Expert system for security evaluation and its implementation on a Windows Platform

Ishika Prasad, Tejas Raval, Lipisha Chaudhary
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
(ip1262, tr7550, lc2919)@cs.rit.edu

I. EXECUTIVE SUMMARY

The project was spanned over three phases, the first phase consists of literature survey, specifications etc., based on this information our second phase of the project was implemented and our expert system was built. In this phase of the project we have done the approximation of the input and output of the previously implemented Expert System. We have used *TensorFlow [1]*, a Machine Learning library for approximation as well as prediction of the output from the Expert System.

As we know that, implementation of Expert System requires high computational power, using machine learning algorithms to predict/approximate output of the Expert System will definitely help. A dataset is constructed based on the matrices used in the previous phase and it is used as input to the machine learning model. The data is divided into learning and training data and fed as input to the machine learning model. By this process, the model is set to predict the output like the Expert System without actually having access to the rules of the Expert System.

II. REQUIREMENTS

We were required to build a model based on artificial neural networks (ANN) and multilayer perceptron (MLP). Artificial Neural Network is a computational model in which training data are provided to learn or train the model. The model learns on the basis of training data provided and based on the hidden layers. To test the data in the built model, we provide the test data to model for prediction of the accuracy of the ANN model.

In the previous part of the project, we had selected Windows OS for which we built security evaluation expert system. In this part of the project, we need to build a supervised machine learning model for which we have used ANN model. The dataset is produced by generating all possible combinations of matrices. The dataset is divided into a training set and testing set. The model is trained by training set and tested from training set. The output will provide the accuracy and loss of the test data.

Implementation includes designing, developing and testing a supervised learning model on Windows OS. We have to make our model reliable for use by end users and this will be designed to work on actual Windows PC. Additionally, we have to build our model to be robust and speed efficient.

III. IMPLEMENTATION

As mentioned, we have used *TensorFlow* library for implementing Machine Learning model. *TensorFlow* is Google's library which simplifies the development of Artificial Neural Network along with excelling at numerical computing which is critical for Deep Neural Network. Along with this, we have also used *keras [2]* a high-level library that is built on top of Tensor Flow for building Deep Artificial Neural Networks. We have used Sequential model for *keras* which is a linear stack of layer. In addition to this for prepping the data – splitting – we have used a python package called *pandas [3]*, which convert the data into the desired format in which the *keras* component can understand.

In our implementation we started by making combinations of all the inputs and running it through our expert system and classifying the input data points. This created out dataset to use as sets for training, validation and testing. We have divide the dataset into 70% trainings set and 30% testing set.

Step 1:

```
from keras.models import Sequential
from keras.layers import Dense
import pandas as pd
from matplotlib import pyplot

dataset = pd.read_csv("ES_dataset_rand.csv", delimiter=",")

X_train = dataset.iloc[0:280, 0:7].values
X_test = dataset.iloc[280:, 0:7].values
Y_train = dataset.iloc[0:280, 7:8].values
Y_test = dataset.iloc[280:, 7:8].values
```

