**컴퓨터 보안 Assignment2 Report**

**2018008613 안상욱**

1. **전반적인 설명**

저는 동적 분석 결과 나오는 API를 이용해 악성코드인지 정상코드인지 분류하는 프로그램을 구현했습니다.

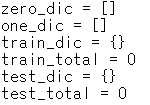
Native API 논문을 참고해 프로그램을 구현했습니다. 우선 파이썬 train\_test\_split API를 이용해 다운받은 API 자료에서 악성코드, 정상코드를 train set, test set으로 각각 나누어주었습니다. 이 때 train set과 test set의 비율을 9:1로 설정해 주었습니다.

그리고 빈도수 중심 분석 TF-IDF을 사용했습니다. TF = log( 1 + w / d) 에서 w는 문서 내 특정 API의 등장 횟수, d는 문서 내 전체 API 수입니다. 그리고 IDF = N / n 에서 N은 전체 문서 수, n은 특정 API가 등장하는 문서 수를 뜻합니다. 이를 이용해 train set에 대해 각 문서에 있는 API마다 TF \* IDF값을 이용해 TF-IDF 값을 각각 구하고 모든 TF-IDF값의 평균을 구했습니다. 그리고 악성 코드에서 평균 TF-IDF 값보다 큰 값을 가지면서 정상코드에서는 등장하지 않는 API를 가지고 virus list를 만들어서 이 virus가 파일에 있다면 악성파일로 간주하는 프로그램을 작성했습니다.

1. **코드 구현**



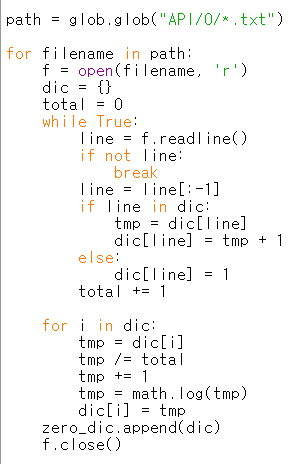
다음과 같은 모듈을 사용했습니다.



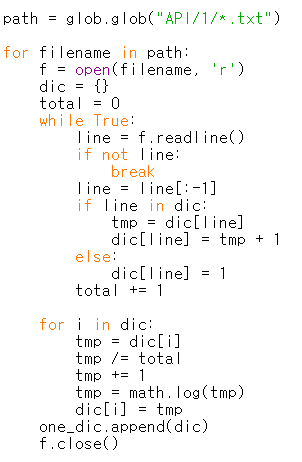
zero\_dic은 0번 폴더, 즉 정상파일 샘플에서 각각의 txt파일에 대해 API당 TF값을 dictionary로 나타냈습니다. 예를 들어 첫번째 txt파일에서 A라는 API의 TF값이 0.7이라면 zero\_dic[0] = {'A' : 0.7 , .....} 처럼 표현했습니다. one\_dic은 1번 폴더, 즉 악성파일 샘플에서 동일하게 저장했습니다.

train\_total과 test\_total은 각각 train할 txt파일의 총 개수, test할 txt파일의 총 개수를 저장했습니다.

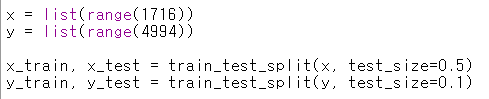
train\_dic은 train파일 중 해당 API가 나오는 파일의 수를 저장했습니다. 예를 들어 A라는 API가 총 1000개의 파일 중 300개의 파일에 존재했다면 train\_dic = {'A' : 300 , ......} 처럼 표현했습니다. test\_dic은 test파일들에 대해 동일하게 저장했습니다.



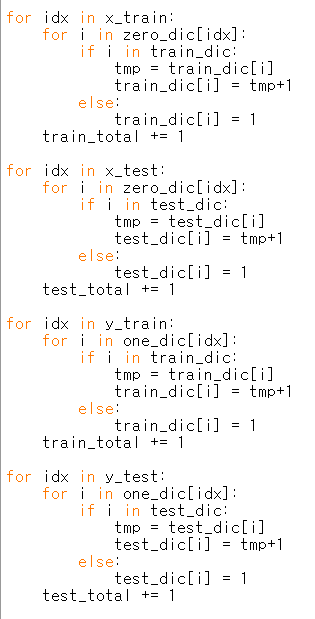
API 폴더의 0폴더 안에 있는 정상파일들을 위에서 설명한 것처럼 zero\_dic에 각 API의 TF값을 dictonary 형태로 저장했습니다. 처음에 dic이라는 빈 dictionary를 선언해 주었습니다. 그 이후 total이라는 변수를 선언해 주었는데 해당 txt파일에서 등장하는 총 API의 개수를 저장합니다. 해당 txt파일에 등장하는 API의 TF값을 TF = log( 1 + w / d) 식을 이용해 구해줍니다.



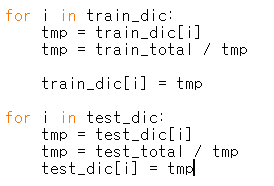
1번 폴더의 악성파일들에 대해서도 똑같이 one\_dic에 처리해줍니다.



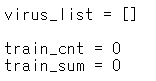
악성파일, 정상파일 각각을 train set, test set으로 나누어줍니다. X는 정상파일들을, y는 악성파일들을 나타내고 악성파일은 train : test = 9 : 1로, 정상파일은 train : test = 1 : 1로 나누어주었습니다.



