Report

빅데이터 응용 보안 보고서

-안드로이드 멀웨어 탐지-

201520932 윤지우

201420956 김도현

목차

프로젝트 개요.....6

프로젝트 선정 이유.....6

프로젝트 탐지 및 분석 방식.....6

프로젝트 진행 방향.....7

프로젝트 흐름도.....8

데이터셋 분석.....9

데이터 수집 방법.....9

데이터의 값 분석 – 권한 데이터.....10

데이터의 값 분석 – 네트워크 패킷.....13

권한 데이터 코드.....18

데이터 전처리 – 권한.....18

모델 훈련 및 평가 – 권한.....18

테스트 환경 분석 – 권한.....20

네트워크 데이터 코드.....21

데이터 전처리 – 네트워크 패킷.....21

모델 훈련 및 평가 – 네트워크 패킷.....22

테스트 환경 분석 – 네트워크 패킷.....25

AUROC 그래프.....27

아레나 분석.....28

결론.....30

<그림 순서>

<그림 1> 프로젝트 분석 프로세스.....7

<그림 2> 프로젝트 흐름도.....8

<그림 3> 권한 데이터.....9

<그림 4> 네트워크 데이터.....9

<그림 5> 권한 데이터 셋 분석.....10

<그림 6> 악성 코드가 요구하는 상위 10개 권한.....,10

<그림 7> 정상 코드가 요구하는 상위 10개 권한.....11

<그림 8> 권한 데이터 수 분포 그래프.....11

<그림 9> 각 권한 데이터 비교 그래프.....12

<그림 10> 네트워크 데이터 속성.....13

<그림 11> 네트워크 데이터 셋.....14

<그림 12> 상관 관계도 그래프.....15

<그림 13> 산점도 그래프.....16

<그림 14> 데이터 조정 코드.....16

<그림 15> 중요도 그래프.....17

<그림 16> 훈련 및 테스트 세트 분리 코드.....18

<그림 17> KNN 모델 권한 데이터.....18

<그림 18> 결정트리 모델 권한 데이터.....18

<그림 19> 랜덤 포레스트 모델 권한 데이터.....19

<그림 20> GaussianNB 모델 권한 데이터.....19

<그림 21> SVM 모델 권한 데이터.....19

<그림 22> 모델 별 실행 시간 비교 – 권한.....20

<그림 23> 데이터 속성 제거.....21

<그림 24> tcp_urg_packet 시각화.....22

<그림 25> 훈련 및 테스트 세트 분리 코드.....22

<그림 26> KNN 모델 네트워크 패킷.....23

<그림 27> 결정트리 모델 네트워크 패킷.....23

<그림 28> 랜덤포레스트 모델 네트워크 패킷.....23

<그림 29> GaussianNB 모델 네트워크 패킷.....23

<그림 30> SVM 모델 네트워크 패킷.....24

<그림 31> 모델 별 실행 시간 비교 – 네트워크.....25

<그림 32> 권한 AUR 그래프.....27

<그림 33> 네트워크 AUR 그래프.....27

<그림 34> 아레나 시뮬레이션 모델.....28

<그림 35> 아레나 결과 그래프 비교.....29

프로젝트 개요

프로젝트 선정 이유

안드로이드 기반의 스마트폰은 언제 어디서나 이용가능한 컴퓨터로 진화함으로써 우리의 삶에 큰 일부를 차지하고 있다. 그러나 현재 이러한 스마트폰은 해커들의 주요 공격 타깃이 되어 바이러스에 감염돼 사용자에게 피해를 주는 사례가 늘고 있다. 휴대전화는 PC와 다르게 프로그램 실행 환경에 제약이 있기 때문에 바이러스 또한 제약이 있을 것이라 여겨진다. 그러나 어플리케이션을 다운 받는 것만으로도 스마트폰에 있는 개인정보를 탈취할 수 있을 뿐만 아니라, 휴대폰 소액결제 등 금전적으로 피해를 입을 수 있다. 특히 안드로이드의 경우 마켓의 개방성과 스마트폰 시장의 높은 점유율로 인해 악성 코드를 쉽게 유포할 수 있다. 이러한 이유로 모바일 악성코드의 대부분은 주로 안드로이드 단말을 공격대상으로 삼고 있다. 따라서 우리는 최소한 이런 악성코드를 탐지할 수 있는 보안 수준을 가져야한다.

프로젝트 탐지 및 분석 방식

악성코드에는 탐지, 분류, 예방 등 여러 가지 분야가 있지만 본 프로젝트에서는 악성코드를 탐지하는 것만 다룰 것이다. 주어진 데이터셋을 훈련데이터와 테스트데이터로 나누어 머신러닝을 통해 해당 어플리케이션이 악성코드인지 아닌지 탐지할 수 있도록 훈련시키는 것이 목표이다. 더 나아가 훈련이 끝난 뒤 알고리즘을 수정하고 훈련 모델을 추가하여 탐지 정확성을 높일 것이다.

이번 프로젝트에서 악성코드 탐지 방식은 어플리케이션이 요구하는 권한과 어플리케이션이 실행되면서 서로 주고받는 네트워크 데이터 두 가지 관점에서 탐지하였다.

