Traffic classification programming example

Tran Thien Thi

June 2019

Contents

1 Example 1: traffic classification **1**
1.1 Pre-processing 2
1.2 Model training 6
1.3 Evaluation 7
1.4 All code for this example 1 8

2 Example 2: further evaluations **10**
2.1 All codes for this example 2 12

3 Example 3: R version of both examples **12**
3.1 R version of Example 1 13
3.2 R version of Example 2 17

Introduction

In this document, we will go through case where we handle the dataset with Python scikit-learn library. The dataset contains Youtube and web browsing data in flows. The process is supervised learning, and the aim is traffic classification. The coding contains three steps: Pre-processing, training model, and evaluation.

Generally about scikit library, it performs well handling both dataframes or arrays, so that's why sometimes switching to the arrays was not necessary in this example. However, other programming languages or libraries might not be able to do same. Therefore, usually it is good idea to convert dataframes to arrays in order to have less computation times or avoid possible errors.

```
1 Template for codes and outputs
```

1 Example 1: traffic classification

In this first example, we go through how the Internet traffic of captured traffic can be classified using Python's Scikit-library. The traffic was captured for

youtube traffic (capturing packets while video was running 30min) and for regular web browsing (30min).

The pre-processing phase contains following procedures: feature extraction, sampling, categorization, imputing missing values, standardization, and feature selection. After pre-processing, there will be model training and basic evaluation with score metrics and confusion matrix.

1.1 Pre-processing

Let's assume that we have only different .pcap files for two different applications available at the beginning.

First we need to extract some useful features from the .pcap file, such as average packet sizes, etc. Using NetMate, we are able to easily extract 44 features (the description of the features can be found from its doumentation page). You could also use CoralReef for extracting 11 features but some of them can be useless unless you build more features manually based on them.

NetMate can be cloned from github repo, installed on school computers without admin rights (make sure the installation folder is at somewhere your home directory, and update the path variable "PATH=$PATH:~/bin"). After installation, you need to have the .pcap file and the rule file (XML) at the same directory. You can use the example rule file but make sure to change the Netmate output location in that rule file so you can find it later.

Then you should be able to run the following command:

```
$netmate -r  [XML_FILE] -f [PCAP_FILE]
```

This will then create new csv-file-like with 44 different columns. Keep in mind: if you utilize the command again, it will append the new output instead of overwriting, so good idea would be to rename the output file immediately once you have used the command. There are no headers so you can add them manually, see the Netmate documentation for each headers.

Currently, the dataset is unlabeled you need to label each .pcap file individually. For example, by adding additional column containing the specific label for youtube traffic or web browsing traffic. After this, you could combine all dataset together for one .csv file. I recommend programming a code that adds headers, labels, and combine datasets to simplify this process until this. Here before pre-processing, the .csv file contains the output of Netmate, and also the additional column "label" for the classification target value. Youtube traffic was labeled as "youtube" and web browsing was labeled as "webBrow".

Now we should have labeled but raw dataset available. Let's import the labeled and combined data and see how it initially looks like. (We also save the headers so that we can use them later)

```
directory = '~/Documents/dataset.csv'
raw_df = pd.read_csv(directory)
headers = raw_df.columns.values
print(raw_df.head(n=3))
print("Shape of whole dataset: " , raw_df.shape)
```

```
             srcip  srcport           dstip  ...  total_fhlen  total_bhlen    label
0  130.233.145.193    51738  109.105.98.204  ...          476          476  youtube
1  130.233.145.193    38766  130.233.227.12  ...          320          216  youtube
2  130.233.145.193    56320   130.233.251.6  ...          268          204  youtube

[3 rows x 45 columns]
Shape of whole dataset:  (705, 45)
```

Let's check initial stats regarding amount of flows that belong to Youtube or web browsing.

```
print(raw_df.groupby(['label']).size())
```

```
label
webBrow    234
youtube    471
```

We can see that Youtube has much more flows than web browsing, so let's balance the dataset. Since this particular dataset is small (about 700 flows in total), we will perform oversampling for web browsing. However, it is good to keep in mind that in general, oversampling might lead to overfitting.