위에서 언급했듯이 train에서 쓰일 txt파일의 총 개수를 train\_total에, test는 test\_total에 저장해 주었습니다. train\_dic은 train파일 중 해당 API가 나오는 파일의 수를 저장했습니다. 예를 들어 A라는 API가 총 1000개의 파일 중 300개의 파일에 존재했다면 train\_dic = {'A' : 300 , ......} 처럼 표현했습니다. test\_dic은 test파일들에 대해 동일하게 저장했습니다.

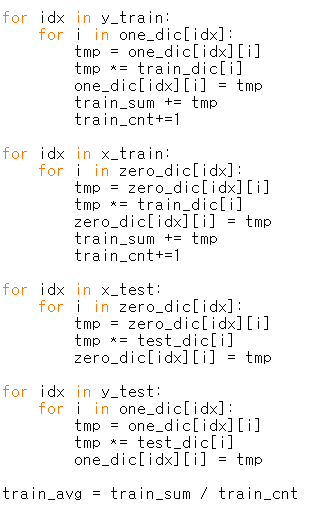


Train\_dic에 각 API마다 IDF값을 저장했습니다. IDF값은 N / n 식을 이용해 구했습니다. 예를 들어 A라는 API가 총 900개의 파일 중 300개의 파일에 존재했다면 IDF값은 900 / 300 = 3이 될거고, train\_dic = {‘A’ : 3 , …}처럼 표현했습니다. Test\_dic은 test파일들에 대해 동일하게 저장했습니다.



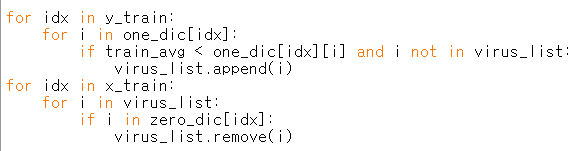
Virus\_list에는 해당 API가 들어가 있으면 악성파일로 판단되는 API들을 리스트로 표현했습니다. 이 리스트에 추가하는 기준은 train 파일들에서 TF-IDF값을 TF \* IDF라고 하고 모든 train파일의 API에 대해 TF-IDF값의 평균을 구해서 악성파일 train set에서 TF\*IDF 값이 평균 이상이면서 정상파일 train set에 들어가 있지 않은 API들을 선정했습니다.

Train\_cnt와 train\_sum은 모든 train파일의 API의 TF-IDF값의 평균을 구하기 위해 사용되었습니다.

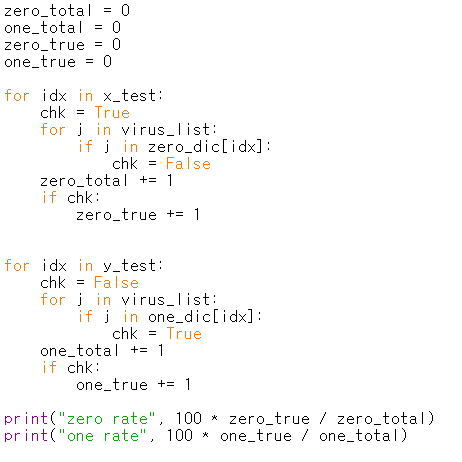


One\_dic에는 악성파일에서 해당 txt파일의 API에 대한 TF-IDF값들을 저장했습니다. 예를 들어 악성파일 중 첫번째 txt파일의 A라는 API의 TF값이 0.5이고, IDF값이 4라면 TF-IDF값은 0.5 \* 4 = 2이므로, one\_dic[0] = {‘A’ : 2 , …} 처럼 저장했습니다. Zero\_dic에 대해서도 똑같이 실행했습니다.

이 과정을 수행하면서 train set의 전체 TF-IDF값의 평균을 train\_avg에 저장했습니다.



앞에서 설명했듯이 악성파일 train set에서 TF\*IDF 값이 평균 이상이면서 정상파일 train set에 들어가 있지 않은 API들을 virus\_list에 저장했습니다. 이 virus\_list에 존재하는 API를 가지고 있는 파일을 악성파일로 분류했습니다.



앞에서 말한 분류기준을 이용해 test\_set에서 악성파일을 분류해냈습니다. Zero\_total은 test에 사용되는 전체 정상파일의 수이고, zero\_true는 전체 정상파일 중 테스트 결과 정상파일로 분류된 파일의 수입니다. 따라서 정상파일들에 대해서 이 테스트를 진행한 결과 정확도는 100 \* zero\_true / zero\_total 로 계산할 수 있습니다. 악성파일에 대해서도 똑같이 실행해 보았습니다.

1. **실행 결과**



다음과 같이 정상파일에 대해서는 98.48%의 정확도로 정상파일로 판단했지만, 악성파일에 대해서는 19.0%의 정확도를 가지고 악성코드를 잡아냈습니다.

1. **결과 분석**

프로그램 구현 결과 정상파일에 대해서는 98.48%라는 높은 정확도를 가졌지만, 악성파일들에 대해서는 19.0%라는 다소 낮은 정확도를 가짐을 확인해 볼 수 있었습니다. 이는 테스트 케이스에 존재하는 대부분의 악성파일들이 서로 다른 API를 이용해 악성 행위를 하여 train set에서 찾아낸 virus들의 목록에 존재하지 않는 API를 이용해 test set에 있는 악성코드들이 활동해서 이런 결과가 나왔다는 예측을 했습니다.

또, 악성 코드들은 악성 행위를 들키지 않기 위해 악성 행위에 사용하는 API 이외에도 불필요한, 정상적인 API들을 여러 번 사용하므로 이런 행위로 인해 악성 행위를 하는 Native API가 숨겨졌을 것이라는 분석을 해 보았습니다.