance, consider the word ‘hi’. In ASCII ‘h’ is $(104)_{10}$ or $(01101000)_2$ and ‘i’ is $(105)_{10}$ or $(01101001)_2$. If these are concatenated, they form the binary string $(0110100001101001)_2$ or $(26729)_{10}$. This integer can be used as plaintext for the RSA algorithm has the function for this in Python only take integer values.

The benefit of this process is that is also reversible. Again consider $(26729)_{10}$ this can then be converted into its binary representation $(0110100001101001)_2$. This binary string can then be split into chunks of 8 which gives the binary strings $(01101000)_2$ and $(01101001)_2$. Their decimal representation is $(104)_{10}$ and $(105)_{10}$ respectively. These integers then correspond to the ASCII values for ‘h’ and ‘i’ which when concatenated from the word “hi”.

The script to run to generate the training/test data is run the script *gen_dataset.py*. This script takes the data from the following files:

- *encrypted_with_small_primes.txt*
- *encrypted_with_same_primes.txt*
- *encrypted_with_non_primes.txt*
- *encrypted_with_close_primes.txt*

The *gen_dataset.py* selects entries from each of these files using *secrets.SystemRandom().choice()*. This function uses the best resources available to the OS for its pseudorandom number generation and is cryptographically secure. Using this function also means that all weaknesses are almost equally represented in the outputted dataset. A script was created, *dataset_distribution.py*, to check this (Figure 5.1.5). From here the creation of the machine learning models could begin.

```
dataset_distribution.py > main
1   import time as t
2
3   def main():
4       dataset_set_file = 'Corpus/dataset-29-05-2025 22-30-12.txt'
5       f = open(dataset_set_file, 'r')
6       data_set = f.read().splitlines()
7       distro = [0,0,0,0]
8       target = 10000
9
10      for x in range(0,target):
11          data_set[x] = data_set[x].split(" ")
12
13
14      print(data_set[21][3])
15
16      for i in range(0, target):
17          if int(data_set[i][3]) == 1:
18              distro[0] += 1
19          elif int(data_set[i][3]) == 2:
20              distro[1] += 1
21          elif int(data_set[i][3]) == 3:
22              distro[2] += 1
23          elif int(data_set[i][3]) == 4:
24              distro[3] += 1
25          else:
26              continue
27
28      print('[+] Numerical distribution: {distribution}'.format(distribution = distro))
29      distro_percent = [0,0,0,0]
30
31      for i in range(0, len(distro)):
32          distro_percent[i] = (distro[i]/target)*100
33
34      print('[+] Percent distribution: {distribution}'.format(distribution = distro_percent))
35
36  if __name__ == '__main__':
37      start_time = t.time()
38      main()
39      end_time = t.time() - start_time
40      print('[+] Time to execute: ' + str(end_time) + ' seconds')
```