In the code below, we create two new temporary dataframes that contains only youtube or web browsing flows. Since youtube has more flows, we will take that amount and sample it from webBrow category. Then we combine both together and display its statistics, where we can see the oversampling went correctly.

```
df_youtube = raw_df[raw_df['label'] == 'youtube']
df_web = raw_df[raw_df['label'] == 'webBrow']

amount = (raw_df['label']=='youtube').sum()
df_web = df_web.sample(amount, replace=True)

raw_df = pd.concat([df_youtube, df_web], axis=0)

print(raw_df.groupby(['label']).size())
```

```
label
webBrow    471
youtube    471
Shape of whole dataset after oversampling: (942, 45)
```

Our dataset contains only three particular columns which are not numerical, let's encode them. However, keep in mind that we already know what the dataset look like, which is the reason for such hard-coding. Normally we should implement the code to detect non-numerical column contents and encode only them. There are several other more advanced encoding methods to avoid such situations, for example: OneHotEncoder(). But for simple demonstration regarding Internet traffic classification, let's just use LabelEncoder().

(In case at some point you need to decode the numbers, you can use inverse_transform() function to look at the original values)

```
le = LabelEncoder()
raw_df['srcip'] = le.fit_transform(raw_df['srcip'])
raw_df['dstip'] = le.fit_transform(raw_df['dstip'])
raw_df['label'] = le.fit_transform(raw_df['label'])
print(raw_df.head(n=3))
```

```

#print(le.inverse_transform(raw_df['label']))
```

```
   srcip  srcport  dstip  dstport  ...  burg_cnt  total_fhlen  total_bhlen  label
0     35    51738      7      443  ...         0          476          476      1
1     35    38766     17      111  ...         0          320          216      1
2     35    56320     21       88  ...         0          268          204      1

[3 rows x 45 columns]
```

As we can see from the output, "youtube" was coded as 1(and "webBrow" was coded as 0).

Next the dataframes will be converted to arrays to handle the code easier. Shuffling is to ensure the diversity of the sets since the dataset was originally ordered since youtube flows were at the beginning of the dataset and web browsing flows at the end.

Then we are going to split the data to training set and test set before data transformations. The reason for this is to avoid 'data leakage' by separating the test set completely before adjusting the transformations. The split ratio between training set and test set usually depends on the application, let's now keep it 9:1 because the whole dataset is quite small already

In addition, since this is classification problem, it is often good idea to perform transformations only for the feature set (and not target value). That's why we also separate the features (X) and targets (Y) from each others. We also reshape the target column because in Python, array shape of ABC(x,) and ABC(x,1) is different. The former is vector (1D array) and the latter is matrix (2D array with one 'column'). This matrix form is preferred in some of Python library functions.

```
raw_df = raw_df.to_numpy()
raw_df = shuffle(raw_df)
print(raw_df)
print("Shape of whole dataset: ", raw_df.shape)

train_size = int(len(raw_df) * 0.90)
train = raw_df[:train_size, :]
test = raw_df[train_size:, :]
print("Shape of training dataset: ", train.shape)
print("Shape of test dataset: ", test.shape)

X_train = train[:, :-1]
Y_train = train[:, -1]
Y_train = Y_train.reshape(Y_train.shape[0], 1)

X_test = test[:, :-1]
Y_test = test[:, -1]
Y_test = Y_test.reshape(Y_test.shape[0], 1)

print("Shape of training dataset (targets): ", Y_train.shape)
print("Shape of training dataset (features): ", X_train.shape)
print(X_train)
```

```
[[    8    68    73 ...   128     0     1]
 [   35 54970    16 ...   372   372     0]
 [   29   138    14 ...   148     0     1]
```

```
 ...
 ]]
Shape of whole dataset:  (942, 45)
Shape of training dataset:  (847, 45)
Shape of test dataset:  (95, 45)
Shape of training dataset (targets):  (847, 1)
Shape of training dataset (features):  (847, 44)
[[    8    68    73 ...     0   128     0]
 [   35 54970    16 ...     0   372   372]
 [   29   138    14 ...     0   148     0]
 ...
 ]]
```

Next we will fill empty data values (NaN). Even though the imported dataset might not have any NaN values, it is good idea to perform "data-imputing" just in case.

This is done by taking means of their respective column. There are also other ways to do this, such as dropping such rows or taking median or using particular learning algorithm that takes into account all features, but let's just keep it simple example. In scikit-library, SimpleImputer() is used for univariate values, i.e., target value depends on a single column.