어플리케이션이 특정 행동을 할 때 시스템에 권한을 요청하는 과정이 필요하다. 정상적인 안드로이드 어플리케이션과 악성 어플리케이션에서 요구하는 권한이 차이가 있다는 점에 주목하여 데이터를 분석할 것이다.

어플리케이션에서는 PC와 마찬가지로 네트워크를 통해 데이터를 주고받는다. 이 때 정상 어플리케이션과 악성 어플리케이션이 주고 받는 데이터는 분명 차이가 있다. 이러한 정상적인 요청과 비정상적인 요청에 주목하여 데이터를 분석할 것이다.

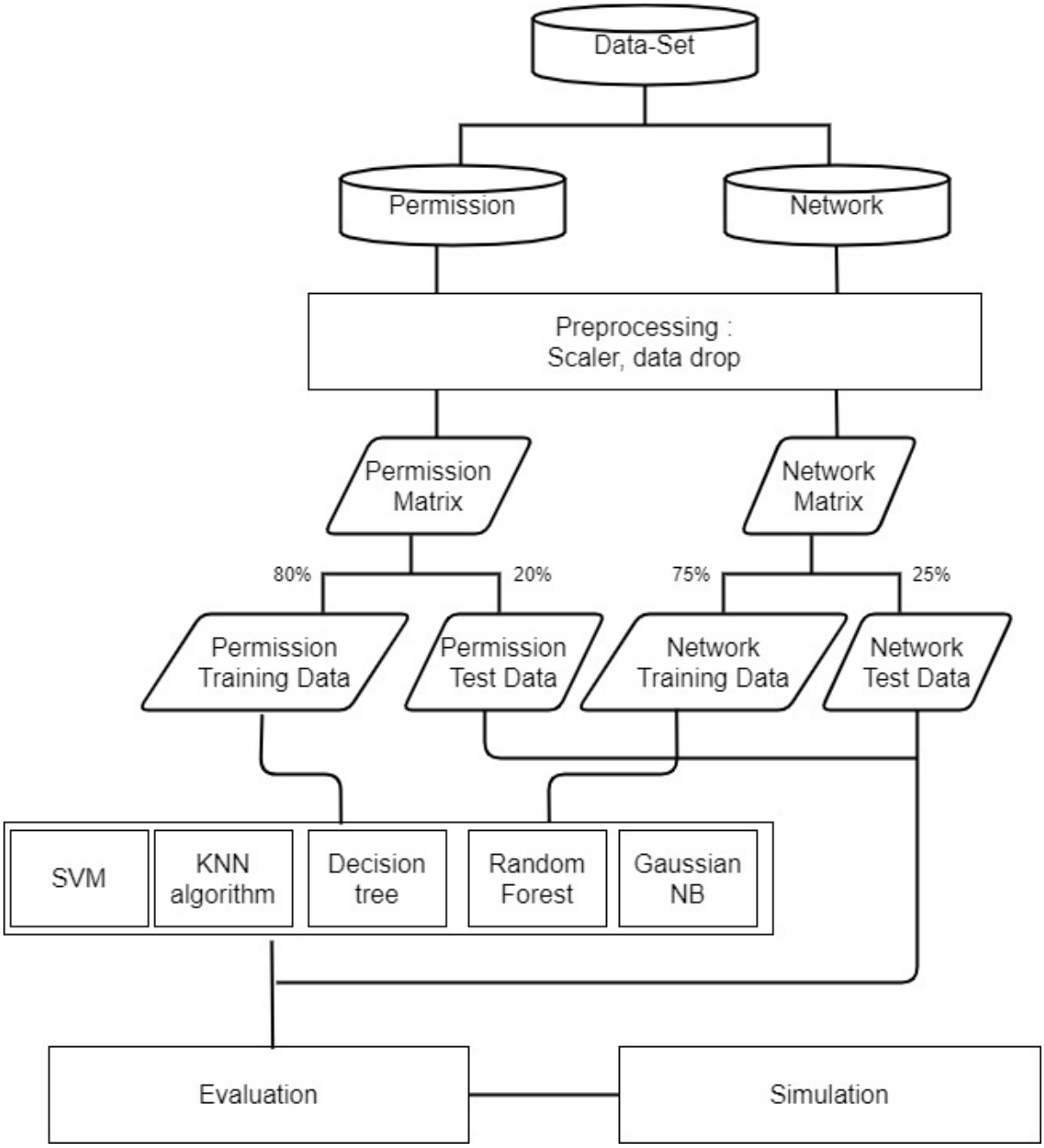
프로젝트 진행 방향

프로젝트는 주제 선정, 데이터 수집 및 가공, 수집한 데이터 자료를 분석, 모델 선택 및 훈련, 테스트 및 평가로 순서대로 이루어진다.

최종 프로젝트 목표는 가장 정확한 결과를 산출해내는 모델을 찾아 모바일 악성코드 탐지하는 것이다.

