Algorithmics for Data Mining

Deliverable 3: Detecting Intrusions Using Data Mining Techniques

Ali Arabyarmohammadi June 2022

1 Introduction

Maintaining the security of a network system is critical in today's world. In order to protect ourselves from hackers, we need a secured and safe network infrastructure. In a network, an intrusion detection system is used to detect various forms of attacks. IDS come in a variety of configurations, including network-based, host-based, and hybrid, depending on the technology they detect in the market. We need a safe and reliable network system because the current system does not give that level of security.
In this work, we study intrusion detection systems (IDS) that use a **Decision Tree** and **Particle Swarm Optimization (PSO)** technique to efficiently identify intruder attacks.
In order to implement these mentioned data mining methods, python was used, and the also libraries **Sklearn** and **Zoofs** libraries (for Particle Swarm Optimization-PSO) are used.

```python
# Load Libraries
import warnings
from IPython import get_ipython
warnings.filterwarnings("ignore")
import itertools
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix,accuracy_score,recall_score,
    precision_score,f1_score
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
from zoofs import ParticleSwarmOptimization
from sklearn.metrics import accuracy_score, precision_score, f1_score,
    recall_score
from sklearn.tree import DecisionTreeClassifier
```

Listing 1: Python libraries were used

2 Data Understanding

Using **pandas** library, the dataset file **KDDTest+.csv** and **KDDTrain+.csv** was loaded. The database is used adopted from **Canadian Institute for Cybersecurity** and the website **https://www.unb.ca/cic/datasets/nsl.html** which is contained a data set suggested to solve some of the inherent problems. In fact, the NSL-KDD data set is a new version of the KDD'99 data collection. This is a useful benchmark data set for academics to use when comparing various intrusion detection systems.

The setting is composed by 1 training set and 2 testing set:

- **KDDTrain+**: The full NSL-KDD **train** set including attack-type labels in CSV format
- **KDDTest+**: The full NSL-KDD **test** set including attack-type labels in CSV format.

Using these two lines of code below, we are reading the datasets.

```
training_df = pd.read_csv('KDDTrain+.csv', header=None)
testing_df = pd.read_csv('KDDTest+.csv', header=None)
```

Listing 2: Loading the datasets

Then by writing the commands:

```
training_df.head()
testing_df.head()
```

Listing 3: Printing the five top rows of the tables

Five top rows of the tables are shown below.

	0	1	2	3	4	5	6	7	8	9	...	33	34	35	36	37	38	39	40	41	42
0	0	tcp	ftp_data	SF	491	0	0	0	0	0	...	0.17	0.03	0.17	0.00	0.00	0.00	0.05	0.00	normal	20
1	0	udp	other	SF	146	0	0	0	0	0	...	0.00	0.60	0.88	0.00	0.00	0.00	0.00	0.00	normal	15
2	0	tcp	private	S0	0	0	0	0	0	0	...	0.10	0.05	0.00	0.00	1.00	1.00	0.00	0.00	neptune	19
3	0	tcp	http	SF	232	8153	0	0	0	0	...	1.00	0.00	0.03	0.04	0.03	0.01	0.00	0.01	normal	21
4	0	tcp	http	SF	199	420	0	0	0	0	...	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	normal	21

5 rows × 43 columns

Figure 1: Training dataset

	0	1	2	3	4	5	6	7	8	9	...	33	34	35	36	37	38	39	40	41	42
0	0	tcp	private	REJ	0	0	0	0	0	0	...	0.04	0.06	0.00	0.00	0.0	0.0	1.00	1.00	neptune	21
1	0	tcp	private	REJ	0	0	0	0	0	0	...	0.00	0.06	0.00	0.00	0.0	0.0	1.00	1.00	neptune	21
2	2	tcp	ftp_data	SF	12983	0	0	0	0	0	...	0.61	0.04	0.61	0.02	0.0	0.0	0.00	0.00	normal	21
3	0	icmp	eco_i	SF	20	0	0	0	0	0	...	1.00	0.00	1.00	0.28	0.0	0.0	0.00	0.00	saint	15
4	1	tcp	telnet	RSTO	0	15	0	0	0	0	...	0.31	0.17	0.03	0.02	0.0	0.0	0.83	0.71	mscan	11

5 rows × 43 columns

Figure 2: Testing dataset

There are 41 columns in the NSL-KDD dataset. Name of each column are listed below.