We fit the imputer scaler with only training data, and then apply the imputer scaler to both training set and test set to avoid any data leakages.

```
si = SimpleImputer(strategy = 'mean')
si = si.fit(X_train)
X_train = si.transform(X_train)
X_test = si.transform(X_test)
```

Next we will perform standardization for the data, i.e., making it more Gaussian like. Again, same strategy is applied here: fitting the scaler with only training dataset, and then apply it to both sets.

```
standard_scaler = StandardScaler()
standard_scaler = standard_scaler.fit(X_train)
X_train = standard_scaler.transform(X_train)
X_test = standard_scaler.transform(X_test)

print(X_train)
```

```
[[-2.13142727 -1.54998456  1.7602874  ...  0.         -0.06690683  -0.07138781]
 [-0.08044645  0.84177445 -0.58296051 ...  0.         -0.0612467   -0.06234995]
 [-0.53621996 -1.54693507 -0.66517974 ...  0.         -0.06644288  -0.07138781]
 ...
 ]]
```

As we can see above, every values were normalized according to their columns.

Next we will perform feature selection, i.e., choosing only few best features from the existing ones that will be used for training. Motivation for this is to reduce the amount of processed data without losing much accuracy.

Scikit-library have many algorithms for estimating best features from whole dataset. Popular algorithms used are, for example, chi2. However, keep in mind that chi2 only work with non-negative values.

In the code snippet below, we choose ten most relevant columns, i.e., highly correlated to the target column. Again, same strategy here: adjust the scaler only using training data, and then apply it to both sets.

```
  number_of_desired = 10
  selector = SelectKBest(f_classif, k=number_of_desired)
  selector = selector.fit(X_train, Y_train)
  X_train = selector.transform(X_train)
  X_test = selector.transform(X_test)

  print(X_train)
  print(X_train.shape)
```

```
[[ 1.7602874   1.15470054 -0.38110055 ... -0.40034708 -0.3117161  -0.42675503]
 [-0.58296051 -0.8660254  -0.32595765 ... -0.38645449 -0.3117161  -0.17678165]
 [-0.66517974  1.15470054 -0.38110055 ... -0.40034708 -0.3117161  -0.42675503]
 ...
 ]]
(847, 10)
```

We can see that there are now 10 most relevant features (instead of whole 44 features).

Now that we have all pre-processing done, we could save the resulted dataset in order to use it again later for other cases. Having such clean dataset would not require any further pre-processing then.

So, let's combine all training sets and test sets back together as new clean dataframe. The new clean dataframe doesn't have headers, so we also add the selected column names for it.

```
  train = np.concatenate((X_train, Y_train), axis=1)
  test = np.concatenate((X_test, Y_test), axis=1)
  clean_df = np.concatenate((train, test), axis=0)

  clean_df = pd.DataFrame(clean_df)

  cols = selector.get_support(indices=True)
  selected_columns = headers[cols]
  selected_columns = np.append(selected_columns, 'target')
  clean_df.columns = [selected_columns]

  print(clean_df.head(n=3))
  print(clean_df.shape)

  clean_df.to_csv('~/Documents/clean_dataset.csv', encoding='utf-8'
    , index=False)
```

```
      dstip     proto  mean_bpktl  max_bpktl  std_bpktl  mean_biat   max_biat   std_biat
    std_active  fpsh_cnt  target
  1.760287  1.154701  -0.381101 -0.355844 -0.424496 -0.358467 -0.387996 -0.400347
     -0.311716 -0.426755     1.0
 -0.582961 -0.866025  -0.325958 -0.330491 -0.394012 -0.340772 -0.378973 -0.386454
     -0.311716 -0.176782     0.0
 -0.665180  1.154701  -0.381101 -0.355844 -0.424496 -0.358467 -0.387996 -0.400347
     -0.311716 -0.426755     1.0
(942, 11)
```

1.2 Model training

Then the actual training to create the model. We will use KNN as ML technique to train the model. There are also many other options available, feel free to choose other options among scikit-library. However, when using KNN, make

sure that the neighbor parameter is odd-numbered due to the nature of the algorithm (in order to avoid tie results).

