MALWARE CLASSIFICATION

1. 컴파일 환경

Windows 10 home 64-bit

Python 3.8으로 구현

Python Numpy, Matplotlib 사용

2. 실험 설계

I. N – gram 방식 분석

추가 자료로 동봉된 논문을 참고하였습니다. 위 실험에서는 API를 4 – gram을 통해서 분석하였습니다. 단순히 4개의 API를 엮어서 만들었으며, 불용어 제거 등의 처리는 하지 않았습니다.

II. TF-IDF 방식 분석

마찬가지로 논문을 참고하였습니다. TF는 특정 문서에 특정 4 – gram의 등장 횟수로 설정하였으며, IDF는 특정 4 – gram이 등장한 문서 수인 DF에 역수를 취했습니다. 위 식들에 대한 계산 식은 다음과 같습니다.

$$TF = \frac{w}{d} \ (w : \textit{등장횟수}, d\text{: }\textit{문서의 전체 } n-gram \textit{ 수})$$

$$IDF = \log_{10}\left(\frac{n}{1+df(t)}\right) \ (n\text{: }\textit{전체 문서 수}, df(t)\text{: }\textit{등장한 문서 수})$$

III. 코사인 유사도

문서 마다 4 – gram에 대한 벡터를 만들고, 그 TF x IDF를 저장하였습니다. 그리고 이 결과를 코사인 유사도로 검사하였습니다.

$$similarity = cos(\Theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}}$$

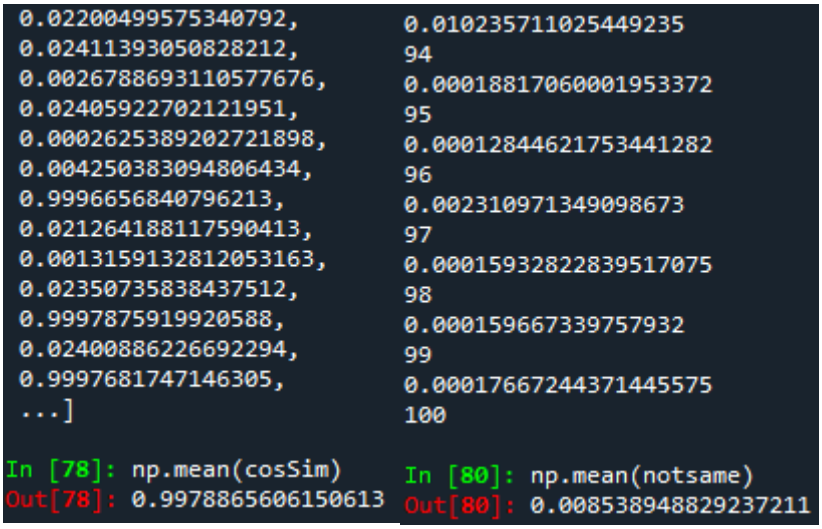
IV. KNN classification

실험데이터와 대조군 데이터를 생성해 kNN 알고리즘을 적용하여 악성코드인지 정상파일인지 확인하였습니다.

3. 실험 내용

I. 데이터 전처리

먼저 실험에 앞서서, TF만 가지고 악성코드 파일 100개, 정상 파일 150개를 크기가 큰 순으로 코사인 유사도를 확인해보았습니다.



왼쪽은 정상코드끼리 만 비교한 것입니다. (150 x 149 / 2 회)

오른쪽은 악성코드와 정상 파일을 비교한 것입니다. (100 x 100 회)

그 결과 값이 한 쪽은 너무 높고, 한 쪽은 너무 낮다는 판단 하에 데이터를 정리하였습니다.

악성코드와 정상 파일 각각 1000개씩 정리하여 사용하였습니다. 크기는 최소 10KB 최대 1200KB이며, 악성코드와 정상 파일의 개수를 크기에 맞추어 정리하였습니다. (ex. 10kb 20개씩, 20kb 10개씩... 100kb 10개씩...)

II. IDF 생성

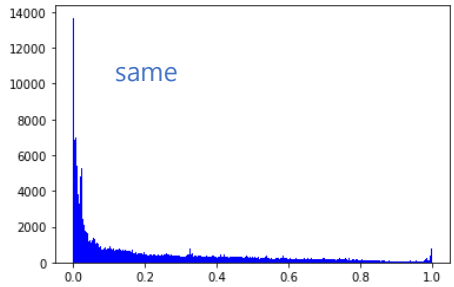
GetIDF.py를 실행시켜, 4-gram IDF를 Dict = {string : int} 꼴로 저장한 IDFLIST.txt 파일을 생성하도록 하였습니다. 4-gram은 전처리된 파일 2000개와 테스트로 사용할 파일 100개에 대해서 생성하도록 하였습니다.

