컴퓨터 보안

Assignment #2 Report

컴퓨터소프트웨어학부
2018008395
박정호

1. 실행 환경
   A. 사용한 라이브러리
      i. python의 numpy

      필자가 구현한 모델은 API sequence를 벡터의 형태로 나타내서, 머신 러닝을 통해서 classifier를 구현하는 방식이다. 여기서, 벡터 관련 method와 class를 사용하기 위해서 numpy를 사용했다. 물론 파이썬에서 제공하는 List를 사용하는 방식으로도 충분히 구현할 수 있었겠지만, 좀 더 간단하고 직관적인 구현을 위해서 numpy를 사용했다. 버전은 다음과 같다.

```
numpy                    1.18.2
```

      ii. python 의 scikit-learn

      앞에서도 언급했다시피, 필자는 머신 러닝 모델을 통해서 malware classifier 를 구현했다. 따라서 이 머신 러닝 모델을 사용할 수 있는 라이브러리가 필요했는데, 많은 라이브러리 중 scikit-learn 을 선택했다. 과제 명세에 언급된 라이브러리이기도 하며, 같은 학기에 수강 중인 "인공 지능"에서 배운 classifier 들의 기본 구현이 거의 갖춰진 이 라이브러리가 구현의 편의성에 더 도움이 될 것 같았기 때문이다. 버전은 다음과 같다.

```
scikit-learn             0.23.2
```

   B. python 버전와 운영체제

   사용한 언어는 위에서 언급했듯이 파이썬이며, 버전은 아래의 스크린샷에서 확인할 수 있다. 또한 운영체제는 윈도우 10 을 사용했다.

```
C:₩Users₩pch68>python --version
Python 3.8.2
```

2. 구현

A. 기본 구상

기본적으로는 각 trace 별로 API 를 몇 번 호출했는지 계산한 후에, 그 호출 횟수를 벡터화해서 각 trace 의 identity 로 삼았다. 이 identity 의 차이를 분석하게 될 경우, malware 와 정상 소프트웨어를 구별할 수 있을 것이라 생각했다.

classifier 모델은 머신 러닝 모델 중 nearest neighbor model 을 선택했다. 이유는 다음과 같은데, 우선 API sequence 에서 API frequency 를 분석하는 필자의 방식이 상당히 naïve 했기 때문이다. 따라서 malware 와 정상 소프트웨어의 frequency 벡터의 분포가 linear separable 할 것이라는 확신이 없었고, 이로 인해서 logistic regression 등의 linear classifier algorithm 을 사용하기 어렵다고 판단했다. 그래서 단순히 유클리드 거리가 가까운 데이터에 기반해서 classify 를 진행하는 모델이 더 효과적일 것이라 생각했다.

또한 다른 classify 알고리즘에 대해서는 필자가 아는 바가 거의 없었기 때문에, 고도의 알고리즘을 사용해서 더 괜찮은 결과를 얻게 되더라도, 실제로 필자가 하게 되는 것은 단순히 API 를 따라 적는 것뿐이라고 생각했다. 따라서 필자가 기본적인 지식이 있는 nearest neighbor model 을 사용하게 되었다.

필자가 구현한 모델의 기본적 흐름은 다음과 같다.

i. 전체 API 의 수 파악 및 API 와 정수값의 매핑
ii. API sequence 의 벡터화
iii. 벡터화한 데이터를 이용한 기계 학습 (K-Nearest Neighbor Method)

각 단계의 세부 설명은 다음 페이지의 2.B 부터 설명하겠다.

B. 1 단계 : 전체 API 의 수 파악 및 API 와 정수값의 매핑

각 API 의 호출 횟수를 벡터화하기 위해서는 두가지 정보가 필요했다.

첫번째, API 는 총 몇 가지가 존재하는가?

이 정보가 있어야 각 API sequence 를 벡터화할 때 그 벡터들의 차원을 똑같이 맞출 수 있었다. 그리고 벡터의 차원이 같아야 그 벡터들을 학습의 데이터셋으로 사용할 수 있었다.

두번째, 각 API 는 벡터의 몇번째 항에 대응하는가?