Also remember that the training is only performed for the training set, we will keep test set out of this.

```
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, Y_train)
```

1.3 Evaluation

Next we will perform model evaluation if it seems good enough. We will let the model to predict the test set for which Internet traffic type it belongs to. Then we see how correct its predictions were compared to the real test set values. The following code snippet gives accuracy score, confusion matrix, and classification report.

```
predicted_Y = model.predict(X_test)

score = accuracy_score(Y_test, predicted_Y)
print("KNN accurary score is : " + str(score))

conf_mat = confusion_matrix(y_true=Y_test, y_pred=predicted_Y)
print('Confusion matrix:\n', conf_mat)

class_report = classification_report(y_true=Y_test, y_pred=
  predicted_Y)
print('Classification report:\n', class_report)
```

```
KNN accurary score is : 0.7157894736842105
Confusion matrix:
 [[36 20]
 [ 7 32]]
Classification report:
              precision    recall  f1-score   support

           0       0.84      0.64      0.73        56
           1       0.62      0.82      0.70        39

   micro avg       0.72      0.72      0.72        95
   macro avg       0.73      0.73      0.72        95
weighted avg       0.75      0.72      0.72        95
```

By default, the scikit's confusion_matrix() method sorts the values according to their labels in increasing order (from left to right and from up to down: 0 to infinite).

Remember that web browsing traffic was labeled as 0 and youtube as 1. From the confusion matrix output can be seen, that the created model thought that there were 36+7=43 instances of web traffic and 20+32=52 instances of Youtube traffic in this test set. However, in reality there was 36+20=56 instances of web traffic and 7+32=39 instances of Youtube traffic in this test set.

The scikit's classification_report() method is quite useful for seeing scores for accuracy, precisions, and recalls.

Finally, save the model for later use if needed. For example, using it for another new unseen data (which has been pre-processed in similar fashion).

```
model_file = input("Name of your model? Use .sav format.\n")
joblib.dump(model, model_file)
```

When you later load the particular for new test set, you can do the following. Also, make sure that the Python version is similar as when first saved your model in order to ensure the compability of the model.

```
loaded_model = joblib.load('knn.sav')
predicted_Y = loaded_model.predict(X_test)
...
```

1.4 All code for this example 1

```
# Importing needed libraries
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.externals import joblib

def main():

  # Making the results reproducible for testing purposes
  np.random.seed(1)

  # Importing the labeled raw dataset
  directory = '~/Documents/dataset.csv'
  raw_df = pd.read_csv(directory)
  headers = raw_df.columns.values


  # Sampling (oversampling)
  df_youtube = raw_df[raw_df['label'] == 'youtube']
  df_web = raw_df[raw_df['label'] == 'webBrow']
  amount = (raw_df['label']=='youtube').sum()
  df_web = df_web.sample(amount, replace=True)
  raw_df = pd.concat([df_youtube, df_web], axis=0) # merge
    vertically


  # Categorization
  le = LabelEncoder()
  raw_df['srcip'] = le.fit_transform(raw_df['srcip'])
  raw_df['dstip'] = le.fit_transform(raw_df['dstip'])
  raw_df['label'] = le.fit_transform(raw_df['label'])

  # Dividing to training and test set
```

```
raw_df = raw_df.to_numpy()
raw_df = shuffle(raw_df)
train_size = int(len(raw_df) * 0.90)
train = raw_df[:train_size, :]
test = raw_df[train_size:, :]

# Dividing both sets to features and target value
X_train = train[:, :-1]
Y_train = train[:, -1]
Y_train = Y_train.reshape(Y_train.shape[0], 1) # change vector into matrix
X_test = test[:, :-1]
Y_test = test[:, -1]
Y_test = Y_test.reshape(Y_test.shape[0], 1)

# Imputing missing values (for features only)
si = SimpleImputer(strategy = 'mean')
si = si.fit(X_train)
X_train = si.transform(X_train)
X_test = si.transform(X_test)

# Standardization (for features only)
standard_scaler = StandardScaler()
standard_scaler = standard_scaler.fit(X_train)
X_train = standard_scaler.transform(X_train)
X_test = standard_scaler.transform(X_test)

# Feature selection (for features only)
number_of_desired = 10
selector = SelectKBest(f_classif, k=number_of_desired)
selector = selector.fit(X_train, Y_train)
X_train = selector.transform(X_train)
X_test = selector.transform(X_test)

# Saving the pre-processed dataset
train = np.concatenate((X_train, Y_train), axis=1) # merge horizontally
test = np.concatenate((X_test, Y_test), axis=1)
clean_df = np.concatenate((train, test), axis=0) # merge vertically
clean_df = pd.DataFrame(clean_df) # change array into dataframe
cols = selector.get_support(indices=True)
selected_columns = headers[cols]
selected_columns = np.append(selected_columns, 'target')
clean_df.columns = [selected_columns] # add column names for dataframe
clean_df.to_csv('~/Documents/clean_dataset.csv', encoding='utf-8', index=False)


# Training the model
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, Y_train)

# Evaluation of the model
predicted_Y = model.predict(X_test)
score = accuracy_score(Y_test, predicted_Y)
```

```
  print(score)
  conf_mat = confusion_matrix(y_true=Y_test, y_pred=predicted_Y)
  print(conf_mat)
  class_report = classification_report(y_true=Y_test, y_pred=
    predicted_Y)
  print(class_report)

  # Saving the model
  model_file = input("Name of your model? Use .sav format.\n")
  joblib.dump(model, model_file)


# Running the main function
if __name__ == "__main__":
  main()
```

2 Example 2: further evaluations

In this example, nice evaluation method K-fold CV is demonstrated.