<그림 1> 프로젝트 분석 프로세스

프로젝트 흐름도

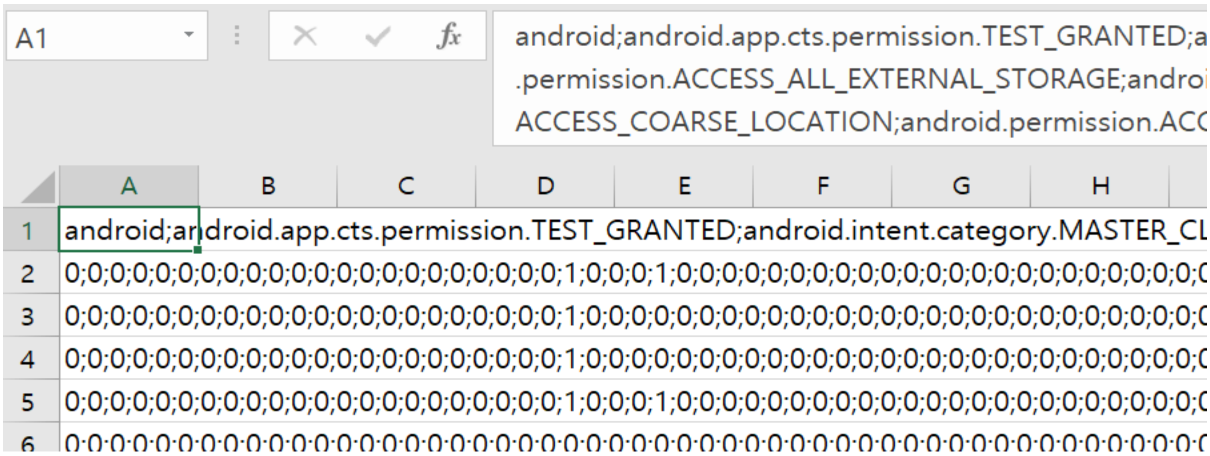


<그림 2> 프로젝트 흐름도

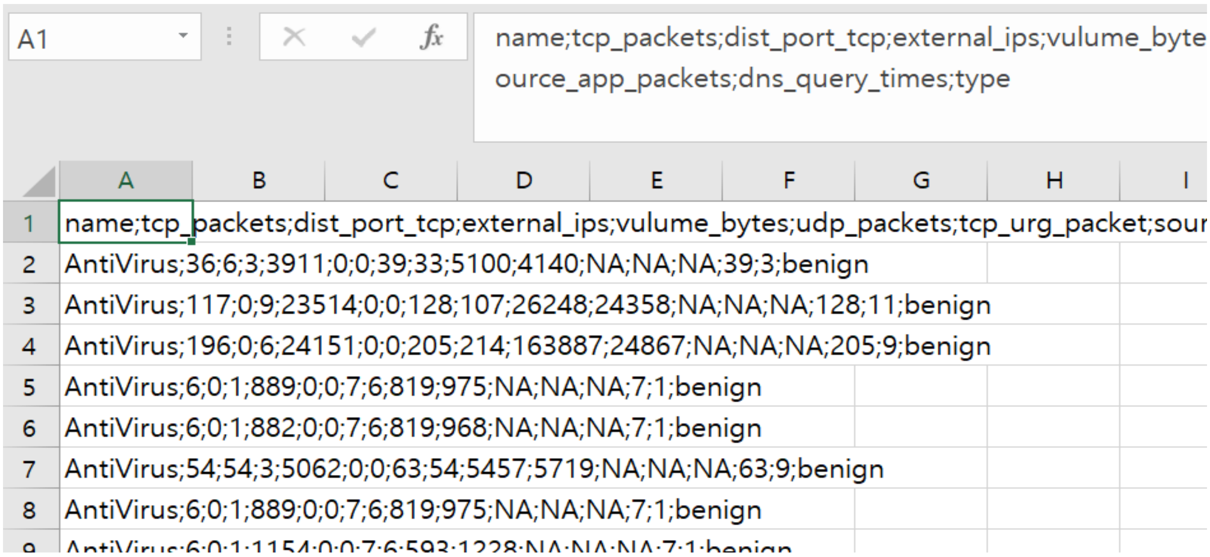
데이터셋 분석

데이터 수집 방법

모델을 훈련시키기 위해서는 많은 양의 데이터가 필요하다. 우리는 많은 데이터를 수집하기 위해 어플리케이션마다 데이터를 뽑아내는 것이 비효율적이라고 생각이 되어 인터넷에서 미리 수집한 데이터셋을 분석하였다. 악성코드 탐지를 위해 사용한 데이터는 어플리케이션의 권한을 모아놓은 데이터와 어플리케이션을 사용하면서 나온 네트워크 데이터 이렇게 총 2가지이다. 데이터는 csv형식으로 저장되어있다.



<그림 3> 권한 데이터



<그림 4> 네트워크 데이터

데이터의 값 분석 – 권한 데이터

권한 데이터는 어플리케이션에서 요구하는 권한과 해당 어플리케이션이 악성코드인지 정상코드인지 분류해 놓은 데이터이다.