**Figure 5.1.5: dataset_distrubution.py**

## 5.2 Machine Learning Model Creation

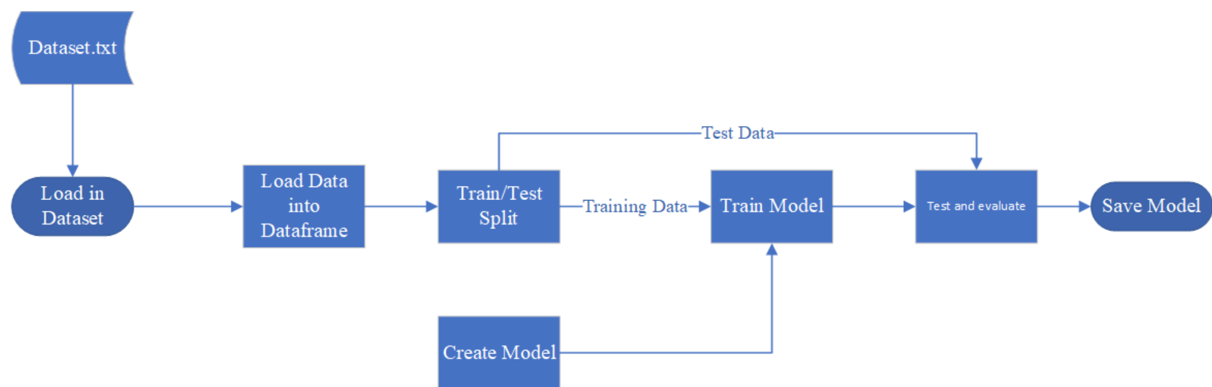


**Figure 5.2.1: Model creation and evaluation workflow**

This subsection describes how the classification models used in RSAnalyzer were created. Since all these models were created in the same way with very similar code and evaluated using the same metrics, this process only needs to be detailed once. All the code here comes from the process that created the Decision Tree Model which is one of the models in the final implementation of RSAnalyzer.

Figure 5.2.1 depicts the workflow followed in the creation, training, testing and evaluation of the machine learning models used by RSAnalyzer. Four machine learning models/techniques were chosen for this project, Decision Tree, Random Forest, Support Vector Machine and a Neural Network.

Implementations for all of these are found in *scikit-learn* library for Python. This library also provides functions for other areas such as splitting datasets into training and test data, training and testing models as well as several evaluation metrics are mentioned in Section 3.2 and will be discussed again in Section 6.

The next library worth mentioning is the *pandas* library which provides the dataframe data structure which stores the data used here. Figure 5.2.2 depicts what the *dataset.txt* looks like after being transformed into a dataframe.

[2]:

| | CIPHERTEXT | E | N | WEAKNESS |
|---|---|---|---|---|
| 0 | 604'748'946'572'838'543 | 451'048'770'367'503'731 | 998'093'883'187'041'169 | 2 |
| 1 | 28'119'964'586'568'152 | 6'643'446'499'187'525 | 29'827'408'340'482'553 | 1 |
| 2 | 70'663'314'081'689'366 | 90'619'567'292'633'869 | 99'792'983'745'075'589 | 4 |
| 3 | 1'272'674'794'633'425'472 | 423'981'693'765'885'523 | 2'370'286'718'938'749'601 | 2 |
| 4 | 1'194'677'993'264'936'249 | 1'413'928'284'072'732'007 | 3'992'456'268'101'033'329 | 2 |
| 5 | 338'282'407'439'008 | 22'838'232'061'649 | 802'803'849'365'672 | 3 |
| 6 | 297'544'715'640'317'235 | 190'122'917'932'945'993 | 585'628'245'507'361'477 | 1 |
| 7 | 80'096'289'798'456 | 7'645'541'723'323 | 217'030'464'722'610 | 3 |
| 8 | 2'789'661'280'142'970'900 | 1'345'824'209'015'861'069 | 3'183'838'855'499'631'169 | 2 |

Figure 5.2.2: Dataset Dataframe

As can be seen in Figure 5.2.2. The dataset contains 4 features:

1. **Ciphertext:** The transformed ciphertext of the plaintext
2. **E:** The exponent component of the public key
3. **N:** The modulus component of the public key
4. **Weakness:** The weakness the modulus has.
    a. The small prime for p and q weakness is encoded as 1
    b. The same prime for p and q weakness is encoded as 2
    c. The non-prime for p and q weakness is encoded as 3
    d. The close prime for p and q weakness is encoded as 4

The reason for these encodings is that models used here only understand numerical data and so categorical data must be given numerical representation. Since the data used here was generated synthetically (see Section 5.1), it saved time and computational resources to just have these encodings in the dataset from the start rather than having them in categorical data from and then having to encode them later with an approach like one-hot encoding.

Next as per Figure 5.2.1 is the Train/Test split phase. In this phase the data is divided into training features and target features. For the purposes of this work the Ciphertext, Exponent (E) and the Modulus (N) become the training features, and the Weakness becomes the sole target feature.