Let's now continue further from Example 1 case. In addition to KNN algorithm, now you would want also to test other algorithms too: linear support vector classification and Naive Bayes. The reason would be to see which ML algorithm performs best and would be best for your training your model.

However, creating just one model for each case and taking the single result from it might lead to different results after each run since the model training is stochastic process. Better idea would be to train multiple models for each algorithms, and then take average score from them. This result would be more stable and comparable. K-fold CV would be suitable for this kind of evaluation.

First, we will load the clean dataset that we did earlier in Example 1 so that we do not need to write the pre-processing code part again. We also separate the dataset from features column and target column.

```
  clean_df = pd.read_csv('~/Documents/clean_dataset.csv', header=0)
  clean_df = clean_df.to_numpy()
  X = clean_df[:, :-1]
  Y = clean_df[:, -1]
  print("Shape of whole dataset: ", clean_df.shape)
  print("Shape of whole dataset (features): ", X.shape)
  print("Shape of whole dataset (target): ", Y.shape)
```

```
Shape of whole dataset:  (942, 11)
Shape of whole dataset (features):  (942, 10)
Shape of whole dataset (target):  (942, )
```

Then, we configure the K-fold CV to split the dataset to 10 instances. Therefore, we will train 10 different models for each algorithms, and finally take the average score from them. In this first case, we perform K-fold CV for KNN algorithm. It is recommended to read the theory section regarding K-fold CV in order to understand the code below properly.

```
  kf = KFold(n_splits = 10)
  scores = list()
```

```
  for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    Y_train, Y_test = Y[train_index], Y[test_index]
    model = KNeighborsClassifier(n_neighbors=5)
    model.fit(X_train, Y_train)
    predicted_Y = model.predict(X_test)
    scores.append(accuracy_score(Y_test, predicted_Y))

  print("All KNN scores : ", scores)
  print("Average KNN score is : ", sum(scores)/len(scores))
```

```
All KNN scores :  [0.7263157894736842, 0.7368421052631579, 0.8297872340425532,
    0.7127659574468085, 0.6702127659574468, 0.648936170212766, 0.7978723404255319,
    0.776595744680851, 0.776595744680851, 0.7446808510638298]
Average KNN score is :  0.742060470324748
```

The previous code could have been shortened drastically by using function cross_val_score(). We need to tell this algorithm that which training algorithm, the features, the target, and also the cross validation method will be used. We will use now this cross_val_score() for the rest algorithms.

```
  kf = KFold(n_splits = 10)
  model = KNeighborsClassifier(n_neighbors=5)
  k_cross_score = cross_val_score(model, X, Y, cv=kf)
  print("All cross_val_scores : ", k_cross_score)
  print("cross_val_score on average is : ", k_cross_score.mean())
```

```
All cross_val_scores :  [0.72631579 0.73684211 0.82978723 0.71276596 0.67021277
    0.64893617
 0.79787234 0.77659574 0.77659574 0.74468085]
cross_val_score on average is :  0.742060470324748
```

Now we repeat the same procedure for the rest of the algorithms LinearSVC and Naive Bayes, respectively.

```
  model2 = svm.LinearSVC(max_iter=100000)
  k_cross_score = cross_val_score(model2, X, Y, cv=kf)
  print("All cross_val_scores : ", k_cross_score)
  print("cross_val_score on average is : ", k_cross_score.mean())

  model3 = GaussianNB()
  k_cross_score = cross_val_score(model3, X, Y, cv=kf)
  print("All cross_val_scores : ", k_cross_score)
  print("cross_val_score on average is : ", k_cross_score.mean())
```

```
All cross_val_scores :  [0.74736842 0.71578947 0.76595745 0.72340426 0.74468085
    0.69148936
 0.77659574 0.76595745 0.73404255 0.70212766]
cross_val_score on average is :  0.7367413213885778
All cross_val_scores :  [0.69473684 0.70526316 0.76595745 0.62765957 0.74468085
    0.54255319
 0.75531915 0.71276596 0.70212766 0.60638298]
cross_val_score on average is :  0.6857446808510639
```

As from the result can be seen, KNN algorithm seems to give best average scores in case of K-fold CV. Therefore in practice, when building final model, we would choose KNN algorithm for training WHOLE dataset (no need to split to training or test anymore).

