Hands-on Lab Description

CS-ML-00200 – Python Machine Learning Data Processing Modules

Category:

```
CS-ML: Machine Learning
```

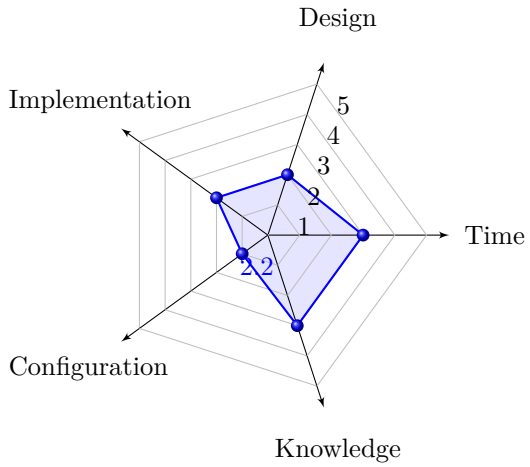
Objectives:

```
1 Learn Python-based data pre-processing
2 Learn the procedure to processing data to address a data science problem
```

Estimated Lab Duration:

```
1 Expert: 120 minutes
2 Novice: 360 minutes
```

Difficulty Diagram:



Difficulty Table.

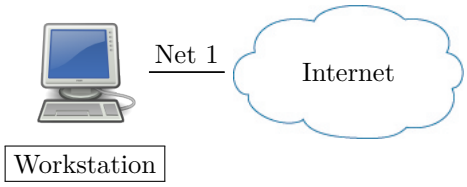
Measurements	Values (0-5)
Time	3
Design	2
Implementation	2
Configuration	1
Knowledge	3
Score (Average)	2.2

Required OS:

```
Linux: Ubuntu 18.04 LTS
```

Lab Running Environment:

```
VirtualBox https://www.virtualbox.org/ (Reference Labs: CS-SYS-00101)
```




```
1 Server: Linux (Ubuntu 18.04 LTS)
2 Network Setup: connected to the Internet
```

Lab Preparations:

```
Python and Anaconda software packages installed on the Linux VM. Reference Lab:
    CS-ML-00001
NSL-KDD dataset. Reference Lab: CS-ML-00101
Python Machine Learning Tutorial-Basic Concepts: CS-ML-00199
```

Lab Overview

In this lab you will practice data pre-processing techniques using Python. The first part of the lab will give a few simple examples to illustrate basic concepts of data pre-processing such as scaling, normalization, binarization, one hot encoding, and label encoding; later, the lab will present a few examples including how to load a dataset and extract label data, how to generate feature mapping of a given dataset, how to create a customized dataset, and how to create and use a user defined data pre-processing module.

The assessment of the lab is based on the practice of given Python programs running on NSL-KDD datasets. Students need to self-assess the performance and understanding of given Python programs and produced results.

Task 1 Preparation of setting up lab environment

Suggestions:

1. Review and exercise the following labs CS-ML-00001 (How to setup ML running environment),
2. Review CS-ML-00199 (Python Machine Learning Concepts)

Note that the ML running environment such as Anaconda may have already been setup on your workstation. You need to verify if these servers are setup properly to conduct your required tasks.

In this lab, several ML Python programs are provided for exercise. You can download from the lab resource repository. You can download lab resource files by using following *wget* command. Note that check if *wget* is installed using command

```
$    wget --version
```

If *wget* is not installed, you can install it using the following command:

```
$    sudo apt install wget
```

And then, download the zipped lab file:

```
$    wget https://raw.githubusercontent.com/SaburH/CSE548/main/lab-cs-ml-00200.zip

$    unzip lab-cs-ml-00200.zip
$    cd lab-cs-ml-00200
```

In the folder, the following Python programs exist for testing and exercising:

```
1  NSL-KDD (folder) dataset
2  categoryMappings (folder) contains the category data and mappings generated by using
       categoryMapper.py on NSL-KDD dataset
3  pre-processing-sample.py: It is a Python example file to show some basic data pre-
       processing methods.
4  distinctLabelExtractor.py: this Python program loads an NSL-KDD file and extracts
       attack names and types.
5  DataExtractor.py: this Python program creates customizable training and testing
       datasets by extracting a subset of attack classes from the training and testing
       datasets.
6  categoryMapper.py: this Python program creates labeled category data for string-based
        features.
```

```
7  data_preprocesor.py: this Python program creates a Python library that can ease the
       data pre-processing procedure of Python-based ML data processing solutions.
```