The built-in function *train_test_split* of *sklearn.model_selection* (*sklearn* is the same as *sci-kit learn*) then splits the dataset into train and test data with a 70%-30% split between training and test data. Figure 5.2.3 depicts the code that performs this process.

```
•[34]: features = ['ciphertext', 'e', 'n']
       X = df[features]
       y = df.weakness

       X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
```

**Figure 5.2.3: Train/Test split**

Next is the training phase and where the model itself is created along with setting the parameters of the model. The time taken for the training phase here as well. Figure 5.2.4 shows the code used for this.

```
DT_classifier = DecisionTreeClassifier(criterion='log_loss')
start_time = t.time()
DT_classifier = DT_classifier.fit(X_train.values, y_train.values)
end_time = t.time()
training_time = end_time - start_time
print('[+] Training Compete!')
print('[+] Time Elapsed: {time}'.format(time=training_time))

[+] Training Compete!
[+] Time Elapsed: 0.024672508239746094
```

**Figure 5.2.4: Training phase**

```
[+] Time Elapsed: 0.024672508239746094

[5]: y_pred = DT_classifier.predict(X_test.values)
     print('[+] Predictions Complete!')

[+] Predictions Complete!
```

**Figure 5.2.5: Testing phase**

Next the testing phase is performed. This where the newly trained model attempts to predict the weakness of the 30% of the dataset that it has not seen. The code for this can be seen in Figure 5.2.5. These predictions will then be used in the calculations of the performance metrics in the evaluation phase.

```
print("--------------ACCURACY--------------------")
model_accuracy = metrics.accuracy_score(y_test, y_pred)
print("[+] Accuracy: {accuracy}".format(accuracy=model_accuracy))
print("")
print("--------------PRECISION--------------------")
print("[+] Precision (macro): {precision}".format(precision=model_precision_macro))
model_precision_micro = metrics.precision_score(y_test, y_pred, average='micro')
print("[+] Precision (micro): {precision}".format(precision=model_precision_micro))
model_precision_weighted = metrics.precision_score(y_test, y_pred, average='weighted')
print("[+] Precision (weighted): {precision}".format(precision=model_precision_weighted))
print("")
print("--------------RECALL-----------------------")
model_recall_macro = metrics.recall_score(y_test, y_pred, average="macro")
print("[+] Recall (macro): {recall}".format(recall = model_recall_macro))
model_recall_micro = metrics.recall_score(y_test, y_pred, average="micro")
print("[+] Recall (micro): {recall}".format(recall = model_recall_micro))
model_recall_weighted = metrics.recall_score(y_test, y_pred, average="weighted")
print("[+] Recall (weighted): {recall}".format(recall = model_recall_weighted))
print("")
print("--------------F1 Score-----------------------")
model_f1_score_macro = metrics.f1_score(y_test, y_pred, average="macro")
print("[+] F1 Score (macro): {f1}".format(f1 = model_f1_score_macro))
model_f1_score_micro = metrics.f1_score(y_test, y_pred, average="micro")
print("[+] F1 Score (micro): {f1}".format(f1 = model_f1_score_micro))
model_f1_score_weighted = metrics.f1_score(y_test, y_pred, average="weighted")
print("[+] F1 Score (weighted): {f1}".format(f1 = model_f1_score_weighted))
print("")
print("--------------Confusion Matrix-----------------------")
import datetime
model_confusion_matrix = metrics.confusion_matrix(y_test, y_pred, labels=[1,2,3,4])
cm_labels = ["small", "same", "non", "close"]
cm_disp =  metrics.ConfusionMatrixDisplay(confusion_matrix=model_confusion_matrix, display_labels=cm_labels)
cm_disp.plot()

time_stamp = datetime.datetime.now().strftime("%d-%m-%Y %H-%M-%S")
cm_disp.figure_.savefig("cm_log_loss" + time_stamp + ".png")
```

**Figure 5.2.6: Evaluation Phase – Code**

Figure 5.2.6 depicts the code used for the evaluation phase. It contains code for all metrics mentioned in Section 3.2.1, accuracy, precision, recall and f1 score. The results outputted during this phase will be discussed in Section 6.

