2020

ML Crypto Analyzer – Using ML to Leak Information from Ciphertext

A STUDY PROJECT BY
ROY DAR

Contents

Introduction

One definition of a learning algorithm is its ability to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E (Tom Mitchell/1998).

In the last couple of years, the ML field has improved significantly allowing sophisticated tasks to be performed using learning algorithms.

A significant part of the improvement in this field is attributed to the increase in computational power allowing models which were only theoretical in the past to become practical (e.g. neural networks).

In this study, I attempted to find out if a learning algorithm can leak information about a cipher text.

In this trial I tried both classic/simple encryption methods (for example "Caesar" like cipher or simple XOR cipher) and more modern ciphers (e.g. AES, DES, 3-DES).

The trials I used had the following goals:

1. Detecting the encryption schema that was used
2. Detecting the type of the original plaintext (e.g. binary file, English text)

Project description

The project implementation was done in **python 3.x**.

The following libraries were used:

1. **numpy** – mathematic library for python
2. **pycryptodome** – An implementation of modern ciphers for python
3. **tensorflow** – A base core component for ML implementation
4. **keras** – A more advanced API for ML on top of tensorflow to easily create, train and test more advanced and powerful models.
5. **termplot** – A small library allowing plotting charts on terminal

The project has the following main components:

1. **Plain text generators** (generators/plaintext_generator.py)
    a. A generator allowing generating amounts of plaintext for the training stage (and for the testing as well) of dynamic sizes.

    The package includes the following generators:
        i. Binary plaintext generator (class BinaryPlaintextGenerator)
            1. The binary plaintext generator will generate a random binary stream of information.
        ii. Dictionary based generator (class DictionaryPlaintextGenerator)
            1. This class will generate a random plaintext based on an input dictionary of words. At this stage the generation is completely random, but it can be improved in the future to allow more sophisticated generation algorithms such as ML to provide a text with a proper entropy of words according to the language.
            2. I used a 400K words dictionary (that is usually used for ML projects) in this project. See the "Running this project" section for more details

2. **Encryption manager** (encryption/ encryption_manager.py)
    a. The encryption manager supports encryption and decryption of data.
    b. Accepts a key-size and block-size as parameters (padding is done automatically)
    c. Currently the following encryption ciphers are supported by the encryption manager:
        i. Shift cipher (Based on the classical 'Caesar' cipher only for 8-bit charecters). Uses keys like the Vigenère cipher.

1. Supports only ECB mode

ii. XOR cipher
1. Supports only ECB mode

iii. AES
1. Supported modes: ECB, CBC, CFB, OFB, CTR

iv. 3-DES (called DES3 in this project)
1. Supported modes: ECB, CBC, CFB, OFB

v. DES
1. Supported modes: ECB, CBC, CFB, OFB

d. Depending on the selected mode, an IV or NONCE may be returned by the encryption operation. If those are returned, they must be used for the decryption process as well.

3. **Datastore** (datastore/datastore.py)
   a. The datastore allows storing and retrieval of information.
   b. Each record must be in a dictionary format to be stored (and it will be returned on the same format)
   c. The file based datastores will automatically convert data into a binary stream (type of 'bytes'). Only 'bytes', 'bytearray' or 'str' formats are supported
   d. Originally, the file based datastore were implemented to allow generating the training data for the ML and then training based on that data. However, using that approach did not have any performance improvement comparing to on-line generation of the training data. When generating the data on-line, the amount of data used for the training is infinite and therefore there is no risk of overfitting to the data (i.e. high variance)

The package includes the following datastore implementation:

i. In Memory Datastore (class InMemDatastore)
1. Stores all the information in the memory (on an in-mem queue)

ii. File Datastore (class FileDatastore)
1. Stores the data on a disk file
2. Use iteration for reading and writing, so the data is never on the machine memory
3. Is designed for huge datasets which does not fit the main machine memory
4. Was originally designed to be used as an input for the training process without any memory limitation, however since Keras requires knowing the number of mini-batches in advance, this implementation is not compatible, and therefore the mult-file implementation can be used instead

iii. Split Files Datastore (class SplitFilesDatastore)
1. Implementation that supports only writing data

2. Will write multiple files on the disk, depending on the configured parameter of number of records per file
3. Uses the File datastore for the internal write operation, so each file can be read later with the File Datastore class.