Task 2 Data Pre-processing Using Python

In Machine Learning (ML), we usually come across lots of raw data which is not fit to be readily processed by ML algorithms. The raw data need to be pre-processed before feed them into various ML algorithms. The data pre-processing becomes a crucial step and may impact the ML analysis results dramatically if carefully design and processed. Here, we present a few data pre-processing techniques using Python.

In this section, we use the simple *pre-processing-sample.py* as an example with the following content at its beginning:

```
import numpy as np
from sklearn import preprocessing
# Sample data
input_data = np.array([[3, -1.5, 3, -6.4], [0, 3, -1.3, 4.1], [1, 2.3, -2.9, -4.3]])
```

Task 2.1 Mean Removal

It involves removing the mean from each feature so that it is centered on zero. Mean removal helps in removing any bias from the features.

You can use the following code for mean removal:

```
data_standardized = preprocessing.scale(input_data)
print("\n Mean = ", data_standardized.mean(axis = 0))
print("Std deviation = ", data_standardized.std(axis = 0))
```

Now run the command

```
$    python prefoo.py
```

The output should looks like:

```
Mean = [ 5.55111512e-17 -3.70074342e-17 0.00000000e+00 -1.85037171e-17]
Std deviation = [1. 1. 1. 1.]
```

Observe that in the output, mean is almost 0 and the standard deviation is 1.

Task 2.2 Scaling

The values of every feature in a data point can vary between random values. So, it is important to scale them so that this matches specified rules.

You can use the following code for scaling

```
data_scaler = preprocessing.MinMaxScaler(feature_range = (0, 1))
data_scaled = data_scaler.fit_transform(input_data)
print("\nMin max scaled data = ", data_scaled)
```

Now run the code and you can observe the following output

```
Min max scaled data = [[1.       0.        1.        0.       ]
```

```
[0.         1.         0.27118644 1.        ]
[0.33333333 0.84444444 0.         0.2       ]]
```

Then, all the values have been scaled between the given range.

Task 2.3 Normalization

Normalization involves adjusting the values in the feature vector so as to measure them on a common scale. Here, the values of a feature vector are adjusted so that they sum up to 1. We can add the following lines to the prefoo.py file:

```
data_normalized = preprocessing.normalize(input_data, norm = 'l1')
print("\nL1 normalized data = ", data_normalized)
```

The code will show the following results:

```
L1 normalized data = [
[ 0.21582734 -0.10791367 0.21582734 -0.46043165]
[ 0.         0.35714286 -0.1547619 0.48809524]
[ 0.0952381  0.21904762 -0.27619048 -0.40952381]]
```

Normalization is used to ensure that data points do not get boosted due to the nature of their features.

Task 2.4 Binarization

Binarization is used to convert a numerical feature vector into a Boolean vector. You can use the following code for binarization:

```
data_binarized = preprocessing.Binarizer(threshold=1.4).transform(input_data)
print("\nBinarized data =", data_binarized)
```

The code will produce the following results:

```
Binarized data = [
[1. 0. 1. 0.]
[0. 1. 0. 1.]
[0. 1. 0. 0.]]
```

This technique is helpful when we have prior knowledge of the data.

Task 2.5 One Hot Encoding

It may be required to deal with numerical values that are few and scattered, and you may not need to store these values. In such situations you can use One Hot Encoding technique.

If the number of distinct values is k, it will transform the feature into a k-dimensional vector where only one value is 1 and all other values are 0.

