EE379K Enterprise Network Security Lab 4 Report

Student: Sean Wang, szw87
Professor: Mohit Tiwari, Antonio Espinoza
Department of Electrical & Computer Engineering
The University of Texas at Austin

November 10, 2019

Exercise 1

Part 1

For this part, some additional data processing was needed before the pipeline. The following are the steps taken:

```
# get data from CSVs
train = pd.read_csv(
  'data/exercise1/UNSW_NB15_training-set.csv',
  index_col='id'
)
test = pd.read_csv(
  'data/exercise1/UNSW_NB15_testing-set.csv',
  index_col='id'
)
# remove all rows that are not data on 'Normal' or 'Fuzzers'
# attacks and make a train/test split
train_data = train.loc[
  (train['attack_cat'] == 'Normal') |
  (train['attack_cat'] == 'Fuzzers')
]
test_data = test.loc[
  (test['attack_cat'] == 'Normal') |
  (test['attack_cat'] == 'Fuzzers')
```

```
]
train_labels = train_data['attack_cat']
test_labels = test_data['attack_cat']
# keep only features between 'spkts' and 'is_sm_ips_ports'
train_data = train_data.iloc[:,4:42]
test_data = test_data.iloc[:,4:42]
```

Part 2

In this part, the code was similar except the train and test CSVs were combined and the train/test split was created by randomly dividing the combined DataFrame.

```
# combine CSVs into one DataFrame
com = pd.concat([train, test], ignore_index=True)
...
# after filtering for columns and rows like in Part 1,
# create train/test split randomly
X_train, X_test, y_train, y_test
  = train_test_split(com_data, com_labels, test_size=0.5)
```

Questions

The inference and results section was similar for both parts, since the process of creating a model, fitting it to the data, and then predicting on test data is similar besides the data used to fit the model. The positive label is 'Fuzzers' and the results are as follows:

```
TRAIN PREDS  part1                part2
precision = 0.9963600264725347  0.9744233104006476
   recall = 0.9934015176509403  0.9851076016692578
f1measure = 0.9948785726086238  0.979736328125
 accuracy = 0.9985602155032279  0.9915050406836907

TEST PREDS  part1                part2
precision = 0.8503908380294491  0.7202686685104702
   recall = 0.25725912890453145 0.758004158004158
f1measure = 0.3950179438463163  0.7386547811993517
 accuracy = 0.8068451418095546  0.8899749245176808
```

1) In part 1, the precision and recall of the predicted training labels are about 0.9964 and 0.9934, respectively. On the other hand, the precision

and recall scores of the predicted testing labels are about 0.8504 and 0.2573, respectively.

2) The results on the training data were all above 0.99 while the results on the test data were much lower for recall and f1 measure. The decrease of each score in general is likely since the test data has entries that are dissimilar to the entries in the training data. In particular, the low recall score in the test predictions for part 1 is likely due to a much greater number of false negatives than false positives, since the only difference in the formulas for precision and recall is false positives and false negatives, respectively. This means that in the test data, the model trained on the training data had a significant number of predictions of 'Normal' attacks that were actually 'Fuzzers' (false negatives), in comparison to the number of false positives, where 'Fuzzers' was predicted, but the true label was 'Normal.'

3) The test predictions have a much more balanced spread of scores since the classifier was exposed to data from the test set while training. Overall, this produces a better classifier, since accuracy (a ratio of the correctly predicted labels to the total number of data points) has gone up, despite precision going down. This simply means that in part 2, the number of false positives out of all the positive label predictions is higher compared to the ratio in part 1. In this case, if the organization is worried about fuzzing attacks, then it might be better to have more false positives and be more wary. Additionally, the recall score went up in part 2, meaning that there are relatively less false negatives in the predictions than in part 1. In other words, there are less occurrences of an attack being labeled as 'Normal' when it actually is 'Fuzzers.' Again, this is a good thing when the goal is to separate fuzzing attacks from normal attacks.