Figure 1: Data splitting into training and testing set

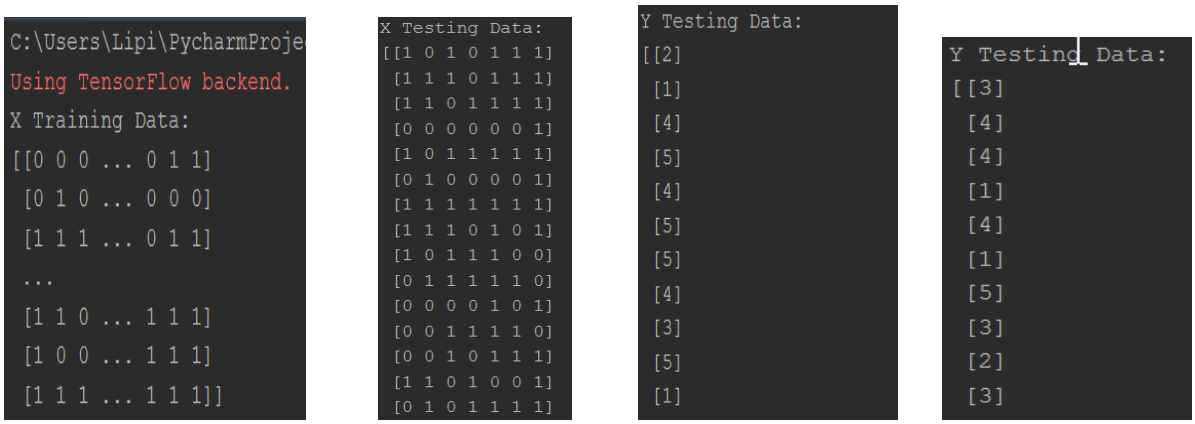


Figure 2: Output after splitting data

We have used a Sequential model this means we define layers in a linear stack. This in *keras* is done by adding layers, in addition to this we can pass number of neurons as parameters.

Step 2:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(neurons, activation='relu'))
model.add(Dense(neurons, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
```

Figure 3: Adding layers in the model

After adding the layers, we need to compile the model by defining some parameters pertaining to the requirement of the model. In here we have used an optimizer called *Adam* [4], this was used because the results were able to converge more accurately.

Step 3:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Figure 4: Model compile component

The model is then fit or trained to make the system act like a certain way. In this it was to trained to evaluate the security of a Windows OS platform.

```
model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Figure 5: Model fit component

After training the model we have to check whether the model has learned the required information correctly or not. This is done by evaluating the model for testing, the testing data set is passed to the model.evaluate() function. The input is the testing set and the labels/classes of the test dataset.

```
print(model.evaluate(X_test, Y_test))
```

Figure 6: Model evaluate component

The return output of this evaluate function is the error value and the accuracy of the test dataset.

Step 4:

This step shows the implementation of layers with having different number of neurons and the results of which are depicted the result section of this documentation. We had to repeat our iterations for 7 times.

Iteration 1:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(3, activation='relu'))
model.add(Dense(3, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Iteration 2:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(6, activation='relu'))
model.add(Dense(6, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Iteration 3:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Iteration 4:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Iteration 5:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(48, activation='relu'))
model.add(Dense(48, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Iteration 6:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(96, activation='relu'))
model.add(Dense(96, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Iteration 7:

```
model.add(Dense(32, activation='relu'))
model.add(Dense(192, activation='relu'))
model.add(Dense(192, activation='relu'))
model.add(Dense(6, activation='sigmoid'))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, Y_train, validation_split=0.33, epochs=35)
```

Step 5:

We repeated the 4th step almost for 7 iterations, which in turn reduced our error value to minimum i.e. as required less than 1%. We had the results as a gradual decreasing errors and an increasing accuracy of the model.

- **User guide**:

1. As this is a Python based code, Python IDE (Preferably, PyCharm) is required to run the code.
2. The Python version 3.6 is needed. Along with this, the various other Python. Packages needed are given below:
    a. TensorFlow
    b. Matplotlib [5]
    c. Panda
    d. Numpy
    e. Keras.

Generic command to install any package is:

pip install <package name>

3. To run the code, run the file with .py extension.

python filename.py

IV. EXPERIMENTS

The initial stage of the project the data set created did not provide our model with enough learning that can help our model to evolve with the knowledge gained. There were data points for class 0 & 5 which were not enough for the model to know that a particular data belongs to that class. We had to correct this arduous data, the one solution which we came up was to increase the data combination for each class and randomize this data set. After feeding this data to our model, the accuracy of the training was almost to a point where we could say that our model was ready for testing.

As specified in the previous section we had to iterate the model description over a certain number to lessen the error of the model fitting or testing. The following table gives the result of the iterations:

Number of neurons	Accuracy	Loss
3	0.28	1.65
6	0.36	1.5
12	0.46	1.25
24	0.78	0.44
48	0.89	0.29
96	0.89	0.21
192	0.98	0.07

Figure 7: Results of different iterations

As seen from the results the accuracy gradually increases and the error rate decreases as the number of neurons are doubled in number. There is a steady rate of accuracy seen after 48 number of neurons were used. It was observed that the number of neurone and the rate of accuracy are directly proportional while the rate of error is inversely proportional.