You can use the following code for one hot encoding:

```
encoder = preprocessing.OneHotEncoder()
encoder.fit([ [0, 2, 1, 12],
[1, 3, 5, 3],
[2, 3, 2, 12],
[1, 2, 4, 3]
])
print("\nPrint encoder:", encoder)
print("Print encoder categories:", encoder.categories_)
```

```
encoded_vector = encoder.transform([[2, 3, 5, 3]]).toarray()
print("Encoded vector =", encoded_vector)
```

Then, it generates results:

```
Print encoder: OneHotEncoder(categories='auto', drop=None, dtype=<class
    'numpy.float64'>,
handle_unknown='error', sparse=True)
Print encoder categories: [array([0, 1, 2]), array([2, 3]), array([1, 2, 4, 5]),
    array([ 3, 12])]
Encoded vector = [[0. 0. 1. 0. 1. 0. 0. 0. 1. 1. 0.]]
```

In this example, the *fit* method will convert the matrix into a concise format in that each feature only show each category once, i.e.,

```
[array([0, 1, 2]), array([2, 3]), array([1, 2, 4, 5]), array([ 3, 12])]
```

Then, the transform will only consider each value of the input vector *[2, 3, 5, 3]* for each feature. For example, let's consider the third feature in each feature vector. The values are 1, 2, 4, 5, after fitting. There are four separate values here, which means the one-hot encoded vector will be of length 4. If we want to encode the value 5, which is the third value of the transform vector, it will be a vector [0, 0, 0, 1]. Similarly, other three vectors will generate [0,0,1], [0,1], and [1,0] corresponding to [2,3, , 3] in the transform vector. In this transform, only one value can be 1 in each feature. Thus, it generates 11 values in total: [0. 0. 1. 0. 1. 0. 0. 0. 1. 1. 0.].

Task 2.6 Label Encoding

In supervised learning, we mostly come across a variety of labels which can be in the form of numbers or words. If they are numbers, then they can be used directly by the algorithm. However, many times, labels need to be in readable form. Hence, the training data is usually labeled with words. Label encoding refers to changing the word labels into numbers so that the algorithms can understand how to work on them.

```
label_encoder = preprocessing.LabelEncoder()
input_classes = ['suzuki', 'ford', 'suzuki', 'toyota', 'ford', 'bmw']
label_encoder.fit(input_classes)
print("\nClass mapping (words are mapped to index values):")
for i, item in enumerate(label_encoder.classes_):
print(item, '-->', i) # for loop needs indent in python
```

Then, the output is:

```
Class mapping (words are mapped to index values):
bmw --> 0
ford --> 1
suzuki --> 2
toyota --> 3
```

In this example, words have been changed into 0-indexed numbers. Now, when we deal with a set of labels, we can transform them as follows:

```
labels = ['toyota', 'ford', 'suzuki']
encoded_labels = label_encoder.transform(labels)
print("\nLabels =", labels)
print("Encoded labels =", list(encoded_labels))
```

The results will show as:

```
Labels = ['toyota', 'ford', 'suzuki']
```

```
Encoded labels = [3, 1, 2]
```

You can also check by transforming numbers back to word labels as shown in the code here:

```
encoded_labels = [3, 2, 0, 2, 1]
decoded_labels = label_encoder.inverse_transform(encoded_labels)
print("\nEncoded labels =", encoded_labels)
print("Decoded labels =", list(decoded_labels))
```

The results show as follow:

```
Encoded labels = [3, 2, 0, 2, 1]
Decoded labels = ['toyota', 'suzuki', 'bmw', 'suzuki', 'ford']
```