4. **ML model creator** (ml/keras_model_creator.py)
    a. This package is responsible for generating the ML models for this project
    b. All models end with a final FC layer using softmax activation for the classification prediction
    c. All models support dropout to avoid overfitting
    d. All models support a configurable number of layers
    e. The input supports the following options
        i. Normal float values (of the binary encoding of the inputs). Will require a Batch Normalization on the first layer for maximum training performance. Only supported by the FC NN network.
        ii. Embedding of the binary value (into a one-hot 0-255 classification of the binary value). Required for the CNN and RNN networks.
    f. The following models are supports
        i. NN - Fully Connected Neural Network (class KerasFullyConnectedNNModelCreator)
            1. Will create a FC NN
            2. Uses the same number of hidden units in each hidden layer (configurable)
        ii. CNN - Convolutional Neural Network (class KerasCNNModelCreator)
            1. Creates a CNN network which uses Conv1D units.
            2. Accepts as configuration the number of hidden layers
            3. Accepts as configuration the # of filters on the first layer, kernel size on first layer and the stride on the first layer
            4. In each layer after that the # of filters doubles and the kernel size and strides halves
            5. Requires embedding for the input
            6. CNN did not yield any better performance than FC NN in the context of this project.
        iii. RNN - Recurrent Neural Network (class KerasRNNModelCreator)
            1. Creates an RNN network
            2. Ideal for variable size data
            3. The number of parameters of the network does not grow as the input size grows
            4. Uses LSTM (long-short term memory) units
            5. For best results uses Conv1D before the first RNN layer
            6. Requires embedding for the input

7. This is the only network that supports not specifying the input size (since the model supports variable input sizes)

5. **Other components**:
    a. Training Data Generator (ml/ training_data_generator.py) –
        i. Generates the training data needed for the ML
        ii. Each training example is generated randomly from the input plaintext generators and encrypted using random encryption manager
        iii. Outputs the data to a datasource implementation
    b. Keras Data Generator (ml/ keras_data_generator.py) –
        i. Implementation of the Keras 'Sequence' class for input of training data
        ii. Supports getting the input data from either multi-file directory (generated by the split file data source implementation) or generating on-line (using a training data generator as the engine)
        iii. Converts the data into the required format for ML training
            1. Input data is converted to a numpy matrix of size (num of training examples in mini-batch * max input size).
            2. Ouput data is converted to a numpy matrix of size (num of training example in mini-batch * num of classifications). The implementation will convert each training example Y value to a one-hot vector.
            3. The implementation also supports scaling the inputs which is required for training unless input normalization is used on the model itself.
    c. ML Utils (ml/ml_utils.py)
        i. A collection of utilities for ML which allows:
            1. Creating a training a model
            2. Evaluating a model
            3. Showing statistics of a model
    d. Unit tests (directory /tests)
        i. The project includes various unit tests
    e. ML trials main runners (director /ml/trials)
        i. This includes the main trials logic (see section below)
    f. Application main runner file (main.py)
        i. The file is used for running the project

The trials (tests)

All test sizes are currently tweaked to run under the machine requirements (see running section below). To run on other machines the sizes of input sizes, batch sizes and network sizes can be changed in code. On the tests below the aim is to minimize loss (needs to be as close to 0) and maximize the accuracy (as close to 1).

All the trials use on-line training data generation. Originally, I generated a file stored training set and used it, but I got into scenarios of over fitting to the training data. I then switched to on-line generation and found it to have no implication on the performance.

For all trials I used a mix of the types and sizes of NN and the types of input encoding.

Due to computational resource limitations, I was limited in the sizes of inputs and model complexity I could use.

The results you see before you are the ones that gave me the best performance.

The project includes the following tests. Each can be run separately.

The list of trials is laid out on the next pages.

Trial #1 – A sanity test

This test aim was to see if the NN is performing as expected under a very easy task. The ML in this task receives an un-encrypted plaintext, and its job is just to learn the classification between a dictionary (English) text or binary data.

Model: FC NN, 1 hidden layer
Training: 50 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few seconds
Result: Success

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 128)               256128
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 258
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 264,386
Trainable params: 260,386
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.00031924519687891005
categorical_accuracy - 1.0
Model has excellent performance! ;-)
```

Trial #2 – Detect plaintext type from a legacy SHIFT encrypted ciphertext

In this trial I used the SHIFT encryption schema with a long but hard-coded (single) key.
The job of the ML was that, given enough classified inputs, recognize the type of plaintext from the cipher text (without knowing the encryption key of course)

Model: FC NN, 1 hidden layer
Training: 50 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few seconds
Result: Success

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 128)               256128
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 258
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 264,386
Trainable params: 260,386
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 1.1563289262994659e-07
categorical_accuracy - 1.0
Model has excellent performance! ;-)
```

Trial #3 – Detect plaintext type from a legacy XOR encrypted ciphertext

In this trial I used the XOR encryption schema with a long but hard-coded (single) key.
The job of the ML was that, given enough classified inputs, recognize the type of plaintext from the cipher text (without knowing the encryption key of course)