Finally, if the evaluation metrics are satisfactory, the model can be saved off using the *pickle* library to be used in RSAnalyzer. The code for this can be seen in Figure 5.2.7.

```
import pickle
import datetime

timestamp = datetime.datetime.now().strftime("%d-%m-%Y %H-%M-%S")
model_file = 'decision_tree_log_loss-' + timestamp + '.pkl'
# Models_folder = 'D:\College\Thesis\Code\models\decision_trees'
Models_folder = 'models\decision_trees'
full_path = '{folder}\{file}'.format(folder = Models_folder, file = model_file)

file = open(full_path, 'wb')
pickle.dump(DT_classifier, file)
```

**Figure 5.2.7: Save model phase**

## 5.3 RSAnalyzer

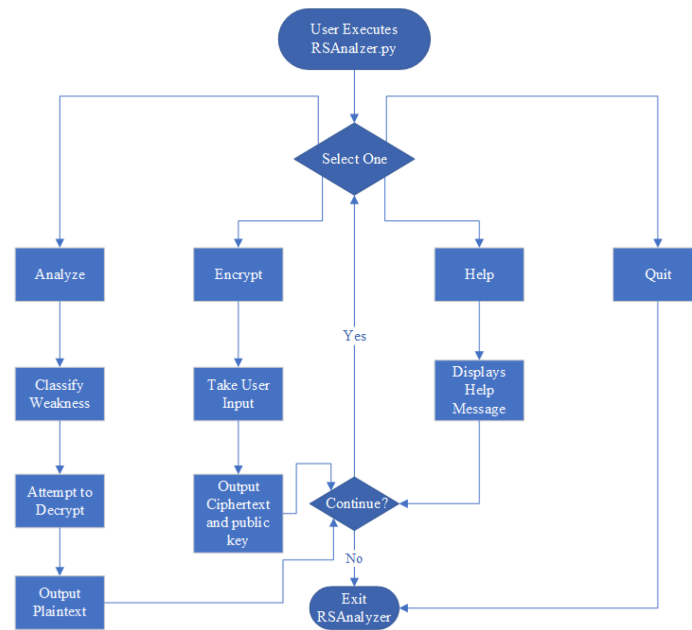


**Figure 5.3.1: RSAnalyzer Flowchart**

RSAnalyzer once started has 4 options presented to the user, Analyze, Encrypt, Help, Quit. An overview of how RSAnalyzer operated when each of these options are chosen is depicted in Figure 5.3.1. The Help and Quit options are straightforward, Help just gives a brief description of what RSAnalyzer does and Quit exits the program. The Analyze and Encrypt options merit further discussion, however.

### Analyze

Choosing "Analyze" brings the user to the main purpose of RSAnalyzer. This mode prompts the user to input a ciphertext, exponent and modulus to be cryptanalyzed (See Figure 5.3.2)

```
PS D:\College\Thesis\Code> & C:/ProgramData/anaconda3/python.exe d:/College/Thesis/Code/RSAnalyzer/RSAnalyzer.py
------------Welcome to RSAnalyzer---------------
[+] What would you like to do?
[?] Analyze (A)
[?] Encrypt (E)
[?] Quit (Q)
[?] Help (H)
[> A
[+] Please provide input in the following order Ciphertext e n:
[> 1642155502727012610 20272330115753995531 2397740012936410081
[+] Select a machine learning model to analyse with
[?] Decision Tree (DT)
[?] Random Forest (RF)
[?] Support Vector Machine (SVM)
[?] Neural Network (NN)
[> dt
[+] Loading model
[+] Model Loaded!
[+] 1642155502727012610 20272330115753995531 2397740012936410081
[+] Performing cryptanalysis now.....
[+] N was constructed using the same prime for p and q.
[+] Factorising n....
[+] The value of p and q is: 1548463759
[+] Calculating Euler Totient Function of n....
[+] Euler Totient Function of n is: 2397740011387946322
[+] Calculating d...
[+] 20272330115753995531^-1 mod 2397740011387946322 = 496980620865997997
[+] Decrypting Ciphertext...
[+] Decrypted message is (decimal format): 1718579060
[+] Converting to text.....
[+] The Original Message is: foot
[?] Would you like to continue using RSAnalyzer? (Y/N)
[> 
```