Task 3 Loading Dataset and Extract Labels

In this task, let's look into a real data analysis example and extract data labels from a labeled dataset file. The illustration is based on the provided Python program file *distinctLabelExtractor.py*. The Python codes are presented as follows:

```
import pandas as pd

# Data file Path
DatasetPath='NSL-KDD/'
# Data file name
dataset_filename='KDDTrain+.txt'

#All attacks in NSL-KDD classed based on their attack classes: DoS, Prob, U2R, and R2L
attacks_subClass = [['apache2', 'back', 'land', 'neptune', 'mailbomb', 'pod', 'processtable',
    'smurf', 'teardrop', 'udpstorm', 'worm'],
['ipsweep', 'mscan', 'portsweep', 'saint', 'satan'],
['buffer.overflow', 'loadmodule', 'perl', 'ps', 'rootkit', 'sqlattack', 'xterm'],
['ftp.write', 'guess.passwd', 'httptunnel', 'imap', 'multihop', 'named', 'phf', 'sendmail',
    'snmpgetattack', 'spy', 'snmmpguess', 'warezclient', 'warezserver', 'xlock', 'xsnoop']
]

# Four attack classes
expectedAttackClasses = ['DoS (A1)', 'Probe (A2)', 'U2R (A3)', 'R2L (A4)']

# Load data
dataset = pd.read_csv(DatasetPath + dataset_filename, header=None, encoding="ISO-8859-1")

# Read values
evaluator = dataset.iloc[:, :].values

# Initialize empty subClass and currentAttackClasses in the loaded file
subClasses = []
currentAttackClasses = []

# The for loop check if the attack identified, then it update subClasses and currentAttackClasses
for i in range(len(evaluator)):
subClass = str.lower(evaluator[i, -2])
if subClass not in subClasses:
subClasses.append(subClass)
```

```
33  for i in range(len(attacks_subClass)):
34  if subClass in attacks_subClass[i] and expectedAttackClasses[i] not in currentAttackClasses:
35  currentAttackClasses.append(expectedAttackClasses[i])
36
37  # Print finding results:
38  print("\nSub classes of Attacks")
39  print(subClasses)
40  print("\n\nAttack Classes")
41  print(currentAttackClasses)
```

At the beginning of the example codes, *panda* package is imported to allow file related methods. Line 3 to Line 6 defined the file name to be loaded and its data path relevant to the Python program.

The analyzed data file is from NSL-KDD file. For more details, please refer to the lab CS-ML-00101 (Understanding NSL-KDD Dataset). NSL-KDD is comprised of four sub datasets: KDDTest+, KDDTest-21, KDDTrain+, KDDTrain+_20Percent, although KDDTest-21 and KDDTrain+_20Percent are subsets of the KDDTrain+ and KDDTest+.

The NSL-KDD data set exists 4 different classes of attacks: Denial of Service (DoS), Probe, User to Root(U2R), and Remote to Local (R2L), and altogether there are 39 sub-classes of attacks. They are summarized in Table CS-ML-00200.2.

Table CS-ML-00200.2
Classification of NSL-KDD Dataset Attacks

Classes	Dos	Probe	U2R	R2L
Sub-classes	apache2 back land neptune mailbomb pod processtable smurf teardrop udpstorm worm	ipsweep mscan nmap portsweep saint satan	buffer_overflow loadmodule perl ps rootkit sqlattack xterm	ftp_write guess_passwd httptunnel imap multihop named phf sendmail snmpgetattack spy snmpguess warezclient warezserver xlock xsnoop
Total	11	6	7	15

The sub-classes of attacks are defined in *attacks_subClass*, and the classes of attacks are defined in *expectedAttackClasses*. Line 19 loads a file into *dataset* and then line 22 reads the values in *evaluator*.

In Task 2, a trivial dataset is used. ML analysis models usually load files from data repository for analysis. Here, you can use *pandas* to load the data. You can also use pandas next to explore the data both with descriptive statistics and data visualization. Observe the following code and note that we are specifying the names of each column when loading the data. From line 29 to line 35, the codes basically try to identify a sub-class attack from the file and their corresponding class and put the information in *subClass* and *currentAttackClasses*, respectively. Finally, the codes print these two arrays. For the loaded example file *KDDTrain+.txt*, the outputs are:

```
Sub classes of Attacks
['normal', 'neptune', 'warezclient', 'ipsweep', 'portsweep', 'teardrop', 'nmap',
    'satan', 'smurf', 'pod', 'back', 'guess_passwd', 'ftp_write', 'multihop',
    'rootkit', 'buffer_overflow', 'imap', 'warezmaster', 'phf', 'land',
    'loadmodule', 'spy', 'perl']
```

```
Attack Classes
['DoS (A1)', 'R2L (A4)', 'Probe (A2)', 'U2R (A3)']
```

There are 22 sub-class attacks identified and they belong to all for attack classes. Note that *normal* is a label for normal traffic but not an attack.