V. RESULTS:

Building an Expert system required us to classify the data achieved from the system or users into various classes. We had divided our decision of how to measure the security on five separate measures namely – Least secure, Low secure, Medium secure, Most secure, Highly secure. These classes acted as our labels for building up our ML model. As suggested we implemented a Multi-Layer Perceptron (MLP) model. In this model we have an input layer, several hidden layers and a single output layer. The basic of idea of this model is that the system or the methodology used should perform the functionality of an actual brain. The layer's fire neurons which are nothing but a threads containing some information for learning of the model. We first divide the dataset into two parts a training set and testing set. The training set coaches the model to learn the required knowledge and the testing data set checks whether whatever the model has learned is correct or not.

Along with evaluation we have predicted a few values for a test data set created. We considered a single data point on which studied the results achieved.

[0.0000000e+00 0.0000000e+00 3.5762787e-07 2.5212765e-05 **3.6358833e-06** 0.0000000e+00]

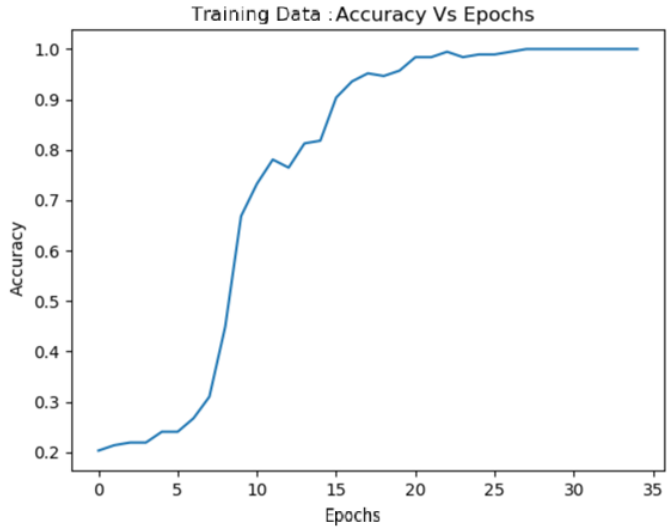
Figure 8: Results of predictions

The above results depict the confidence level of the classes in which the data point belongs to. The predict function returns a 10 number range points which act as the decision in which the data point belongs. We decide this by taking the maximum of the list numbers. Here the length of list represents the number of classes, so if a certain class has maximum value the data point belongs to that class. In the above example we can clearly see that the data point taken belongs to the class 5 i.e**. Most Secure**.

```
predictions_test = model.predict(X_test)
print(predictions_test[0])
```

Figure 9: Predict function snippet

- Graphs:
    a. Training data: Accuracy vs. Epochs



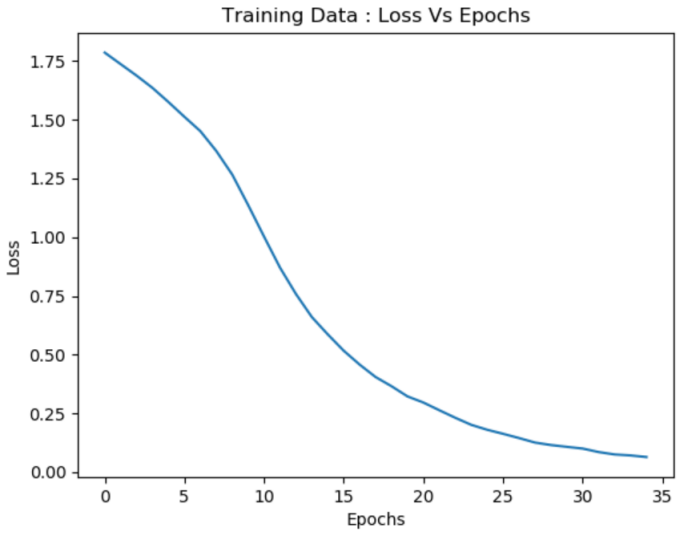
This is the graph for training data for Accuracy vs. Epoch.