Number	Data features	Number	Data features	Number	Data features	Number	Data features
1	Duration	12	Logged_in	23	Count	34	Dst_host_same_srv_rate
2	Protocol_type	13	Num_compromised	24	Srv_count	35	Dst_host_diff_srv_rate
3	Service	14	Root_shell	25	Serror_rate	36	Dst_host_same_src_port_rate
4	Flag	15	Su_attempted	26	Srv_serror_rate	37	Dst_host_srv_diff_host_rate
5	Src_bytes	16	Num_root	27	Rerror_rate	38	Dst_host_serror_rate
6	Dst_bytes	17	Num_file_creations	28	Srv_rerror_rate	39	Dst_host_srv_serror_rate
7	Land	18	Num_shells	29	Same_srv_rate	40	Dst_host_rerror_rate
8	Wrong_fragment	19	Num_access_files	30	Diff_srv_rate	41	Dst_host_srv_rerror_rate
9	Urgent	20	Num_outbound_cmds	31	Srv_diff_host_rate		
10	Hot	21	Is_host_login	32	Dst_host_count		
11	Num_failed_logins	22	Is_guest_login	33	Dst_host_srv_count		

Figure 3: Dataset features

The next table is showing a summary of the normal and 4 other type of attacks is consider in the dataset.

Category	Train	Test
Normal	67343	9711
Dos	11656	7458
Probe	45927	2421
U2R	52	200
R2L	995	2754
Total	**125973**	**22544**

Table 1: Description of NSL-KDD dataset

The suggested intrusion detection system, which comprises 41 features and five classifications, was evaluated using the NSL-KDD database (Normal, DOS, R2L, U2R, and Probe). There are two sections to this data set: TrainSet and TestSet. The Dos class is made up of records that use system resources while denying standard requests. The R2L class contains evidence that an intruder connected to the victim's system remotely and used the user's legal account. An intruder successfully obtaining control of a victim system is recorded in the U2R class. Intruders attempting to gather information about network services are likewise recorded in the Probe class.

2.1 Data Preparation

At the beginning of the code block below, we will give names to each column according to our dataset description of factors.

```
columns = [
    'duration',
    'protocol_type',
    'service',
    'flag',
    'src_bytes',
    'dst_bytes',
    'land',
    'wrong_fragment',
    'urgent',
    'hot',
    'num_failed_logins',
    'logged_in',
    'num_compromised',
    'root_shell',
    'su_attempted',
    'num_root',
    'num_file_creations',
    'num_shells',
    'num_access_files',
    'num_outbound_cmds',
    'is_host_login',
    'is_guest_login',
    'count',
    'srv_count',
    'serror_rate',
    'srv_serror_rate',
    'rerror_rate',
    'srv_rerror_rate',
    'same_srv_rate',
    'diff_srv_rate',
    'srv_diff_host_rate',
    'dst_host_count',
    'dst_host_srv_count',
    'dst_host_same_srv_rate',
    'dst_host_diff_srv_rate',
    'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate',
    'dst_host_serror_rate',
    'dst_host_srv_serror_rate',
    'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate',
    'outcome',
    'difficulty'
]
training_df.columns = columns
testing_df.columns = columns
```

Listing 4: Naming each column

Here we will print the number of records for our train and test data tables.

```
print("Training set has {} rows.".format(len(training_df)))
print("Testing set has {} rows.".format(len(testing_df)))
```

Listing 5: Number of records

- Training set has 125973 rows.
- Testing set has 22543 rows.

In the next block of code, we will print all possible outcomes.

```
training_outcomes=training_df["outcome"].unique()
testing_outcomes=testing_df["outcome"].unique()
print("The training set has {} possible outcomes \n".
format(len(training_outcomes)) )
print(", ".join(training_outcomes)+".")
print("\nThe testing set has {} possible outcomes \n".
format(len(testing_outcomes)))
print(", ".join(testing_outcomes)+".")
```

Listing 6: Print the outcomes

The training set has 23 possible outcomes.

neptune, normal, saint, mscan, guess_passwd, smurf, apache2, satan, buffer_overflow, back, warezmaster, snmpgetattack, processtable, pod, httptunnel, nmap, ps, snmpguess, ipsweep, mailbomb, portsweep, multihop, named, sendmail, loadmodule, xterm, worm, teardrop, rootkit, xlock, perl, land, xsnoop, sqlattack, ftp_write, imap, udpstorm, phf.