Task 4 Features Mapping

In ML data processing, we can use training data to create a pattern detection model; then we use testing data to evaluate the performance of the training results. For both supervised and semi-supervised learning approaches, training and testing datasets are all labeled. It is critical to check if the training dataset contains the same number of features and categories as the testing dataset. For a given dataset, several information need to be extracted from it, including:

- The number of features, i.e., the number of columns
- The number of columns are non-number, usually they are strings
- The number of distinct strings in each feature (or column)

Once understanding these features, we need to convert strings into unique values for each feature, and usually strings are converted into integers. This procedure is open called dataset standardization. To achieve this goals, the following Python codes implement the information extraction from a labeled dataset, and transfer the original dataset into a standardized dataset. Now, we look into this file *categoryMapper.py*

```
1  import numpy as np
2  import pandas as pd
3  import random
4  import os
5
6  # Define variables
7  file_extension='.csv' # .csv or .txt
8  file_folder='NSL-KDD/'
9
10 # load files and content into X
11 fileToStandardize = input("Please enter the file to be standardized without the extension\n")
12 dataset = pd.read_csv(file_folder + fileToStandardize + file_extension, header=None,
       encoding="ISO-8859-1")
13 X = dataset.iloc[:, :].values
14
15 # the number of features equals to the number of columns of the loaded file
16 nFeatures = len(X[0])
17 print("Extracting String Columns....")
18
19 stringColumns = []
20 i = random.randint(10, 100) #Just choose a row at random (a value between 10 and 100) to check for
       string columns
21 for j in range(nFeatures):
22 try:
23 floatValue = float(X[i, j])
24 except:
25 stringColumns.append(j)
26 print("String Columns are : " + str(stringColumns)) # print string column indexes starting from 0
27
28 shouldSaveFeatureMappings = input("Do you want to save feature mappings result[y/n]?")
```

```
29  if shouldSaveFeatureMappings == 'y':
30  directory_featureMapping = input("What local directory to store the created feature mappings?\n")
31  if not os.path.exists(directory_featureMapping):
32  os.mkdir(directory_featureMapping)
33  print("************************************************")
34  # the for loop store each string column into a file with name of the column index
35  for j in stringColumns:
36  distinctValues = []
37  featureMap = []
38  print("Distinct values for feature index " + str(j) + " are: ")
39  for i in range(len(X)):
40  if X[i, j] not in distinctValues:
41  distinctValues.append(str(X[i, j]))
42  featureMap.append(str(X[i, j]) + ", " + str(len(distinctValues)-1))
43  #If header exists prints, feature name, else prints feature value as an example
44  print(str(distinctValues))
45  if shouldSaveFeatureMappings == 'y':
46  featureMapFile = directory_featureMapping+"/" + str(j) + file_extension
47  np.savetxt(featureMapFile, np.array(featureMap), delimiter=',', fmt="%s")
48  print(featureMapFile + " has been saved for column index " + str(j))
49  print("************************************************")
50  # the following if converts the original data file into a standardized file by replacing strings
        to their indexing ID
51  shouldContinue = input("Do you want to standardize the given input file?[y/n/]?")
52  if shouldContinue == 'y':
53  #featuresMappingsFolder = input("Please enter the path to the featureMappings folder\n")
54  print("Now, Standardizing.....")
55  for j in stringColumns:
56  featureMapping = (pd.read_csv(directory_featureMapping + "/" + str(j) + file_extension,
        header=None, encoding="ISO-8859-1")).iloc[:, :].values
57  for i in range(len(X)):
58  for k in range(len(featureMapping)):
59  if str.lower(str(X[i, j])) == str.lower(str(featureMapping[k, 0])):
60  X[i, j] = featureMapping[k, 1]
61  print("Creating standardized file.....")
62  standardizedFile = file_folder + fileToStandardize + "_standardized"+file_extension
63  np.savetxt(standardizedFile, np.array(X), delimiter=',', fmt="%s")
64  print("************************************************")
65  print("Standardized file " + standardizedFile + " has been created")
66  else:
67  print("Execution completed")
68  else:
69  print("Quit")
```