분석할 데이터는 그림 2에서 확인할 수 있듯이 총 398개의 데이터 크기와 331개의 특징들을 가지고 있다. 이때 각 특징들은 안드로이드의 권한 요청을 의미하며 type이 0이면 정상적인 어플리케이션, type이 1이면 악성코드이다.

```
In [3]: df = pd.read_csv("./input/datasets_android.csv", sep=";")
        df = df.astype("int64")
        df.type.value_counts()

Out[3]: 1    199
        0    199
        Name: type, dtype: int64

In [4]: df.shape

Out[4]: (398, 331)
```

<그림 5> 권한 데이터 셋 분석

다음은 악성 코드로 분류된 어플리케이션에서 요구하는 상위 10개의 권한이다.

```
In [5]: # 악성 코드 샘플에 사용되는 상위 10개의 권한
        # Malicious
        pd.Series.sort_values(df[df.type==1].sum(axis=0), ascending=False)[1:11]

Out[5]: android.permission.INTERNET                 195
        android.permission.READ_PHONE_STATE         190
        android.permission.ACCESS_NETWORK_STATE     167
        android.permission.WRITE_EXTERNAL_STORAGE   136
        android.permission.ACCESS_WIFI_STATE        135
        android.permission.READ_SMS                 124
        android.permission.WRITE_SMS                104
        android.permission.RECEIVE_BOOT_COMPLETED   102
        android.permission.ACCESS_COARSE_LOCATION    80
        android.permission.CHANGE_WIFI_STATE         75
        dtype: int64
```

<그림 6> 악성 코드가 요구하는 상위 10개 권한

10개의 권한 중 안드로이드 공식 문서에서 위험 권한으로 분류된 권한은 READ_PHONE_STATE, READ_SMS, WRITE_SMS, ACCESS_COARSE_LOCATION, WRITE_EXTERNAL_STORAGE 이렇게 5개이다. 특히 기본적인 INTERNET 권한을 제외한 READ_PHONE_STATE는 가장 높았는데 이는 SIM 하드웨어 ID, 수신전화의 전화번호 등 사용자에게 민감한 정보에 광범위하게 액세스할 수 있는 권한이다. 악성 코드에서는 일

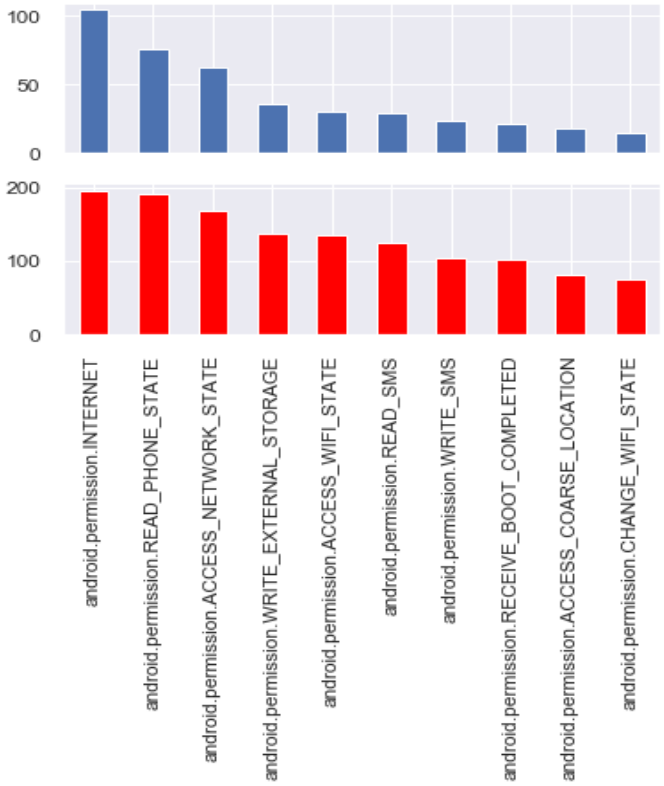
반적으로 어플리케이션의 기능과 상관없이 사용자의 정보에 접근할 수 있는 권한을 요구하는 비율이 많다는 것을 알 수 있다.

```
In [6]: # 정상 샘플에 사용되는 상위 10개의 권한
        # Benign
        pd.Series.sort_values(df[df.type==0].sum(axis=0), ascending=False)[:10]
```