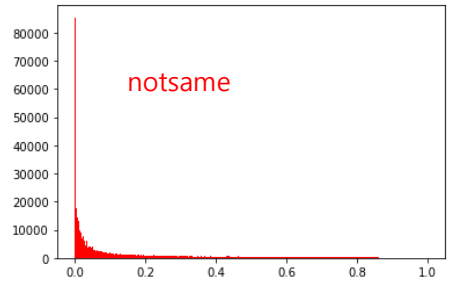
```
GetSystemInfoLdrGetDllHandleLdrGetProcedureAddressLdrGetProcedureAddress 0.7516763548520218
LdrGetDllHandleLdrGetProcedureAddressLdrGetProcedureAddressLdrGetProcedureAddress 0.08492695716646048
LdrGetProcedureAddressLdrGetProcedureAddressLdrGetProcedureAddressRegOpenKeyExW 0.3222192947339193
LdrGetProcedureAddressLdrGetProcedureAddressRegOpenKeyExWRegOpenKeyExW 0.6661210927210873
```

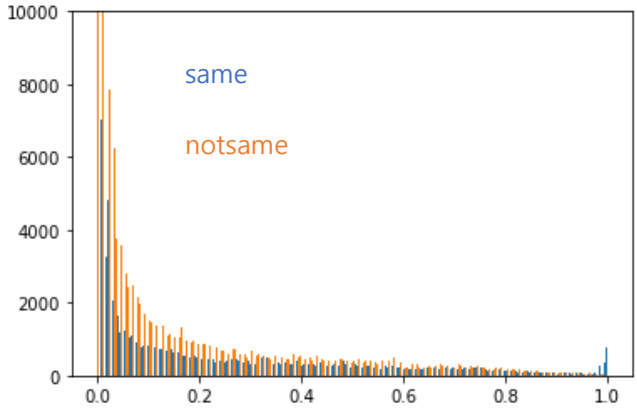
위와 같은 형태로 생성하였습니다.

III. 각 요소 별 코사인 유사도 비교

classification에 앞서, TF를 이용해 대조군 데이터 각 1000개에 대해 코사인 유사도를 확인해보았습니다. GetCosSimMean.py를 실행하였습니다.



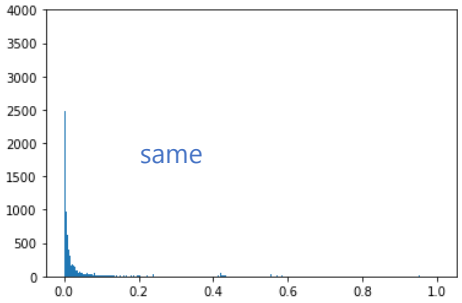


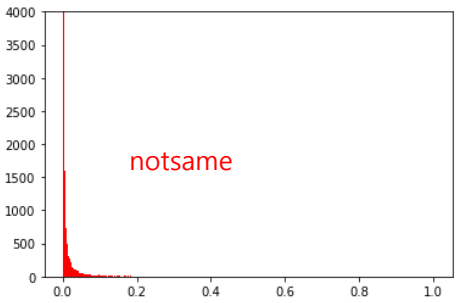


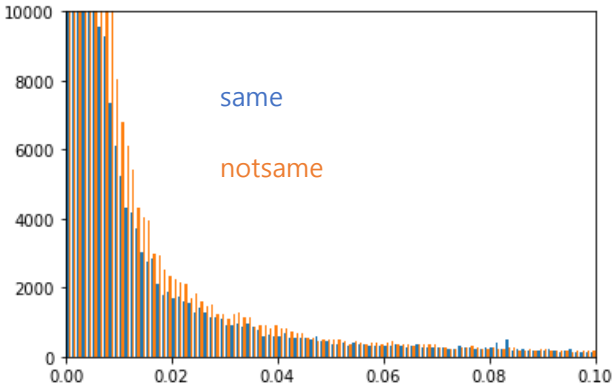

```
In [6]: same
Out[6]: 0.23110149895343238

In [7]: notsame
Out[7]: 0.16322988847073616
```

same은 정상코드끼리의 코사인 유사도, notsame은 정상코드와 악성코드를 비교한 것입니다. (각 1000 x 1000 회 , 1000 x 999 / 2 회)

이 때 두 유사도가 유의미하지 않은 것으로 판단하고 TF-IDF 처리한 상태의 코사인 유사도를 확인하였습니다.