The comments in *categoryMapper.py* can mostly explain what they are doing for each method. At the high-level, the codes from line 28 to line 49 will generate individual files to enumerate each string features with their indexes as the file name. For example, if we want to analyze *KDDTest+.csv* dataset, it will generate 4 individual files: *1.csv*, *2.csv*, *3.csv*, and *41.csv*. Each *.csv* file contains the string and assign unique integer. For example, the file *1.csv* has the following content:

```
tcp, 0
icmp, 1
udp, 2
```

It means that the column 1 of the loaded dataset has 3 strings *tcp*, *icmp*, and *udp*, and each is mapped to a unique integer. Thus, in the loaded dataset, there are four columns containing strings and they are in columns 1, 2, 3, and 41.

The rest codes in the Python program is to generate a standardized dataset, e.g., *KDDTest+_standardized.csv* is generated in the example, It will replace strings by their corresponding assigned integer values.

Task 5 Dataset Manipulation

In this task, we present how to manipulate training and testing datasets to rearrange labeled data to create different training and testing datasets. We still use the NSL-KDD dataset that contains both labeled normal and attack data. As presented in CS-ML-00101 lab, the NSL-KDD data set exists 4 different classes of attacks: Denial of Service (DoS), Probe, User to Root(U2R), and Remote to Local (R2L), and altogether there are 39 sub-classes of attacks, which are summarized in Table CS-ML-00200.2. Both training and testing datasets contains these four classes of attacks. The following Python program *DataExtractor.py* allows you to customize labeled datasets by extracting specific class of attacks.

```
import numpy as np
import pandas as pd

# Define variables
# Data file Path
DatasetPath='NSL-KDD/'
# Data file name
input_train = "KDDTrain+.txt"
input_test = "KDDTest+.txt"
file_extension = '.csv' # .csv or .txt
num_attack_class = 4 # total number of attack classes

#All attacks in NSL-KDD classed based on their attack classes: DoS, Prob, U2R, and R2L
attacks_subClass = [['apache2', 'back', 'land', 'neptune', 'mailbomb', 'pod', 'processtable',
    'smurf', 'teardrop', 'udpstorm', 'worm'],
['ipsweep', 'mscan', 'portsweep', 'saint', 'satan'],
['buffer.overflow', 'loadmodule', 'perl', 'ps', 'rootkit', 'sqlattack', 'xterm'],
['ftp.write', 'guess.passwd', 'httptunnel', 'imap', 'multihop', 'named', 'phf', 'sendmail',
    'snmpgetattack', 'spy', 'snmmpguess', 'warezclient', 'warezserver', 'xlock', 'xsnoop']
]

# selected training attack classes
training_attack_class_list = []
# selected testing attack classes
testing_attack_class_list = []
attack_class_1 = list(map(int, input("(Training Dataset) Please enter the attack class(es) that
    you want from the below list:\n(Note that you can choose one or multiple classes, e.g., 1 3,
    and input 0 means nothing is chosen) \na1 -> DoS (Enter 1 for this selection)\na2 -> Probe
    (Enter 2 for this selection)\na3 -> U2R (Enter 3 for this selection)\na4 -> R2L (Enter 4 for
    this selection)\n\n").split()))
attack_class_2 = list(map(int, input("(Testing Dataset) Please enter the attack class(es) that you
    want from the below list:\n(Note that you can choose one or multiple classes, e.g., 1 3, and