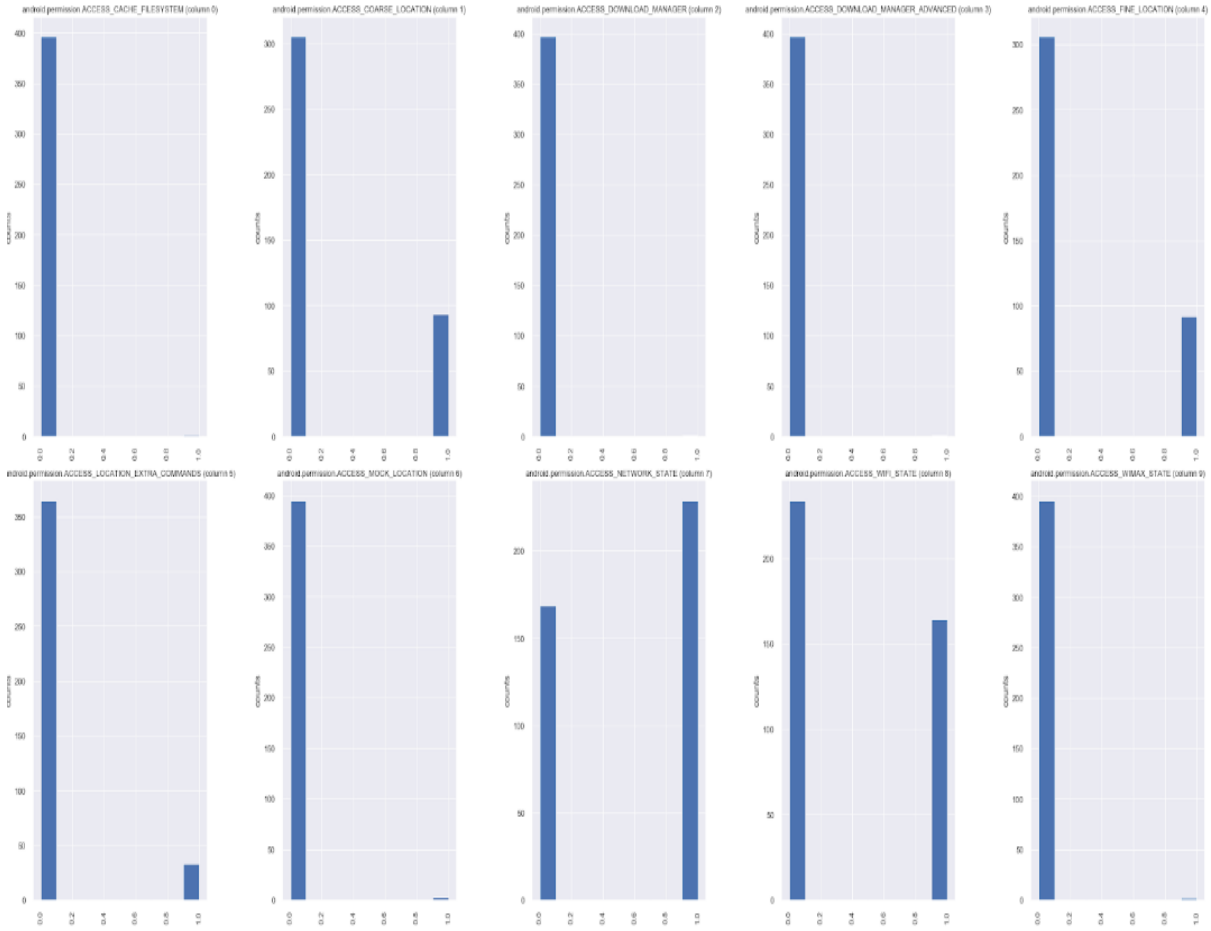
```
Out[6]: android.permission.INTERNET                104
        android.permission.WRITE_EXTERNAL_STORAGE   76
        android.permission.ACCESS_NETWORK_STATE     62
        android.permission.WAKE_LOCK                36
        android.permission.RECEIVE_BOOT_COMPLETED   30
        android.permission.ACCESS_WIFI_STATE        29
        android.permission.READ_PHONE_STATE         24
        android.permission.VIBRATE                  21
        android.permission.ACCESS_FINE_LOCATION     18
        android.permission.READ_EXTERNAL_STORAGE    15
        dtype: int64
```

<그림 7> 정상 코드가 요구하는 상위 10개 권한

다음은 정상 코드로 분류된 어플리케이션에서 요구하는 상위 10개의 권한이다. 정상 코드 또한 위험 권한으로 분류된 권한이 4개이다. 악성 코드 5개와 비교하면 하나 적은 것을 볼 수 있다. 또한 위험 권한을 요구하는 수가 악성 코드에 비해 상대적으로 적은 것을 볼 수 있다.



<그림 8> 권한 데이터 수 분포 그래프



<그림 9> 각 권한 데이터 비교 그래프

이러한 데이터 차이는 그림 7 과 그림 8 의 시각 자료를 통해 보다 정확하게 비교할 수 있다. 그림 7 은 파란색은 정상 코드, 빨간색은 악성 코드가 요구하는 권한 수를 나타내고 있다. 또한 그림 8 은 오른쪽은 정상 코드, 왼쪽은 정상 코드 권한 수이다. 한눈에 볼 수 있듯이 악성 코드가 정상 코드보다 더 많은 권한과 위험 권한을 요청하는 것을 확인할 수 있다.

안드로이드 공식문서에서 명시된 위험 권한을 요구한다고 해서 무조건 악성 코드라고 단정지을 수는 없다. 왜냐하면 어플리케이션에서 기능을 사용할 때 실제로 필요해서 요구할 수도 있기 때문이다. 그러나 개인 정보에 접근하거나 다른 이득을 위해서 이러한 권한이 필요한 것도 사실이다. 따라서 이러한 정상 코드와 악성 코드의 특징을 참고하여 기계학습을 할 예정이다.

데이터의 값 분석 – 네트워크 패킷

네트워크 데이터는 어플리케이션을 실행시키면서 나오는 네트워크 데이터를 수집한 데이터이다. 앞서 제시한 권한과는 다르게 네트워크 트래픽을 통해 악성 어플리케이션을 탐지하는데 목적을 두고 있다. 각 요소들은 네트워크에서 이루어지는 데이터 정보로 tcp 패킷 수, 오고 가는 데이터의 양, dns의 쿼리 시간 등으로 구성되어 있으며 악성 어플리케이션이면 malicious, 정상적인 어플리케이션의 네트워크 정보는 benign으로 구별된다. 그림 9에서 확인할 수 있듯이 총 데이터 속성은 16개로 구별되며 타겟은 type이다.

```
In [28]: data = pd.read_csv("./input/datasets_android2.csv", sep=";")
         data.head()
```