이 정보가 있어야 API sequence 를 벡터화할 때 각 API 의 호출 횟수를 기록할 때 같은 API 는 같은 항에 대응시킬 수 있었다. 만약 *GetSystemTimeAsFileTime* 라는 API 의 호출 횟수가 1 번 API sequence 벡터에는 0 번째 항, 2 번 API sequence 벡터에는 10 번째 항, ... 이런 식으로 일관성 없이 저장되어 있다면, 이 벡터들을 가지고 학습을 했을 때 유효한 결과를 얻을 수 없을 것이다. 1 번 벡터에서 0 번째 항에 저장되었다면 2 번 벡터에서도 0 번째 항에 저장되어야 하기 때문이다. 따라서 각 API 의 종류 별로 벡터의 특정 인덱스에 대응할 수 있도록 매핑해야 했다.

```
10  def construct_API_list():
11      print("construct API list...")
12      API_list = {}
13      cnt = 0
14      for filename in os.listdir('API/0/'):
15          file = open('API/0/'+filename)
16          while True:
17              line = file.readline()
18              if line == "":
19                  break
20              if not (line in API_list):
21                  API_list[line] = cnt
22                  cnt += 1
23      for filename in os.listdir('API/1/'):
24          file = open('API/1/'+filename)
25          while True:
26              line = file.readline()
27              if line == "":
28                  break
29              if not (line in API_list):
30                  API_list[line] = cnt
31                  cnt += 1
32      print("construct API list complete")
33      return API_list, cnt
```

필자가 구현한 소스 코드는 위와 같다. 전체 파일을 순회하면서 dictionary 에 없는 API 가 발견되면 API_list 에 그 API 와 현재 cnt 를 매핑하고, cnt 를 하나 늘리게 된다. 이를 통해서 API_list 에 각 API 에 해당하는 index 를 매핑하고, cnt 에 전체 API 의 수를 저장하게 된다.

C. 2 단계 : API sequence 의 벡터화

1 단계에서 API – index 매핑과, API 수를 모두 구했으니 이제 API sequence 를 벡터화할 수 있게 된다. 총 N 종류의 API 가 존재한다면 벡터의 차원도 N 차원이 된다. 즉, a 번째 API sequence 의 b 번 API 호출 횟수는 a 번째 벡터의 b 번째 항에 해당하게 된다. 횟수를 세는 것은 단순히 API sequence 파일을 한 줄씩 읽으면서 각 API 에 대항하는 항의 값을 1 씩 늘리기만 하면 된다.

단, 여기서 필자가 필요한 것은 API 호출 "횟수"가 아니라 빈도이고, 이 빈도를 API 호출의 "비율"로 둘 것이기 때문에, 생성한 벡터를 단위 벡터화해서 모든 벡터의 길이를 1 로 맞춘다.

```
35  def load_data(API_list, cnt):
36      print("load API sequence data...")
37      data = []
38      label = []
39      print("\tconstruct frequency vector #goodware")
40      for filename in os.listdir('API/0/'):
41          file = open('API/0/'+filename)
42          row = np.zeros((cnt,))
43          while True:
44              line = file.readline()
45              if line == "":
46                  break
47              row[API_list[line]] += 1
48          data.append(normalize_vector(np.array(row)))
49          label.append(0)
50      print("\tconstruct frequency vector #malware")
51      for filename in os.listdir('API/1/'):
52          file = open('API/1/'+filename)
53          row = np.zeros((cnt,))
54          while True:
55              line = file.readline()
56              if line == "":
57                  break
58              row[API_list[line]] += 1
59          data.append(normalize_vector(np.array(row)))
60          label.append(1)
61
62      print("\tsplit data for training and test")
63      train_data, test_data, train_label, test_label = train_test_split(data, label, test_size=0.1, random_state=42)
64      print("load API sequence data complete")
65      return train_data, train_label, test_data, test_label
```

필자가 구현한 코드는 위와 같은데, 앞에서 설명한 대로 모든 API sequence 파일을 순회하면서 각 파일을 끝까지 읽으면서 각 파일에 대한 sequence 벡터를 생성하게 된다. 여기서, normalize_vector 라는 함수를 이용해서 생성한 API 호출 횟수 벡터를 API 호출 빈도 벡터로 바꾸게 되는데, 이 함수는 벡터를 그 절댓값으로 나누는 식으로 단위 벡터화를 진행한다. 또한 label 이라는 리스트에 해당 벡터의 class 가 malware 인지 정상 소프트웨어인지를 같이 기록해서 후의 지도 학습에 이용할 수 있게 한다.

마지막에 train_test_split 함수를 통해서 학습용 데이터와 테스트용 데이터를 분리하는데, 10%의 데이터만 테스트용으로 두었고, 그 순서는 랜덤으로 섞이게 했다.