    input 0 means nothing is chosen) \na1 -> DoS (Enter 1 for this selection)\na2 -> Probe (Enter
    2 for this selection)\na3 -> U2R (Enter 3 for this selection)\na4 -> R2L (Enter 4 for this
    selection)\n\n").split()))
training_attack_class_list.append(attack_class_1)
testing_attack_class_list.append(attack_class_2)

print("Loading", input_train, "and", input_test, "files from the current folder where this script
    resides.....\n")
```

```
dataset_train = pd.read_csv(DatasetPath + input_train, header=None, encoding="ISO-8859-1")
dataset_test = pd.read_csv(DatasetPath + input_test, header=None, encoding="ISO-8859-1")
print("Loading Completed !\n")

X_train = dataset_train.iloc[:, :].values
X_test = dataset_test.iloc[:, :].values

print("Creating training set.....\n")
setA_train = []
# the following for loop choose selected attack classes and normal labeled data and put them into the setA_train.
if training_attack_class_list[0][0] != 0 and len(training_attack_class_list[0]) != num_attack_class:
for i in range(len(X_train)):
# exp., X_train[i, -2] is the label of attack subclass, and attacks_subClass[training_attack_class_list[0][j]-1] identify the selected attack class
for j in range(len(training_attack_class_list[0])):
if str.lower(str(X_train[i, -2])) in attacks_subClass[training_attack_class_list[0][j]-1] or str.lower(str(X_train[i])) == 'normal':
setA_train.append(X_train[i])
trainingFileName="Training"
for i in range(len(training_attack_class_list[0])):
trainingFileName = trainingFileName + "-a" + str(training_attack_class_list[0][i])
trainingFileName = trainingFileName + file_extension
np.savetxt(trainingFileName, setA_train, delimiter=',', fmt="%s" )
print("Files " + trainingFileName + " have been created in the same folder this script resides\n")
elif len(training_attack_class_list[0]) == num_attack_class:
print("No changes is needed for training dataset!\n")
else:
print("No attack classes are chosen, thus no new training file is created!\n")

print("Creating testing set.....\n")
setA_test = []
# the following for loop choose selected attack classes and normal labeled data and put them into the setA_train.
if testing_attack_class_list[0][0] != 0 and len(testing_attack_class_list[0]) != num_attack_class:
for i in range(len(X_test)):
# exp., X_train[i, -2] is the label of attack subclass, and attacks_subClass[training_attack_class_list[0][j]-1] identify the selected attack class
for j in range(len(testing_attack_class_list[0])):
if str.lower(str(X_test[i, -2])) in attacks_subClass[testing_attack_class_list[0][j]-1] or str.lower(str(X_test[i])) == 'normal':
setA_test.append(X_test[i])
testingFileName="Testing"
for i in range(len(testing_attack_class_list[0])):
testingFileName = testingFileName + "-a" + str(testing_attack_class_list[0][i])
testingFileName = testingFileName + file_extension
np.savetxt(testingFileName, setA_test, delimiter=',', fmt="%s" )
print("Files " + testingFileName + " have been created in the same folder this script resides\n")
elif len(testing_attack_class_list[0]) == num_attack_class:
print("No changes is needed for testing dataset!\n")
else:
print("No attack classes are chosen, thus no new training file is created!\n")
```