K-fold CV can be used other things than choosing between different models. It could be also used for tuning one particular model's parameters, such as number of neighbors in case of KNN algorithm.

2.1 All codes for this example 2

```
1
2  # Importing the needed libraries
3  import numpy as np
4  import pandas as pd
5  from sklearn.metrics import accuracy_score
6  from sklearn.neighbors import KNeighborsClassifier
7  from sklearn import svm
8  from sklearn.naive_bayes import GaussianNB
9  from sklearn.model_selection import KFold
10 from sklearn.model_selection import cross_val_score
11
12 def main():
13
14   # Importing the clean dataset (already pre-processed)
15    clean_df = pd.read_csv('~/Documents/clean_dataset.csv', header=0)
16    clean_df = clean_df.to_numpy()
17   X = clean_df[:, :-1]
18   Y = clean_df[:, -1]
19
20   # K-fold CV with 10 different sets
21    kf = KFold(n_splits = 10)
22
23   # K-fold CV for KNN algorithm
24    model = KNeighborsClassifier(n_neighbors=5)
25    k_cross_score = cross_val_score(model, X, Y, cv=kf)
26    print("cross_val_score on average is : ", k_cross_score.mean())
27
28   # K-fold CV for LinearSVC algorithm
29    model2 = svm.LinearSVC(max_iter=100000)
30    k_cross_score = cross_val_score(model2, X, Y, cv=kf)
31    print("cross_val_score on average is : ", k_cross_score.mean())
32
33   # K-fold CV for Naive Bayes algorithm
34    model3 = GaussianNB()
35    k_cross_score = cross_val_score(model3, X, Y, cv=kf)
36    print("cross_val_score on average is : ", k_cross_score.mean())
37
38
39 if __name__ == "__main__":
40   main()
```

3 Example 3: R version of both examples

There are several major differences between R and Python language, here are the most relevant ones in general:

- Regarding indexing in R, it is one-based (Python was zero-based). Indexing overall can be different. For example, in case of 2D array data: In R, choosing all rows only from 2nd column would be like [, 2]. In Python it would be [: , 1].

- Regarding variable assignment in R, usually '<-' is safer to use than '=' to avoid scope confusions and ensuring compabilities of most R packages.
- Regarding indentation in R, they don't matter at all (but it is good practice to keep code readable)
- Many R libraries can handle dataframes (since dataframes are big part of R) but usually converting them to arrays will make computation faster.
- In R, lists 'list()' or vectors 'c()' are often used to represent 1D arrays. Vector consists of elements of same types whereas the list can consist different type of elements. For higher dimension arrays, matrices are used.
- Regarding machine learning libraries, R has a lot of them. On the other hand, this may lead to compability issues so check what type of inputs the library functions require. For example, some ML functions only take arrays as input, some only takes dataframes as input, some can handle both. Feel free to use as many different packages as you want to get the desired results, but check their documentation and examples if you encounter errors.

3.1 R version of Example 1

In this subsection, we will briefly go through similar kind of process as in Example 1 but this time with R programming language. Regarding the code implementation, the major differences or interesting notes are:

- 'caret' package is used as classification here. 'Caret' is one popular ML package for R language but it is basically bunch of different ML packages packed together, so carefully handle the compability of the functions.
- Instead of oversampling of web browsing traffic, we will instead take same amount of random samples as there are youtube flows.
- When dealing with categorical values (srcip and dstip), we convert these IP-addresses into integer using a function 'ip_to_numeric' from 'iptools' package
- The function 'preProcess' from 'caret' is very versatile, it can be used for many preprocessing steps such as dealing with missing values, standardization, normalization, ..., etc. In addition, this function can perform the tuning of hyperparameters at the same time and choose the most optimal ones automatically.
- When dealing with missing values, the possible NaN values are replaced with 'knnImpute' which by default takes the average value of 5 closest neighbors in Euclideaan space. This is more advanced simply taking mean of the column. When we actually use 'knnImpute', it also standardizes the dataset beforehand, so there actually happens method=c('center', 'scale',

'knnImpute'). Therefore, we don't need to standardize the dataset this time.