The **testing** set has 38 possible outcomes.

neptune, normal, saint, mscan, guess_passwd, smurf, apache2, satan, buffer_overflow, back, warezmaster, snmpgetattack, processtable, pod, httptunnel, nmap, ps, snmpguess, ipsweep, mailbomb, portsweep, multihop, named, sendmail, loadmodule, xterm, worm, teardrop, rootkit, xlock, perl, land, xsnoop, sqlattack, ftp_write, imap, udpstorm, phf.

A list of attack names that belong to each general attack type:

```
dos_attacks=["snmpgetattack","back","land","neptune","smurf","teardrop","pod","
    apache2","udpstorm","processtable","mailbomb"]
r2l_attacks=["snmpguess","worm","httptunnel","named","xlock","xsnoop","sendmail","
    ftp_write","guess_passwd","imap","multihop","phf","spy","warezclient","
    warezmaster"]
u2r_attacks=["sqlattack","buffer_overflow","loadmodule","perl","rootkit","xterm","
    ps"]
probe_attacks=["ipsweep","nmap","portsweep","satan","saint","mscan"]
```

Listing 7: A list of attack names

Our new labels:

```
classes=["Normal","Dos","R2L","U2R","Probe"]
```

Helper function to label samples to 5 classes:

```
def label_attack (row):
    if row["outcome"] in dos_attacks:
        return classes[1]
    if row["outcome"] in r2l_attacks:
        return classes[2]
    if row["outcome"] in u2r_attacks:
        return classes[3]
    if row["outcome"] in probe_attacks:
        return classes[4]
    return classes[0]
```

Listing 8: Helper function

Then we combine the datasets temporarily to do the labeling.

```
 test_samples_length = len(testing_df)
df=pd.concat([training_df,testing_df])
df["Class"]=df.apply(label_attack,axis=1)
```

Listing 9: combine the datasets temporarily

The old outcome field is dropped since it was replaced with the Class field, the difficulty field will be dropped as well.

```
df=df.drop("outcome",axis=1)
df=df.drop("difficulty",axis=1)
```

Listing 10: Python libraries were used

Now we again split the data into training and test sets.

```
training_df= df.iloc[:-test_samples_length, :]
testing_df= df.iloc[-test_samples_length:,:]
```

Listing 11: Python libraries were used

After pre-processing of our data, we again print the outcomes.

```
 training_outcomes=training_df["Class"].unique()
testing_outcomes=testing_df["Class"].unique()
print("The training set has {} possible outcomes \n".format(len(training_outcomes)
    ) )
print(", ".join(training_outcomes)+".")
print("\nThe testing set has {} possible outcomes \n".format(len(testing_outcomes)
    ))
print(", ".join(testing_outcomes)+".")
```

Listing 12: print the outcomes again.

The training set has 5 possible outcomes.

Normal, Dos, R2L, Probe, U2R.

The **testing** set has 5 possible outcomes.

Dos, Normal, Probe, R2L, U2R.

Now is the time for normalizing the data. For the numerical data we will user **Normalization** and for non-numeric, the method a helper function for **one hot encoding** is implemented. We have written a Helper function for one hot encoding for scaling continuous values and another **Helper function** for one hot encoding

```

# Helper function for scaling continous values
def minmax_scale_values(training_df,testing_df, col_name):
    scaler = MinMaxScaler()
    scaler = scaler.fit(training_df[col_name].values.reshape(-1, 1))
    train_values_standardized = scaler.transform(training_df[col_name].values.
   reshape(-1, 1))
    training_df[col_name] = train_values_standardized
    test_values_standardized = scaler.transform(testing_df[col_name].values.
   reshape(-1, 1))
    testing_df[col_name] = test_values_standardized
#Helper function for one hot encoding
def encode_text(training_df,testing_df, name):
    training_set_dummies = pd.get_dummies(training_df[name])
    testing_set_dummies = pd.get_dummies(testing_df[name])
```

```
    for x in training_set_dummies.columns:
        dummy_name = "{}_{}".format(name, x)
        training_df[dummy_name] = training_set_dummies[x]
        if x in testing_set_dummies.columns :
            testing_df[dummy_name]=testing_set_dummies[x]
        else :
            testing_df[dummy_name]=np.zeros(len(testing_df))
    training_df.drop(name, axis=1, inplace=True)
    testing_df.drop(name, axis=1, inplace=True)
sympolic_columns=["protocol_type","service","flag"]
label_column="Class"
for column in df.columns :
    if column in sympolic_columns:
        encode_text(training_df,testing_df,column)
    elif not column == label_column:
        minmax_scale_values(training_df,testing_df, column)
```