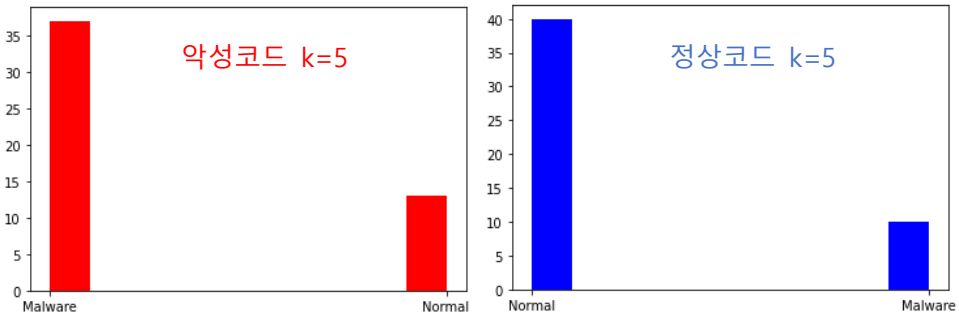

```
In [6]: sameMean
Out[6]: 0.04291225001808879

In [7]: notsameMean
Out[7]: 0.006377167891820636
```

그 결과 둘의 유사도가 7배가량 차이가 나는 유의미한 결과를 얻을 수 있었습니다.

IV. KNN Classification

대조군 데이터 악성코드 1000개, 정상파일 1000개와, 악성코드 테스트 파일 50개, 정상파일 테스트 파일 50개를 kNN classification algorithm을 이용해 분석하였습니다. 테스트 파일은 대조군에 없는 다양한 크기의 데이터 50개씩으로 설정하였습니다.

k값으로 1, 5, 7, 9, 1000을 사용하였으며, k 값이 1일때 가장 높은 정확도를 볼 수 있었습니다. 참고한 논문에서는 k=5를 사용하였는데, 실제 실험에서는 k=1일 때가 가장 높은 정확도를 가졌습니다. 이는 테스트 파일이 50개 밖에 되지 않아 일어난 문제라고 생각합니다.



	k = 1	k = 5	k = 7	k = 9	k = 1000
정상파일	43 /50	37 /50	34 /50	34 /50	49 /50
악성코드	40 /50	40 /50	38 /50	36 /50	9 /50
탐지율	**83 %**	**77 %**	**72 %**	**70 %**	**58 %**

4. 구현 특이 사항

```
for testFile1 in testFiles1:
    cosSim0 = []
    cosSim1 = []

    f1 = open(pathTest1 + testFile1, 'r')
    line1 = f1.readlines()
    dic1 = makeNgram(4, line1)
    size1 = len(line1) - 3

    # 정상 코드와 비교
    for preFile0 in preFiles0:
        f2 = open(pathPre0 + preFile0, 'r')
        line2 = f2.readlines()
        dic2 = makeNgram(4, line2)
        size2 = len(line2) - 3

        dotPro = 0

        for key in dic1.keys():
            if key in dic2:
                dotPro += (dic1[key] / size1 * IDFList[key]) * (dic2[key] / size2 * IDFList[key])

        norm1 = np.linalg.norm(np.array(list(dic1.values())) / size1 * IDFList[key])
        norm2 = np.linalg.norm(np.array(list(dic2.values())) / size2 * IDFList[key])

        cosSim = dotPro / (norm1 * norm2)

        cosSim0.append(cosSim)
        f2.close()
```

4-gram을 모두 구하였을 때 29만여개나 되었기 때문에, TF-IDF 값을 dictionary 꼴로 저장하였습니다. 그래서 두 개의 파일을 비교하여 Dot product를 계산할 때, 동일한 key가 있을 때 계산하는 식으로 구현하였습니다.

```
# kNN classification
k = 5
cnt = 0
mal = 0
nor = 0
while(cnt < k):
    if(cosSim0[0] > cosSim1[0]):
        cosSim0.pop(0)
        nor += 1
        cnt += 1
    elif(cosSim0[0] < cosSim1[0]):
        cosSim1.pop(0)
        mal += 1
        cnt += 1
    elif(cosSim0[0] == cosSim[0]):
        cosSim0.pop(0)
        cosSim1.pop(0)

if(mal > nor):
    res1.append('Malware')
else:
    res1.append('Normal')
```

knn의 경우 악성코드, 정상코드와의 코사인 유사도에서 큰 값을 하나씩 빼서, 값이 더 많은 쪽으로 결정하게 구현하였습니다.

5. 참고 문헌

랜섬웨어 Native API 정보 분석을 통한 탐지 방법 연구

장준영 김우재 옥정윤 임을규 한양대학교

효율적인 악성코드 분류를 위한 최적의 API 시퀀스 길이 및 조합 도출에 관한 연구

최지연 김희석 김규일 박학수 송중석 한국과학기술정보연구원, 과학기술연합대학원대학교

문자열 정보를 활용한 텍스트 마이닝 기반 악성코드 분석 기술 연구

하지희 이태진