The Python program presented above is easy to understand with given comments and printed explanations. This program basically allows the data user to choose one or multiple attack classes in the training and testing datasets. In this way, you can *customize* attack scenarios as training and testing datasets to evaluate processed ML algorithms. The outputs of the program is to create new training and testing dataset. For

example, *Training-a1-a3.csv* represents the training dataset contains labeled class 1 and class 3 attacks data, and *Testing-a1-a2-a3.csv* represents the testing dataset contains labeled class 1, class 2, and class 3 attacks data. Entering 0 or all attack classes (i.e., 1 2 3 4) for training and testing datasets will not produce new files.

Task 6 Create a Data Pre-processing library

To simplify the data pre-processing, we can create a local library file *data-preprocessor.py*, which is presented in the following codes.

```
import numpy as np
import pandas as pd
from keras.utils import np_utils
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

def get_processed_data(datasetFile, categoryMappingsPath, classType='binary'):
inputFile = pd.read_csv(datasetFile, header=None)
X = inputFile.iloc[:, 0:-2].values
label_column = inputFile.iloc[:, -2].values

category_1 = np.array(pd.read_csv(categoryMappingsPath + "1.csv", header=None).iloc[:, 0].values)
category_2 = np.array(pd.read_csv(categoryMappingsPath + "2.csv", header=None).iloc[:, 0].values)
category_3 = np.array(pd.read_csv(categoryMappingsPath + "3.csv", header=None).iloc[:, 0].values)

ct = ColumnTransformer(
[('X_one_hot_encoder', OneHotEncoder(categories=[category_1, category_2, category_3],
    handle_unknown='ignore'), [1,2,3])], # The column numbers to be transformed ([1, 2, 3]
    represents three columns to be transferred)
remainder='passthrough'# Leave the rest of the columns untouched
)
X = np.array(ct.fit_transform(X), dtype=np.float)

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(np.array(X)) # Scaling to the range [0,1]

if classType == 'binary':
y = []
for i in range(len(label_column)):
if label_column[i] == 'normal' or str(label_column[i]) == '0':
y.append(0)
else:
y.append(1)
# Convert ist to array
y = np.array(y)
else:
#Converting to integers from the mappings file
label_map = pd.read_csv(categoryMappingsPath + "41.csv", header=None)
label_category = label_map.iloc[:, 0].values
label_value = label_map.iloc[:, 1].values

y = []
for i in range(len(label_column)):
y.append(label_value[label_category.tolist().index(label_column[i])])
```

```
44  # Encoding the Dependent Variable
45  y = np_utils.to_categorical(y)
46
47  return X, y
```

In this Python program, it defined a *get_processed_data(datasetFile, categoryMappingsPath, classType='binary')* method. It passes *datasetFile* as the input dataset. The parameter *categoryMappingsPath* identifies location of category mapping files (refer to Task 4 on how to create a category/feature mapping). The default classification type is binary, which means yes or no decision for ML algorithm. It can pass multi-class if needed. Note that this data pre-procesing Python program is customized for the NSL-KDD datasets. For other datasets, you need to change the mapping files and modify the related codes accordingly. The main functions of this data preprocessing program include the following functions:

- It takes a given training or testing dataset and checks if its labeled string data (i.e., categories) matches the pre-known knowledge of the entire dataset (includes both training and testing datasets). Note that if the training dataset contains less number of categories than the testing dataset, then the mismatch will cause inaccurate testing for model validation.
- It scales the dataset values within a specified range.
- It use *OneHotEncoder* approach to expand the string-based feature to multiple columns to improve the accuracy of the running ML algorithm.

In order to use the presented data preoprocessing method, here we present a ample code to call the method:

```
1  # Import custom modules
2  import data_preprocessor as dp
3
4  #Build preprocessed training data
5  X_train, y_train = dp.get_processed_data(DatasetPath + TrainingData, categoryMappingsPath,
       classType='multiclass')
6  #Build preprocessed testing data
7  X_test, y_test = dp.get_processed_data(DatasetPath + TestingData, categoryMappingsPath,
       classType='multiclass')
8
9  ... other codes here...
```

Notes:
Files name *__init__.py* is used to mark directories on disk as Python package directories. If you have the files *data_preprocessor.py* in your local directory where your running Python program locates at, then you need to create a local dummy file *__init__.py* in the local directory, and you can import it directly:

```
import data_preprocessor
```

If it locals in a directory such as

```
my-current-dir/sub-dir_name/__init__.py
my-current-dir/sub-dir_name/data_preprocessor.py
```

Then you need to call it as:

```
import sub-dir_name.data_preprocessor
```

or

```
from spam import data_preprocessor
```

The *__init__.py* file is required to make Python treat the directories as containing packages; this is done to

prevent directories with a common name, such as *String*, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, *__init__.py* can just be an empty file, but it can also execute initialization code for the package or set the *__all__.py* variable. If you remove the *__init__.py* file, Python will no longer look for sub modules inside that directory. Given the example above, the contents of the *init* module can be accessed as:

```
import sub-dir_name
```

Related Information and Resource

```
Data preprocessing:
https://www.shanelynn.ie/select-pandas-dataframe-rows-and-columns-using-iloc-loc
-and-ix/
https://medium.com/@contactsunny/label-encoder-vs-one-hot-encoder-in-machine
-learning-3fc273365621
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train
_test_split.html
https://scikit-learn.org/stable/modules/preprocessing.html

Build ANN:
https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/
https://machinelearningmastery.com/rectified-linear-activation-function-for-deep
-learning-neural-networks/
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep
-learning-neural-networks/
```

"nobreak

Lab Assessment

In this lab, students can perform a self-assessment by running the provided Python programs and test them on provide NSL-KDD datasets.