Out[28]:

	name	tcp_packets	dist_port_tcp	external_ips	vulume_bytes	udp_packets	tcp_urg_packe
0	AntiVirus	36	6	3	3911	0	
1	AntiVirus	117	0	9	23514	0	
2	AntiVirus	196	0	6	24151	0	
3	AntiVirus	6	0	1	889	0	
4	AntiVirus	6	0	1	882	0	

```
In [29]: data.columns
```

```
Out[29]: Index(['name', 'tcp_packets', 'dist_port_tcp', 'external_ips', 'vulume_bytes',
                'udp_packets', 'tcp_urg_packet', 'source_app_packets',
                'remote_app_packets', 'source_app_bytes', 'remote_app_bytes',
                'duracion', 'avg_local_pkt_rate', 'avg_remote_pkt_rate',
                'source_app_packets.1', 'dns_query_times', 'type'],
               dtype='object')
```

```
In [30]: data.shape
```

```
Out[30]: (7845, 17)
```

<그림 10> 네트워크 데이터 속성

데이터 속성들이 의미하는 바는 다음과 같다.

name : 어플리케이션 이름

Tcp_packets : 통신 중 TCP 가 보내고 받는 패킷 수

Dist_prot_tcp : TCP 패킷이 다를 때 TCP 와 다른 총 패킷 수

External_ips : 외부 ip, 응용 프로그램이 통신을 시도한 외부 주소 IP 의 수

Volume_bytes : 바이트 수, 응용 프로그램에서 외부 사이트로 보낸 바이트 수

udp_packets : 통신에서 전송된 UDP 패킷의 총 수

source_app_packets : 소스 어플리케이션의 패킷으로 어플리케이션에서 원격 서버로 전송된 패킷 수

remote_app_packets : 원격 어플리케이션 패키지, 외부 소스에서 받은 패키지 수

source_app_bytes : 응용 프로그램 소스의 바이트로 응용 프로그램과 서버 간의 통신량(바이트)

remote_app_bytes : 원격 어플리케이션의 바이트로 서버에서 에뮬레이터까지의 데이터 볼륨(바이트)