Detailed explanations are described below:

1. When Epoch is in range of 0 - 5 then the Accuracy is in the range of 0.2 - 0.25
2. When Epoch is in range of 6 - 10 then the Accuracy bumped in range from 0.25 to 0.75
3. When Epoch is in range of 11 - 23 then the Accuracy is in range of 0.75 - 0.95
4. When Epoch is in range of 24 - 33 then the Accuracy goes up to 1

In conclusion, for the above graph, increase in the Epoch will increase the Accuracy. This shows Epoch and Accuracy is directly proportional to each other.

b. Training data: Loss vs. Epochs



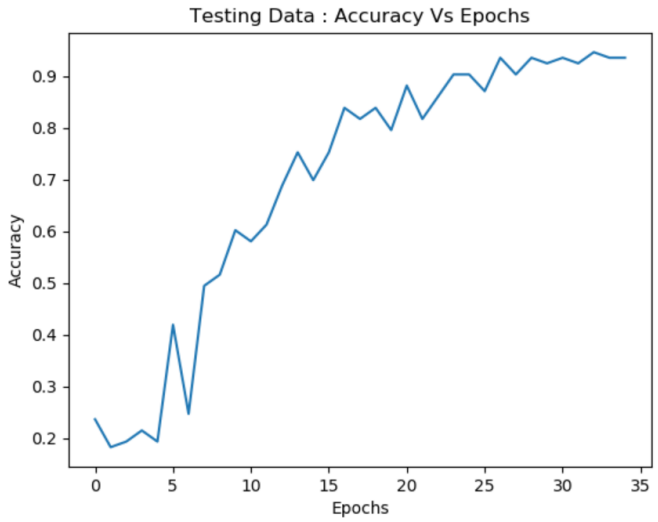
This is the graph for training data for Loss vs. Epoch.

Detailed explanations are given below.

1. When Epoch is in range of 0 - 5 then the Loss is in the range of 1.75 - 1.50
2. When Epoch is in range of 5 - 15 then the Loss undergoes a steep decrease in the range of 1.50 - 0.50
3. When Epoch is in range of 15 - 25, the loss continues to steadily fall and is in the range, 0.50 - 0.25.
4. When Epoch is in range of 25 - 35, the loss is steady and is almost 0.10

In conclusion, for the above graph, increase in the Epoch will decrease the Loss. This shows Epoch and Loss is inversely proportional to each other.

c. Testing data: Accuracy vs. Epochs



This is the graph for testing data for Accuracy vs Epoch.

Detailed explanations are described below:

1. When Epoch is in range of 0 - 5 then the Accuracy is in the range of 0.2 - 0.4
2. When Epoch is in range of 6 - 10 then the Accuracy is in range of 0.25 to 0.6
3. When Epoch is in range of 11 - 23 then the Accuracy is in range of 0.6 - 0.8
4. When Epoch is in range of 24 - 33 then the Accuracy goes up to 0.84

In conclusion, for the above graph, increase in the Epoch will increase the Accuracy. This shows Epoch and Accuracy is directly proportional to each other.

d. Testing data: Loss vs. Epochs

This is the graph for testing data for Loss vs. Epoch.

Detailed explanations are given below.

1. When Epoch is in range of 0 - 5 then the Loss is in the range of 1.75 - 1.40
2. When Epoch is in range of 5 - 15 then the Loss undergoes a steep decrease in the range of 1.40 - 0.40
3. When Epoch is in range of 15 - 25, the loss continues to steadily fall and is in the range, 0.40 - 0.2.
4. When Epoch is in range of 25 - 35, the loss is steady and is almost 0.18

In conclusion, for the above graph, increase in the Epoch will decrease the Loss. This shows Epoch and Loss is inversely proportional to each other.

VI. CONCLUSION

In conclusion the ANN component of the Expert system can be used to predict meaningful conclusions which can help in analysing the security of the OS platform more easily. Although training the model can take a lot of work but once done correctly it can do wonders in accurately predicting data that a human person will take plenty of time to compute, because of the complex computations. The accuracy of our model is up to 84% after testing of the data and the total time taken to complete this process is 6.91 milliseconds, CPU consumption is 31.1%.

VII. REFERENCES

[1] https://www.tensorflow.org/

[2] https://keras.io/

[3] https://pandas.pydata.org/pandas-docs/stable/

[4] https://keras.io/optimizers/

[5] https://matplotlib.org/