Listing 13: **Normalization** and **One hot encoding**

After this step we print our dataset to see the changes.

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	...	flag_REJ	flag_RSTO	flag_R
0	0.0	3.558064e-07	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0	0	
1	0.0	1.057999e-07	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0	0	
2	0.0	0.000000e+00	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0	0	
3	0.0	1.681203e-07	6.223962e-06	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0	0	
4	0.0	1.442067e-07	3.206260e-07	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0	0	

5 rows × 123 columns

Figure 4: Training dataset

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	...	flag_REJ	flag_RSTO	flag_R
0	0.000000	0.000000e+00	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1	0	
1	0.000000	0.000000e+00	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1	0	
2	0.000047	9.408217e-06	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0	0	
3	0.000000	1.449313e-08	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0	0	
4	0.000023	0.000000e+00	1.145093e-08	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0	1	

5 rows × 123 columns

Figure 5: Testing dataset

Then we will Set Attack and Normal Classes.

```
# Classes[0] = 'Normal'
# Classes[1] = 'Dos'
# Classes[2] = 'R2L'
# Classes[3] = 'U2R'
# Classes[4] = 'Probe'
y0=np.ones(len(y),np.int8)
y0[np.where(y==classes[0])]=0
y0_test=np.ones(len(y_test),np.int8)
y0_test[np.where(y_test==classes[0])]=0
```

Listing 14: Setting Attack and Normal Classes

3 Modeling

The high amount of information and a large number of aspects of each attack are two of the challenges in designing intrusion detection systems. The existence of a significant number of these unconnected and redundant features in the data set degrades the machine learning algorithm's performance and adds to the computational complexity.
In this project, we will be combining Particle Swarm Optimization and Decision Tree Algorithms to detect Intrusion in Networks. Because the intrusion detection data contains a high number of features, **Particle Swarm Optimization** (PSO) was used to pick a subset of desired features in this investigation. A model is then shown that uses the usual **Decision Tree** data mining technique to classify the data and detect infiltration. To accomplish this task, we used libraries. Sklearn and Zoofs (for PSO).

```python
def numpy2dataframe(nparray):
    panda_df = pd.DataFrame(data = nparray,
                            index = ['Row_' + str(i + 1)
                            for i in range(nparray.shape[0])],
                            columns = ['Column_' + str(i + 1)
                            for i in range(nparray.shape[1])])
    return panda_df


def objective_function_topass(model,X_train, y_train, X_valid, y_valid):
    model.fit(X_train,y_train)
    P=accuracy_score(y_valid, model.predict(X_valid))

    return P



algo_object=ParticleSwarmOptimization(objective_function_topass,n_iteration=4,
    population_size=4,minimize=False)


xtrain_df = numpy2dataframe(x)
xtest_df = numpy2dataframe(x_test)

clf = DecisionTreeClassifier(random_state=0)
best_feature_list = algo_object.fit(clf, xtrain_df, pd.DataFrame(y), xtest_df, pd.
    DataFrame(y_test), verbose=True)
algo_object.plot_history()
```