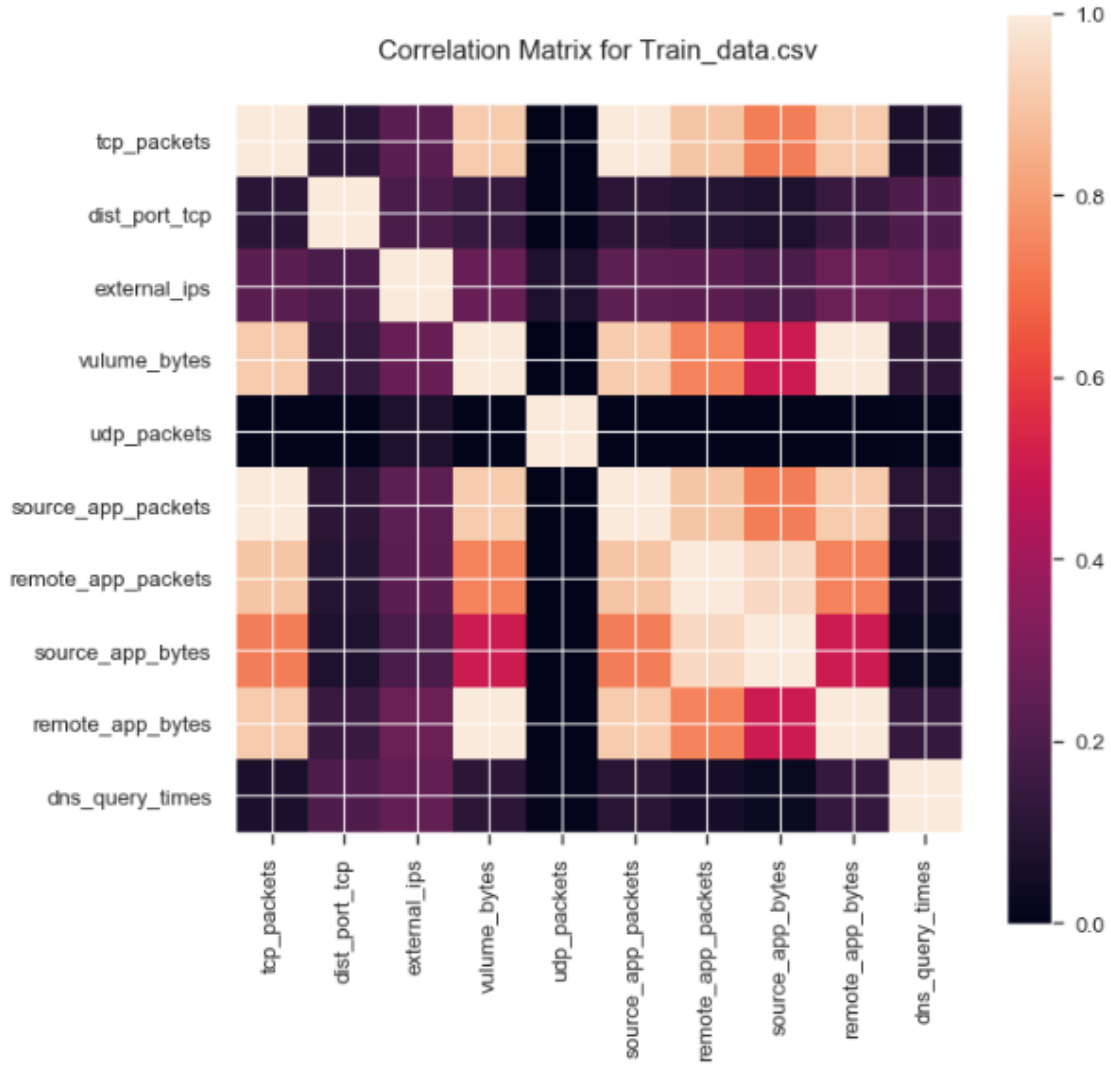
dns_query_times : dns 쿼리 수

type : 악성 및 정상 코드 분류

```
In [31]: data.type.value_counts()
Out[31]: benign       4704
         malicious    3141
         Name: type, dtype: int64

In [32]: data.isna().sum()
Out[32]: name                  0
         tcp_packets           0
         dist_port_tcp         0
         external_ips          0
         vulume_bytes          0
         udp_packets           0
         tcp_urg_packet        0
         source_app_packets    0
         remote_app_packets    0
         source_app_bytes      0
         remote_app_bytes      0
         duracion           7845
         avg_local_pkt_rate 7845
         avg_remote_pkt_rate 7845
         source_app_packets.1  0
         dns_query_times       0
         type                  0
         dtype: int64
```

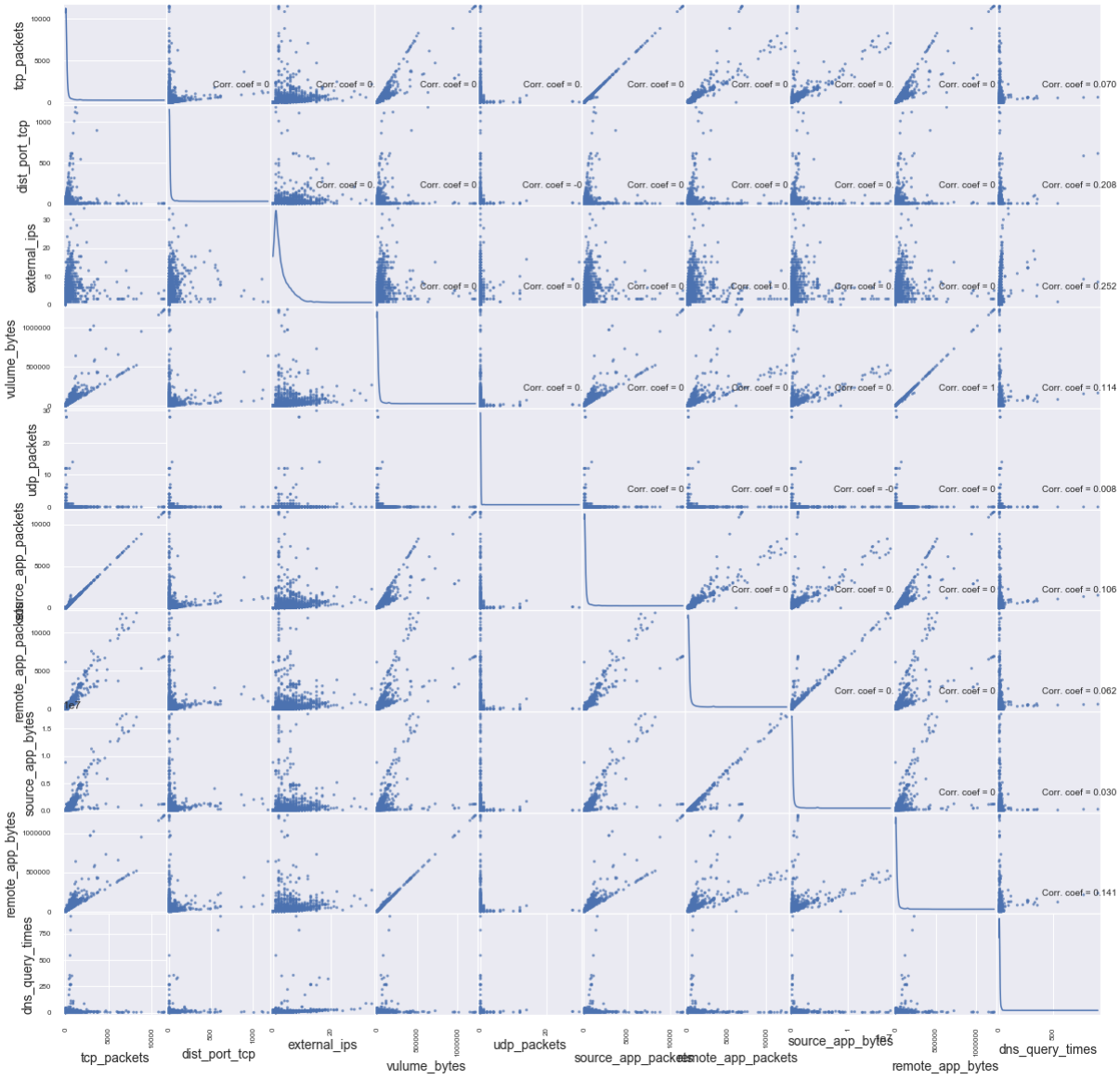
<그림 11> 네트워크 데이터 셋

그림 10 에서 확인할 수 있듯이 데이터셋은 정상적인 어플리케이션 4704 개와 비정상적인 어플리케이션 3141 개에 대해서 구성되어 있으며 null 값으로 채워진 항목이 duration, avg_local_pkt_rate, avg_remote_pkt_rate 총 3 개가 존재한다. 따라서 머신 러닝을 수행하기 위해서 해당 데이터의 전처리 과정이 필요하다고 분석했다.