- The feature selection is done using Recursive Feature Elimination (RFE) which takes all features and removes them one-by-one until there are desired amount of features. This 'rfe' function from 'caret' package could perform many things automatically, such as repeating the process many times and finding most optimal amount of features. However, we will keep it simple with one repeat and no cross-validation. And also force it to choose desired amount of features by indexing even though it may prefer more with its 'predictors' result.

- When saving clean dataset for later use, we use completely new variable (Y_train_new) instead of reusing old variable (Y_train) in order to avoid compability issues. (Feature selection and training process seem to work if the target column is as vector and not matrix).

- Similarly in training process, a lot of things happen automatically and cross-validation is also implemented even though we do not necessarily care about it this time.

- Regarding confusion matrix, it is read vice-versa in R. The predictions are from the left side and the real test values are upper side. (In Python predictions are upper side and the real test values are left side.)

```
# Importing the needed libraries
library(caret)
library(ade4)
library(iptools)

main <- function()
{
  # Importing the labeled dataset
  directory <- '~/Documents/kesaduuni_comnet/capturing_
    packets/combined010203_raw.csv'
  raw_df <- read.csv(directory, header=TRUE)

  # Random sampling on web browsing traffic so that same
    number as youtube traffic
  df_youtube <- raw_df[raw_df$label == 'youtube', ]
  df_web <- raw_df[raw_df$label == 'webBrow', ]
  amount <- sum(raw_df$label == 'youtube')
  df_web <- df_web[sample(nrow(df_web), size=amount, replace
    =TRUE), ]
  raw_df <- rbind(df_youtube, df_web)

  # Categorization (convert the ip-addresses to the integers
    )
  raw_df$srcip <- ip_to_numeric(as.character(raw_df$srcip))
```

```
raw_df$dstip <- ip_to_numeric(as.character(raw_df$dstip))

# Shuffling the dataset row instances before splitting
raw_df <- raw_df[sample(nrow(raw_df), size=nrow(raw_df), replace=FALSE), ]

# Splitting to training and test set
split_index <- createDataPartition(raw_df$label, p=0.90, list=FALSE)
train <- raw_df[split_index, ]
test <- raw_df[-split_index, ]

# Splitting both sets to features and target value
X_train <- train[, 1:44]          # matrix
Y_train <- train[, 45]            # vector

X_test <- test[, 1:44]
Y_test <- test[, 45]

# Standardizing values, and imputing missing values (for features only).
imputer <- preProcess(X_train, method=c('knnImpute'))
X_train <- predict(imputer, X_train)
X_test <- predict(imputer, X_test)

# Feature selection (for features only, using recursive elimination)
number_of_desired <- 10
config <- rfeControl(functions=rfFuncs, number=1, p=1.0)
feature_selected <- rfe(X_train, Y_train, size=number_of_desired, rfeControl=config)
X_train <- X_train[, predictors(feature_selected)[1:number_of_desired]] # ensure to choose only 10 best features (number_of_desired=10)
X_test <- X_test[, predictors(feature_selected)[1:number_of_desired]]

# Saving clean dataset
Y_train_new <- as.matrix(Y_train)   # change vector to matrix
Y_test_new <- as.matrix(Y_test)
colnames(Y_train_new) <- "target"       # rename the column
colnames(Y_test_new) <- "target"
train <- cbind(X_train, Y_train_new)    # merge horizontally
test <- cbind(X_test, Y_test_new)
clean_df <- rbind(train, test)      # merge vertically
print(dim(clean_df))
write.csv(clean_df, file = "clean_dataset2.csv", row.names
```

```
    =FALSE)

  # Training the model (KNN algorithm)
  train_config <- trainControl(method="cv", number=10)
  metric <- "Accuracy"
  model <- train(x=X_train, y=Y_train, method="knn", metric=
    metric, trControl=train_config)

  # Making predictions
  predictions <- predict(model, X_test)

  # Evaluating with confusion matrix
  print(confusionMatrix(predictions, Y_test))

  # Saving the model
  saveRDS(model, file="model.rda")
}
# Running the main function
main()
```

```
            srcip srcport          dstip dstport proto  ... label
1 130.233.145.193   51738 109.105.98.204     443     6  ... youtube
2 130.233.145.193   38766 130.233.227.12     111     6  ... youtube
3 130.233.145.193   56320  130.233.251.6      88     6  ... youtube

        freq
webBrow  234
youtube  471

[1] 705  45

        freq
webBrow  471
youtube  471

[1] 942  45


       srcip srcport       dstip dstport proto  ...  label
1 2196345281   51738 1835623116     443     6  ...  youtube
2 2196345281   38766 2196366092     111     6  ...  youtube
3 2196345281   56320 2196372230      88     6  ...  youtube

[1] 848  45

[1] 94 45


           srcip     srcport      dstip  ... total_bhlen
520   0.07170268  0.8807267 -0.1971636  ... -0.06614489
321.2 0.07170232 -1.6056712 -0.1971971  ... -0.07110173
448   0.07170268  0.6253052 -0.1971712  ... -0.06458981

[1] 848  44

         srcport mean_fpktl  max_fpktl  ... total_fvolume
520    0.8807267 -0.2591967 -0.1203912  ... -0.08717880
321.2 -1.6056712  0.2906930 -0.2383398  ... -0.09392991
448    0.6253052 -0.5492484 -0.3158242  ... -0.09052567

[1] 848  10

[1] 942  11

Confusion Matrix and Statistics

          Reference
```

```
Prediction webBrow youtube
  webBrow      38       3
  youtube       9      44

               Accuracy : 0.8723
                 95% CI : (0.7876, 0.9323)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 2.811e-14

                  Kappa : 0.7447

 Mcnemar's Test P-Value : 0.1489

            Sensitivity : 0.8085
            Specificity : 0.9362
         Pos Pred Value : 0.9268
         Neg Pred Value : 0.8302
             Prevalence : 0.5000
         Detection Rate : 0.4043
   Detection Prevalence : 0.4362
      Balanced Accuracy : 0.8723

       'Positive' Class : webBrow
```