**Figure 5.3.2: RSAnalyzer - Analyze**

The user is then prompted to select between one of the four classifiers trained in Section 5.2. Once the desired model has been selected, RSAnalyzer classifies the weakness and then attempts one of 4 methods to retrieve the original plaintext (See Section 4).

RSAnalyzer then gives a breakdown how it retrieved the plaintext base on the weakness classified. It shows the values for p and q, the Euler Totient function of the modulus N and the exponent component of the private key. This was implemented not only for testing purposes, but it also makes RSAnalyzer seem like less of a black box. The user is then given the option of going back to the main menu or exiting the program.

### Encrypt

The Encrypt function was implemented because it became clear during initial testing of RSAnalyzer that getting the user to use any online RSA calculator to create their inputs produce 2 main problems.

First is that many of these online calculators only let users input relatively small numbers as input. The reason some give for this is to stop a single user's input from crashing the site.

The second issue is that a lot of these online RSA calculators only take numerical input. So even if any of these calculators let the user input whatever integers they wanted, the question remains of what if the user wants to use text input instead. The user would need another external site to encode and decode the plaintext and ciphertext.

To this end the Encrypt option was created. This was relatively easy to implement as most of the code for encoding and encrypting text had already been developed for the data generation stage (See Section 5.1). The only new code was for adding the option to the menu and taking user input.

```
PS D:\College\Thesis\Code> & C:/ProgramData/anaconda3/python.exe d:/College/Thesis/Code/RSAnalyzer/RSAnalyzer.py
------------Welcome to RSAnalyzer---------------
[+] What would you like to do?
[?] Analyze (A)
[?] Encrypt (E)
[?] Quit (Q)
[?] Help (H)
[> e
Please provide a string to encrypt. String must be 7 characters or less.
[> candle
[+] Encrypting with  non-primes for p and q.......
[114216048456381, 110530836769163, 995076155496384]
[+] What would you like to do?
[?] Analyze (A)
[?] Encrypt (E)
[?] Quit (Q)
[?] Help (H)
[>
```

**Figure 5.3.3: Encrypt mode**

This mode also tells the user what weakness the ciphertext has, in case the wrong weakness is classified, and the wrong decryption technique is applied. But this might not always be a problem as can be seen in Section 6.

# 6 Evaluation and Discussions

RSAnalyzer's ability to decrypt ciphertext is almost entirely dependent on whether it can classify the input with the correct weakness or not. To this end, this section will focus on

evaluating the classifiers used in RSAnalyzer's classification step (Algorithm 4.1 – Step 2). Each classifier will be evaluated on the metrics outlined in Section 3.2.

## 6.1 Experimental Setup

For all the experiments in this section the following weaknesses were be included in the dataset:

- N was constructed using small primes for p and q.
- N was constructed using the same prime for p and q.
- N was constructed using non-primes for p and q.
- N was constructed using close primes for p and q.

The data in these experiments only used words which had 7 characters or less and had been encrypted with one of the 4 weaknesses listed above. All these values were selected using Python3's own cryptographically secure pseudorandom number generation library *secrets* and the *secrets.SystemRandom()* function to ensure relatively even distribution between all classes (or weaknesses) in the dataset. Table 6.1.1 gives a sample from the dataset and Table 6.1.2 details the distribution of the data in said dataset.