Model: FC NN, 1 hidden layer
Training: 50 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few seconds
Result: Success

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 128)               256128
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 258
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 264,386
Trainable params: 260,386
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 1.5584715270996095
categorical_accuracy - 0.9700000286102295
Model has excellent performance! ;-)
```

Trial #4 - Detect plaintext type from real OTP (one-time pad) encrypted ciphertext

In this trial I used XOR but with random key for each training example. As proven and expected, the ML failed to recognize the plain text type

Model: FC NN, 1 hidden layer
Training: 50 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few seconds
Result: Success

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 128)               256128
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 258
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 264,386
Trainable params: 260,386
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 33.29538222551346
categorical_accuracy - 0.4399999976158142
Model failure to predict :-(
```

Trial #5 – Detect the type of cipher that was used between XOR or SHIFT

In this trial I used the same plaintext type (English text) but this time the encryption method changed (XOR, SHIFT). I used the same key for each one (as we saw above if the key randomly changes for each training example, ML will be unable to detect anything.
The ML job was given the ciphertext detect the type of encryption schema that was used.

Model: FC NN, 5 hidden layers – 100 units per layer
Training: 50 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few seconds
Result: Success

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 100)               200100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_5 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_6 (Dense)              (None, 5)                 505
_________________________________________________________________
activation_1 (Activation)    (None, 5)                 0
=================================================================
Total params: 249,005
Trainable params: 245,005
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.0
categorical_accuracy - 1.0
Model has excellent performance! ;-)
```

Trial #6 - Detect plaintext type from AES (single key)

Moving on to modern ciphers I now used AES with a single key to encrypt all the training examples. The job of the ML was that, given enough classified inputs, recognize the type of plaintext from the cipher text.
To my surprise the ML model was unable to detect the types even when I used a single key for all the training examples and even after 1000 rounds of training on freshly randomized data. I tried FC NN, CNN and RNN networks but those all failed.

Model: FC NN, 5 hidden layers – 100 units per layer
Training: 1000 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few minutes
Result: Failure

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 100)               200100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_5 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 202
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 248,702
Trainable params: 244,702
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.6987335538864136
categorical_accuracy - 0.4099999964237213
Model failure to predict :-(
```

Trial #7 – Same as #6 only with DES encryption

After the failure to detect anything from AES ciphertext I tried the same for DES. Surprising enough, also here the ML failed to detect anything as well.

Model: FC NN, 5 hidden layers – 100 units per layer
Training: 1000 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few minutes
Result: Failure

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 100)               200100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_5 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 202
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 248,702
Trainable params: 244,702
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.694967930316925
categorical_accuracy - 0.5
Model failure to predict :-(
```

Trial #8 – Detect the type of cipher that was used between AES or DES

This trial is similar to #5, only using AES and DES (both using a single key for all training examples)

Again, the plaintext is English text and the ML goal is to recognize which cipher was used AES or DES.

The result surprised me because the ML was able, to a degree higher than randomness, detect some differences between the cipher texts that were generated.

Model: FC NN, 5 hidden layers – 100 units per layer
Training: 1000 rounds, each round of 1000 training example
Plaintext size: Variable size up to 2000 bytes (with embedding as one-hot enabled)
Training time: a few minutes
Result: Partial Success (not perfect but >80% of accuracy)

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
embedding_1 (Embedding)      (None, 2000, 256)         65536
_________________________________________________________________
flatten_1 (Flatten)          (None, 512000)            0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               51200100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_5 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_6 (Dense)              (None, 5)                 505
_________________________________________________________________
activation_1 (Activation)    (None, 5)                 0
=================================================================
Total params: 51,306,541
Trainable params: 51,241,005
Non-trainable params: 65,536
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.25182858550548554
categorical_accuracy - 0.8889999985694885
Model has good performance :-)
```

Trial #9 – Detect the type of cipher that was used between AES or DES (changing keys)

Same as #8 only now the keys are random and changes between each training example.

As expected, the ML failed to detect the cipher that was used.

Model: FC NN, 5 hidden layers – 100 units per layer
Training: 1000 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few minutes
Result: Failure

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 100)               200100
_________________________________________________________________
dense_2 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_5 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_6 (Dense)              (None, 5)                 505
_________________________________________________________________
activation_1 (Activation)    (None, 5)                 0
=================================================================
Total params: 249,005
Trainable params: 245,005
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.6913213872909546
categorical_accuracy - 0.5
Model failure to predict :-(
```

Trial #10 – Same as #2 only with random changing keys of small size

Using SHIFT encryption schema with a short random key.
The job of the ML was that, given enough classified inputs, recognize the type of plaintext from the cipher text.

Model: FC NN, 5 hidden layers – 100 units per layer
Training: 100 rounds, each round of 100 training example
Plaintext size: Variable size up to 2000 bytes
Training time: a few minutes
Result: Success

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2000)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 2000)              8000
_________________________________________________________________
dense_1 (Dense)              (None, 128)               256128
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_4 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_5 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 258
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 330,434
Trainable params: 326,434
Non-trainable params: 4,000
```

Result:

```
Evaluating:...
*** Model result ***
loss - 0.016739515371446034
categorical_accuracy - 0.9900000095367432
Model has excellent performance! ;-)
```

Trial #11 – Same as #2 only with RNN (with variable sizes of inputs).