Exercise 2

This exercise was similar to the previous exercise in terms of building the model and getting predictions. However, this time accuracy score and a confusion matrix were used to measure performance:

```
from sklearn import metrics
labels_uniq = csv_data['label'].unique()
acc = metrics.accuracy_score(y_test, preds)
matrix = metrics.confusion_matrix(
```

```
  y_test, preds,
  labels=labels_uniq
)
mat_df = pd.DataFrame(
    matrix,
    index=['t:'+x for x in labels_uniq],
    columns=['p:'+x for x in labels_uniq]
)
```

Printing these out resulted in the following (p for predicted, t for true):

```
Accuracy: 0.817503729487817
Confusion Matrix:
[[ 288   71    2    3    2    2    6    3]
 [  59  646   50    6   14    2   35    5]
 [   6   84   52    4    0    1    5    2]
 [   7   27    5  371    5    0   16    1]
 [   6   42    4   14   55    2    2    1]
 [   3    9    2    0    0  512    1    1]
 [  26  103   10   22   16   13  253    0]
 [   0   16    8    1    1    7    1 1111]]
Confusion Matrix with Labels:
                 p:AUDIO  p:BROWSING  p:CHAT  p:FILE-TRANSFER
t:AUDIO              288          71       2                3
t:BROWSING            59         646      50                6
t:CHAT                 6          84      52                4
t:FILE-TRANSFER        7          27       5              371
t:MAIL                 6          42       4               14
t:P2P                  3           9       2                0
t:VIDEO               26         103      10               22
t:VOIP                 0          16       8                1
                 p:MAIL  p:P2P  p:Video  p:VOIP
t:AUDIO               2      2        6       3
t:BROWSING           14      2       35       5
t:CHAT                0      1        5       2
t:FILE-TRANSFER       5      0       16       1
t:MAIL               55      2        2       1
t:P2P                 0    512        1       1
t:VIDEO              16     13      253       0
t:VOIP                1      7        1    1111
```

Then, sorting the features by importance:

```
ft_imps = [
  ft for ft in zip(
    data.columns[5:28],
    model.feature_importances_
  )
]
ft_imps.sort(key=lambda ft: ft[1], reverse=True)
```

Feature	Importance
Flow IAT Min	0.11527062510580362
Bwd IAT Min	0.10232280517910446
Fwd IAT Min	0.09100448643274058
Idle Mean	0.08865546761730136
Bwd IAT Std	0.08159359283202991
Fwd IAT Mean	0.07670357789533365
Bwd IAT Max	0.06202825575499599
Fwd IAT Std	0.06135180168000219
Bwd IAT Mean	0.05392637364959135
Active Max	0.05030235612694543
Active Mean	0.04471946295225557
Active Min	0.0438926147529209
Active Std	0.0428066064735526
Flow IAT Max	0.038563394008160654
Fwd IAT Max	0.03062073033687629
Idle Std	0.0038529394122149645
Idle Min	0.0038186252231571143
Idle Max	0.0

Questions

1) The overall accuracy using the default parameters is about 0.8175, or 81.75%.

2) The confusion matrix is shown above. The model seems to perform very well for 'VOIP' and 'P2P.' On the other hand, some classes where the model performs poorly are 'CHAT' and 'MAIL.'

3) The top 5 most important features, as shown above, are: Flow IAT Min, Bwd IAT Min, Fwd IAT Min, Idle Mean, and Bwd IAT Std.

4) There are a lot of hyperparameters that can be changed to tune the model and to improve performance. Some of these include: `n_estimators`

(number of trees in the forest), `max_depth` (max depth of a tree), and `min_samples_leaf` (minimum number of samples to be considered a leaf). After a few tries of modifying these parameters, the best accuracy obtained was about 0.8329 or 83.29%, as shown in `lab4_exercise2_tuning.ipynb`. However, if the model is extremely overfit to the test data set, then an accuracy of 100% could be possible, although extremely unlikely.

5) Combining lists of values for several hyperparameters into a dictionary and performing cross validation on the combinations can help automatically tune the hyperparameters. For example, using `sklearn.model_selection.GridSearchCV` will return a model with the best performing hyperparameter values. Additionally, using multiple train/test splits can also help automatically tune the hyperparameters. In this case, using `sklearn.model_selection.StratifiedKFold` can be helpful, since it also preserves the percentage of samples for each class. K-fold cross validation can then help determine how to split the data for the model to perform the best.

References