**Table 6.1.1: Sample from Dataset used for Decision Tree Classifier**

| Ciphertext | E | N | Weakness |
|---:|---:|---:|---:|
| 604'748'946'572'838'543 | 451'048'770'367'503'731 | 998'093'883'187'041'169 | 2 |
| 28'119'964'586'568'152 | 6'643'446'499'187'525 | 29'827'408'340'482'553 | 1 |
| 338'282'407'439'008 | 22'838'232'061'649 | 802'803'849'365'672 | 3 |
| 70'663'314'081'689'366 | 90'619'567'292'633'869 | 99'792'983'745'075'589 | 4 |

Weakness Encodings:

- Small Primes = 1
- Same Primes = 2
- Non-Primes = 3
- Close Primes = 4

**Table 6.1.2: Data Distribution for Dataset used in 6.2-6.4**

| Weakness | Number in Dataset | Percentage of Dataset |
|---|---|---|
| Small Primes | 999 | 24.975% |
| Same Primes | 1024 | 25.6% |
| Non-Primes | 992 | 24.8% |
| Close Primes | 985 | 24.625% |

## 6.2 Decision Tree Classifier

The decision tree classifier was evaluated based on the split function used by the classifier. These split functions are Gini Impurity (Table 6.2.1), Entropy (Table 6.2.2) and Log Loss (Table 6.2.3). Each of these Split Functions were evaluated based on Accuracy, Precision, Recall and F1 score.

**Table 6.2.1: Decision Tree – Evaluation Metrics**

| Metric | Gini Impurity | Entropy | Log Loss |
|---|---|---|---|
| **Accuracy** | 0.8275 | 0.8333 | 0.8342 |
| **Precision** | 0.8275 | 0.8333 | 0.8342 |
| **Recall** | 0.8275 | 0.8333 | 0.8342 |
| **F1 Score** | 0.8275 | 0.8333 | 0.8342 |

As can be seen in Tables 6.2.1, the split functions all yielded similar scores on their evaluation metrics. This means that using any one of these over the others would not make a significant difference in the classifications made by the Decision Tree Classifier. However, there is a clear best option when it comes to which one to choose to use in RSAnalyzer and that is the Log Loss function. It has the best scores across all the evaluation metrics. Figure 6.2.2 depicts the Confusion Matrix for the Log Loss function.

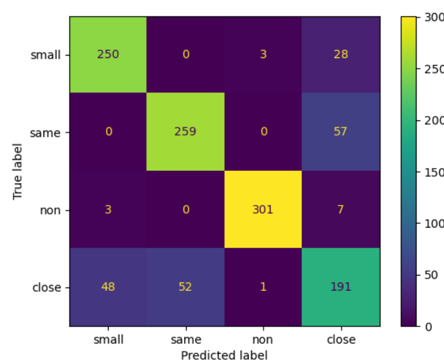


**Figure 6.3.2: Confusion Matrix for Decision Tree using Log Loss**

## 6.3 Random Forest Classifier

Like the decision tree classifier, the Random Forest Classifier was evaluated based on the split function used by the classifier. These split functions are Gini Impurity (Table 6.3.1), Entropy and Log Loss (Table 6.3.2). Each of these Split Functions were evaluated based on Accuracy, Precision, Recall and F1 score. The Entropy and Log Loss functions were combined into one table as the yielded the same results.

**Table 6.3.1: Random Forest – Evaluation Metrics**

| Metric | Gini Impurity | Entropy | Log Loss |
|---|---|---|---|
| **Accuracy** | 0.8533 | 0.8550 | 0.8550 |

| Precision | 0.8533 | 0.8550 | 0.8550 |
|---|---|---|---|
| **Recall** | 0.8533 | 0.8550 | 0.8550 |
| **F1 Score** | 0.8533 | 0.8550 | 0.8550 |

Like the previous classifier the choice of Split Function would make little difference between the quality of the classification made by the Random Forest classifier. Since there was a tie between Entropy and Log Loss, the Log Loss function was chosen only because it showed to perform better in the previous classifier as well. Figure 6.3.2 depicts the confusion matrix of the Random Forest classifier when using Log Loss (or Entropy).