As for interpreting the confusion matrix, remember that confusion matrix in R is little different than in Python since here predictions are on the left and the actual real values are up. We can see that the trained model predicted 38+3=41 flows belonging to web browsing and 9+44=53 flows belonging to Youtube traffic. In reality, 38+9=47 belonged to web browsing traffic and 3+44=47 belonged to Youtube traffic.

3.2 R version of Example 2

In this subsection, we will briefly go through similar kind of process as in Example 2 K-fold CV but this time with R programming language. This time we want to see which of three different ML algorithms (KNN, LinearSVM, Naive Bayes) would be most suitable for our model to handle the same cleaned dataset gotten from previous example.

In R's caret package, the K-fold CV evaluation can be implemented during training process.

```r
# Importing the needed libraries
library(caret)
library(kernlab)
library(klaR)

main <- function()
{
  # Importing the clean dataset (already pre-processed)
  clean_df <- read.csv('clean_dataset2.csv')
  X <- clean_df[, 1:(ncol(clean_df)-1)]
  Y <- clean_df[, ncol(clean_df)]
  print(dim(clean_df))
  print(dim(X))
  print(dim(Y)) # vector's shape is displayed as NULL

  # K-fold CV with 10 different sets and one repeat for each
     cases
```

```
  train_config <- trainControl(method="cv", number=10)
  metric <- "Accuracy"

  # KNN model
  set.seed(1) # seeding to keep results more 'comparable'
  model1 <- train(x=X, y=Y, method="knn", trControl=train_
    config)

  # Linear SVM model
  set.seed(1)
  model2 <- train(x=X, y=Y, method="svmLinear", trControl=
    train_config)

  # Naive Bayes model
  set.seed(1)
  model3 <- train(x=X, y=Y, method="nb", trControl=train_
    config)

  # Display the results
  results <- resamples(list(KNN=model1, LinearSVM=model2, NB
    =model3))
  print(summary(results))
}
# Run the main function
main()
```

```
[1] 942  11
[1] 942  10
NULL


Models: KNN, LinearSVM, NB
Number of resamples: 10

Accuracy
             Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
KNN       0.7447  0.7872 0.8085 0.7983  0.8115 0.8617    0
LinearSVM 0.6383  0.6622 0.6862 0.6931  0.7101 0.7604    0
NB        0.6809  0.6941 0.7128 0.7291  0.7527 0.8085    0

Kappa
             Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
KNN       0.4894  0.5745 0.6170 0.5965  0.6230 0.7234    0
LinearSVM 0.2766  0.3245 0.3723 0.3861  0.4202 0.5208    0
NB        0.3617  0.3883 0.4255 0.4583  0.5053 0.6170    0
```

As we can see from the results, the KNN trained model seem to have best average accuracy score, so this would be our choice in this case.