D. 3 단계 : 벡터화한 데이터를 이용한 기계 학습 (K-Nearest Neighbor Method)

앞에서 구한 데이터를 라이브러리에서 구한 머신 러닝 모델에 넣어서 학습을 하는 단계이다. Nearest Neighbor Method 중에서도 K-Nearest Neighbor Method 를 사용했고, 이 방식은 어떤 데이터의 classify 를 위해 유클리드 거리가 가까운 K 개의 데이터를 이용하는 방식이다. (이때 이 K 개의 데이터는 이미 학습된 데이터들이다.) K 개의 데이터들의 class 를 보고 가장 가까운 class 를 classify 할 데이터의 class 로 정하는 것이다. 물론 이때 "가장 가까운"의 기준으로 각 데이터들의 유클리드 거리를 가중치로 두던가, 단순히 가장 많이 등장한 class 를 사용하는 등 여러가지 방식이 있을 수 있다.

```
86  def main():
87      API_list, cnt = construct_API_list()
88      print("total # of API : %d" % cnt)
89      train_data, train_label, test_data, test_label = load_data(API_list, cnt)
90      print("supervised learning...")
91      classifier = KNeighborsClassifier(n_neighbors = 10, weights = 'distance')
92      classifier.fit(train_data, train_label)
93      print("supervised learning complete")
94      print()
95      print("accuracy : %f" % classifier.score(test_data, test_label))
96      print()
97      print("manual test")
98      while(True):
99          filename = input("input filename, or \"quit\" to stop test > ")
100         if filename == "quit":
101             break
102
103         data = load_single_data(API_list, cnt, filename)
104         if data != None:
105             print("predicted : %s" % str(classifier.predict(data)))
106     print()
107     print("all data test (validation data)")
108     AC = 0
109     WA = 0
110     for i in range(len(test_label)):
111         if classifier.predict([test_data[i]])[0]==test_label[i]:
112             AC+=1
113         else:
114             WA+=1
115     print("total test count : %d" % AC+WA)
116     print("correct : %d wrong : %d" % (AC, WA))
```

필자가 구현한 코드는 위와 같은데, 라이브러리를 그대로 사용했기 때문에 그리 복잡하지는 않다. KNeighborsClassifier 를 사용해서 K-Nearest Neighbor Method 를 구현했고, 이때 classify 에 사용할 이웃의 수 K 를 10 으로, weights 는 'distance'로 두어서 거리가 가까운 이웃이 classify 에 미치는 영향이 크도록 했다.

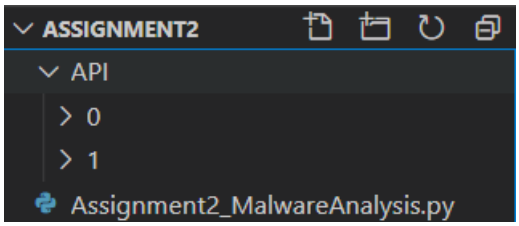
이후 앞에서 구한 학습 데이터와 학습 라벨을 가지고 학습을 진행했고(classifier.fit() 부분), 테스트 데이터와 라벨을 가지고 모델의 정확성이 어느정도인지 확인했다. 이후 수동 테스트와 직접 구현한 전체 테스트 코드를 추가했다.

3. 실행

A. 실행 시 유의사항

제출한 소스코드를 실행하기 위해서는 우선 과제 공지 시 같이 제공된 API 압축파일을 소스코드와 같은 디렉토리에 압축해제해야 한다.

Assignment2 디렉토리 내에 API 폴더와 .py 소스코드가 같이 있는 것을 확인할 수 있다.



B. 실행 스크린샷

```
PS C:\Users\pch68\2020_ite4007_2018008395\Assignment2> python .\Assignment2_MalwareAnalysis.py
construct API list
construct API list complete
total # of API : 301
load API sequence data...
        construct frequency vector #goodware
        construct frequency vector #malware
        split data for training and test
load API sequence data complete
supervised learning...
supervised learning complete

accuracy : 0.909091

manual test
input filename, or "quit" to stop test > API/0/107.txt
predicted : [0]
input filename, or "quit" to stop test > API/0/869.txt
predicted : [0]
input filename, or "quit" to stop test > API/0/2113.txt
predicted : [0]
input filename, or "quit" to stop test > API/1/827.txt
predicted : [1]
input filename, or "quit" to stop test > API/1/1151.txt
predicted : [1]
input filename, or "quit" to stop test > API/1/3621.txt
predicted : [0]
input filename, or "quit" to stop test > quit

all data test (validation data)
total test count : 671
correct : 610 wrong : 61
```

Output (Ctrl+Shift+U)

우선 라이브러리로 구한 정확도가 0.909091 즉 약 91% 정도 되는 것을 확인할 수 있고, 직접 파일명을 입력했을 때 classify 가 거의 다 잘되는 것을 볼 수 있다. 단, 6 번째 경우와 같이 잘못 classify 되는 경우도 역시 확인할 수 있다.

마지막의 all data test 는, 테스트용으로 만들어진 데이터들에 대해서 일일히 테스트를 돌려봄으로써, 맞게 classify 된 것과 틀리게 classify 된 것의 개수를 세어본 것인데, 맞게 된 것이 610 개, 틀리게 된 것이 61 개로 처음에 구한 약 91%와 같은 결과가 나온 것을 확인할 수 있다. (이 테스트에 대한 코드는 위의 main 함수에 그대로 나타나 있다.)