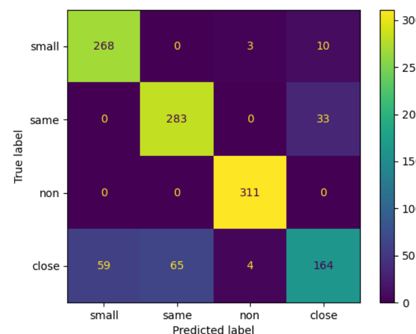


**Figure 6.3.2: Confusion Matrix Random Forest using Entropy or Log Loss**

## 6.4 Support Vector Machine Classifier

Support Vector Machines (SVMs) do not have split functions, but they do have kernel functions. The ones that are considered here are the Radial Basis Function (RBF) kernel (Table 6.4.1) and the Polynomial kernel (Table 6.4.2) and Figure 6.4.3 shows the confusion matrix of an SVM using an RBF kernel.

**Table 6.4.1: Support Vector Machine – Evaluation Metrics**

| Metric | RBF | Polynomial |
|---|---|---|
| **Accuracy** | 0.7025 | 0.6708 |
| **Precision** | 0.7025 | 0.6708 |
| **Recall** | 0.7025 | 0.6708 |
| **F1 Score** | 0.7025 | 0.6708 |

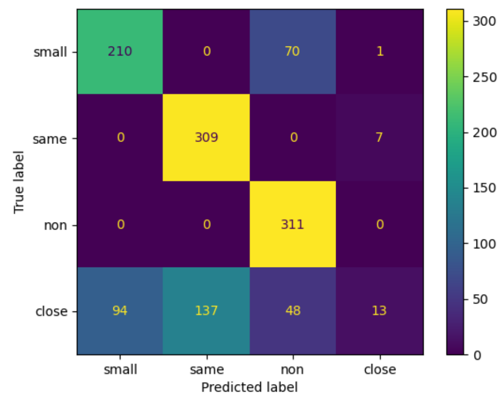


**Figure 6.4.2: Confusion Matrix for SVM using RBF Kernel**

The Radial Basis Function was the best option between the two kernels. This was the kernel the Support Vector Machine in RSAnalyzer.

## 6.5 Neural Network Classifier

The Neural Network Classifier uses optimizer functions referred to as “solvers” in Python to minimise the loss function. The two solvers considered here are Adam and Stochastic Gradient Descent. Table 6.5.1 shows the results of using each of these solvers.

**Table 6.5.1: Neural Networks - Evaluation Metrics**

| Metric | Adam | SGD |
|---|---|---|
| **Accuracy** | 0.7325 | 0.7340 |
| **Precision** | 0.7325 | 0.7340 |
| **Recall** | 0.7325 | 0.7340 |
| **F1 Score** | 0.7325 | 0.7340 |

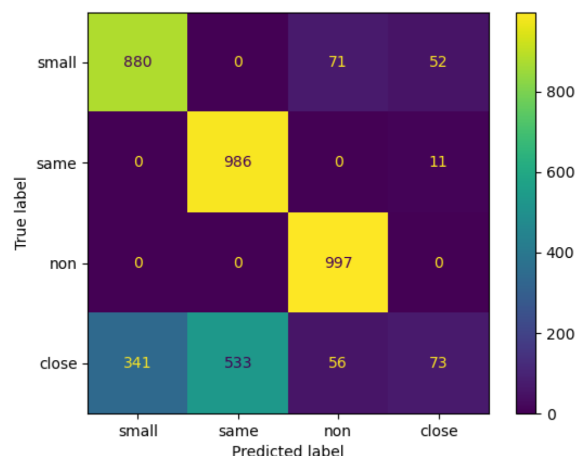


**Figure 6.5.2: Confusion Matrix for Neural Network using SGD Solver**

Here the Stochastic Gradient Descent has a slight edge on the Adam solver in terms of the evaluation metrics used. SGD was the solver chosen for the neural network for RSAnalyzer.