The AIM of this test, was to test the performance of RNN, in using variable sizes of inputs, from very small to very large (100 – 10,000)

As in trial #2, I was using a simple SHIFT cipher with fixed large key.

Model: RNN network (Conv1D -> LSTM -> LSTM -> FC -> ACTIVATION) with dropout ratio 0.05 between each layer
Training: 100 rounds, each round of 100 training example
Plaintext size: Variable size from 100 to 10,000 bytes
Training time: a few minutes
Result: Success on all sizes

Model:

```
Model Summary:
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None)              0
_________________________________________________________________
embedding_1 (Embedding)      (None, None, 256)         65536
_________________________________________________________________
conv1d_1 (Conv1D)            (None, None, 100)         3276900
_________________________________________________________________
dropout_1 (Dropout)          (None, None, 100)         0
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 100)         80400
_________________________________________________________________
dropout_2 (Dropout)          (None, None, 100)         0
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               80400
_________________________________________________________________
dropout_3 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 202
_________________________________________________________________
dropout_4 (Dropout)          (None, 2)                 0
_________________________________________________________________
activation_1 (Activation)    (None, 2)                 0
=================================================================
Total params: 3,503,438
Trainable params: 3,437,902
Non-trainable params: 65,536
```

Result for short inputs (Up to 500 bytes):

```
Testing with short input:
Evaluating:...
*** Model result ***
loss - 0.1164432018995285
categorical_accuracy - 1.0
Model has excellent performance! ;-)
```

Result for medium size inputs (Up to 4000 bytes):

```
Testing with medium size inputs:
Evaluating:...
*** Model result ***
loss - 5.9519370552152395e-05
categorical_accuracy - 1.0
Model has excellent performance! ;-)
```

Result for large size inputs (Up to 10,000 bytes):

```
Testing with large size inputs:
Evaluating:...
*** Model result ***
loss - 5.620559022645466e-05
categorical_accuracy - 1.0
Model has excellent performance! ;-)
```

Running this project

Recommended machine specs for running this project:

1. CPU – Intel i7 8th generation or above
2. Memory – 32 Gb DDRM
3. Disk - ~2GB of free disk space
4. OS: Project was developed using Windows 10 64bit, but should be Linux compatible
5. GPU – Nvidia 1060 GPU with 6GB of mem with CUDA installed and configured correctly for ML projects (Running the project without this can result in 10-100X slower training)

To install and run this project:

1. Install python 3.x on your machine (tested with version 3.7.1)
2. Install pip on your machine
3. Clone or download the sources
4. Go to the installation directory
5. Execute "pip install -r requirements.txt" to install all the project library requirements. Here is the list of the required libraries
    a. numpy (tested with version 1.18.4)
    b. pycryptodome (tested with version 3.9.7)
    c. keras (tested with version 2.3.1)
    d. tensorflow (tested with version 2.2.0)
    e. termplot (tested with version 0.0.2)
6. Download the English dictionary for this project
    a. For the English text generation this project requires the dictionary file named "glove.6B.50d.txt".
    b. The dictionary needs to be downloaded and saved into the /resources directory
    c. The dictionary can be downloaded from the following locations:
        i. https://www.kaggle.com/watts2/glove6b50dtxt
        ii. https://www.dropbox.com/s/nht3vh60gp6z9s5/glove6b50dtxt.zip (a mirror from my personal Dropbox account)
7. Execute "python main.py" to run the project
8. You will be prompted with a menu allowing you the select the trial you wish to run

Conclusion

One may conclude the following from the project results:

1. As expected, when a classical cipher is used, some statistical information emerges enabling some information to leak. (Trial #2, #3, #5, #10)
2. The ability to detect statistical information inside the ciphertext which was generated by modern cryptography is very limited and challenging. In this project the success on those was very limited. This may also require significantly more effort in research. As this was the result for ECB, I did not move forward on using different block methods as well. (Trial #6, #7)
3. As expected, the following can be observed from this study:
    a. OTP are powerful if used correctly (Trial #4)
    b. Short keys may enable information leakage (Trial #10)
    c. Changing keys is recommended (Trial #8, #9)