Listing 15: DecisionTreeClassifier and ParticleSwarmOptimization

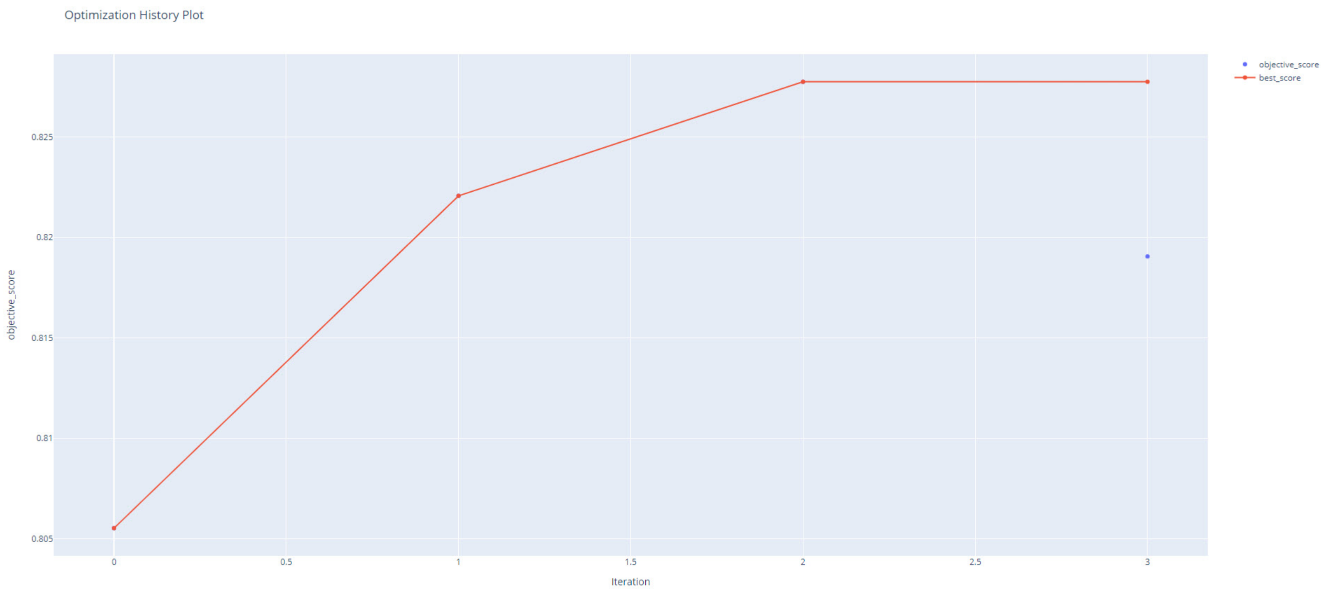
For our optimization algorithm, an **objective function** is defined based on the **accuracy score** of the model.
We could have chosen other parameters for the objective function like *recall_score*, *precision_score*, and *f1_score*. But here we have decided to optimize our model using the ***accuracy_score***.
Then, for classification, we used the default parameters of the decision tree classifier (DecisionTreeClassifier).

4 Results

The PSO algorithm determines the number of effective features for classification in an automated manner. The results of the experiments reveal that the proposed strategy is quite functional. From the optimization history plot, it can be seen that in four iterations, the ***objective_ score*** is improved dramatically by the PSO.

Numerical results in 4 iterations.

iteration 0: objective value 0.8055272146564344. Current best value is 0.8055272146564344
iteration 1: objective value 0.8220733708911857. Current best value is 0.8220733708911857
iteration 2: objective value 0.827751408419465. Current best value is 0.827751408419465
iteration 3: objective value 0.8190569134542873. Current best value is 0.827751408419465



Knowing the best feature set for each type of attack is another necessity of intrusion detection systems. Because in this situation, the intrusion detection system will only be able to identify one set of features that are specific to that attack, rather than being able to detect any form of attack. It is suggested that in future studies, a model with this design capability and performance be evaluated.

References

[1] Shivangee Agrawal, Gaurav Jain, *A Review on Intrusion Detection System Based Data Mining Techniques* International Research Journal of Engineering and Technology (IRJET), e-ISSN: 2395-0056, p-ISSN: 2395-0072, Volume: 04 Issue: 09 | Sep -2017.

[2] Amin Rezaeipanaha, Musa Mojarad,Samaneh Sechin Matoor, *Intrusion Detection in Computer Networks Through Combining Particle Swarm Optimization and Decision Tree Algorithms* March, 1(1), 14-22., March 2021.

[3] Mohamed El Bekri, Ouafaa Diouri, *Pso Based Intrusion Detection: A Preimplementation* ScienceDirect, Procedia Computer Science 160 (2019) 837–842.International Workshop on Emerging Networks and Communications (IWENC 2019), November 4-7, 2019, Coimbra, Portugal.