### 6.6 Misclassification of Weakness

As can be seen throughout this section none of the solutions implemented for classifying weaknesses is 100% percentage effective in its classification. So, what happens to the ciphertexts that have the "wrong" decryption method applied to them?

The answer is a at least some of them will be decrypted anyway. The reason for this did not become evident until producing the confusion matrices for this section. Not all the weakness chosen for this work are mutually exclusive. It is possible for a ciphertext to have more than one weakness. The weakness that seems to share the most overlap with the others is the close weakness. It is possible for p and q to be close primes and small primes, close and non-prime close primes and the same prime. That last one should be intuitive as what could be closer than two numbers being equal?

The only mutually exclusive weakness on paper would be prime and non-prime values for p and q but according to Figure 6.5.2, 71 of the values for p and q the neural network classed as non-primes were small prime numbers.

This means that even if a classifier gets the class wrong it is possible for that ciphertext to still be decrypted. This should hold if the correct formula for the Euler Totient function is used.

If for example a given input has the "small primes" weakness but gets misclassified with the "non-primes" weakness. In this scenario, the ciphertext should still be decrypted as the formula used to calculate the Euler Totient Function is generic (Eq. 4.4.1). However, the reverse would not be true as a "non-prime" input classified as "small primes" would be calculating the Euler Totient function based on a special case which assumes both factors of n are distinct prime numbers (Eq. 4.2.1)

# 7 Conclusion

In the Introduction to this work (Section 1), the question "*Can supervised machine learning models detect known weaknesses or RSA?"* was asked and the aim of this work was to answer that question.

During this research, it has been shown that supervised machine learning models can attain an accuracy of 70.25% - 85.50% depending on the machine learning method used. With the best machine learning technique based on the evaluation metrics used being Random Forest, with the worst performing method was the Support Vector Machine.

It was also shown that if a given ciphertext is misclassified with the "wrong" weakness then that does not mean that the original plaintext will not be retrieved.

For future work, the next obvious step would be to get RSAnalyzer working on text of larger input size, move away from requiring the user to use input that can be represented in ASCII. Adding more weakness to be detected by the classifier such as what p is a prime number and q is not and vice versa.

It would be interesting to see if any regression models could be incorporated into RSAnalyzer for the decryption steps so that in future RSAnalyzer might be entirely machine learning driven.

# References


Ajeet, S., Sivangi, B. & Tentu, N. A., 2024. Machine Learning and Cryptanalysis: An In-Depth Exploration of Current Practices and Future Potential. *Journal of Computing Theories and Applications,* 1(3), pp. 257-272.

dwyl, 2018. *dwyl/english-words.* [Online]
Available at: https://github.com/dwyl/english-words
[Accessed 10 August 2025].

Hardy, G. W. E., 2008. *An Introduction to the Theory of Numbers.* 6th Edition ed. Oxford: Oxford University Press.

Müller, A. C. G. S., 2016. 5. Model Evaluation and Improvement . In: D. Schanafelt, ed. *Introduction to Machine Learning with Python* . Sebastopol: O'Reilly Media Inc., pp. 253-304.

Stallings, W., 2017. *Cryptography and Network Security - Principles and Practice.* 7th ed. Harlow: Pearson Education Limited.

Tolba, Z., Derdour, M. & Dehimi, N. E. H., 2022. *Machine learning based cryptanalysis techniques: perspectives, challenges and future directions.* El Bouaghi, Institute of Electrical and Electronics Engineers (IEEE).

Yaseen, I. F. & Sahasrabuddhe, H., 1999. *A Genetic Algorithm for the cryptanalysis of Chor-Rivest.* Delhi, Computational Intelligence and Multimedia Applications, International Conference on, New Delhi, India, 1999.

Ye, Q. et al., 2022. Efficient Lattice-Based Ring Signature Scheme without Trapdoors for Machine Learning.. *Computational Intelligence & Neuroscience,* Volume 2022, pp. 1-13.