<그림 12> 상관 관계도 그래프

또한 그림 8 에서의 상관관계도에서 각 항목간의 상관 관계를 분석할 수 있는데 tcp_packets 이 volume_bytes, dup_packets, source_app_packets, source_app_bytes, remote_app_bytes 와 높은 상관관계를 가지는 것으로 분석된다. 또한 위의 각 항목들은 또 제각기 서로 밀접한 관계를 가지는 것으로 보인다.

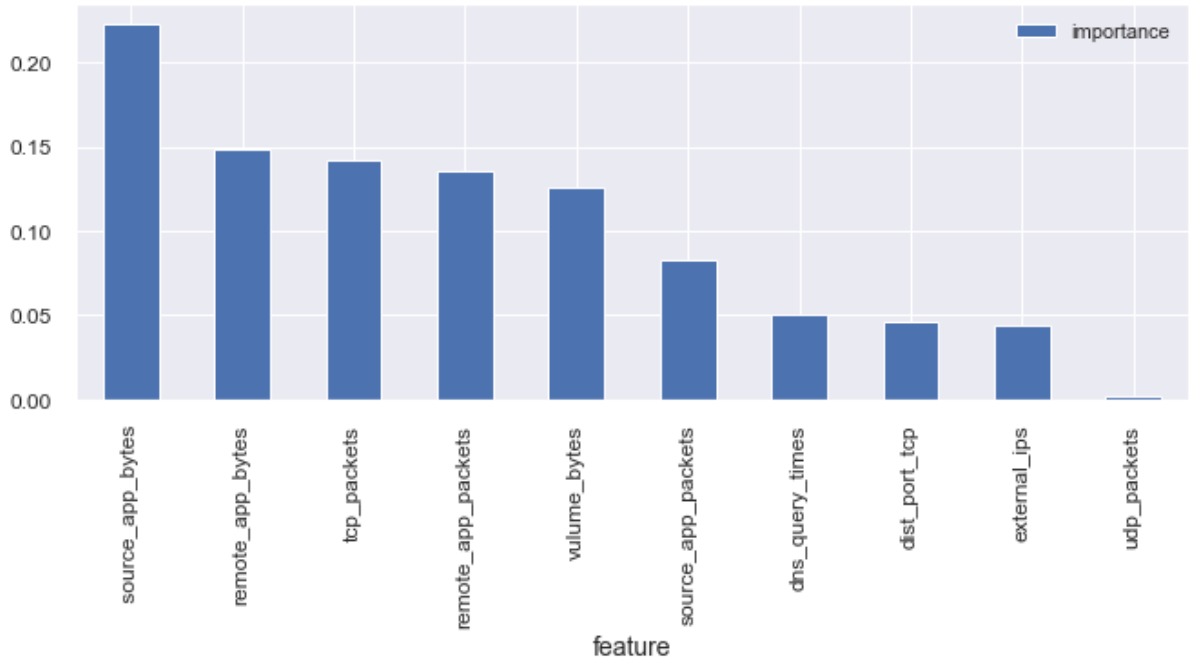


<그림 13> 산점도 그래프

```
In [38]: data=data[data.tcp_packets<20000].copy()
         data=data[data.dist_port_tcp<1400].copy()
         data=data[data.external_ips<35].copy()
         data=data[data.vulume_bytes<2000000].copy()
         data=data[data.udp_packets<40].copy()
         data=data[data.remote_app_packets<15000].copy()
```

<그림 14> 데이터 조정 코드

그림 12 에서는 각 그래프들에 대한 산점도를 분석하여 각 데이터들이 어느 수치가 유효한 데이터 값을 가지는지 파악하여 노이즈 제거를 위해 그림 13 의 작업을 수행하였다.



<그림 15> 중요도 그래프

마지막으로 중요도를 분석하여 중요하지 않은 항목들은 분석할 데이터에서 제외하여 진행하였다. 그림 14 에서 확인할 수 있듯이 Source_app_bytes, remote_app_bytes 등의 10 가지 특징들만 머신 러닝이 훈련하도록 나머지 중요하지 않은 데이터들은 모두 drop 을 하여 삭제하여 진행했다.

권한 데이터 코드

데이터 전처리 – 권한

권한을 갖고 있느냐 없느냐로 판단하기 때문에 데이터 스케일링 과정이 필요하지는 않는다. 다만 의미가 없는 권한들이 많기 때문에 정상적인 데이터와 비정상적인 데이터 모두 권한을 요청하지 않는 특징들은 모델 훈련 과정에서 필요 하지 않기 때문에 제외시키는 과정을 수행한다.

모델 훈련 및 평가 – 권한

훈련 세트와 테스트 셋을 분리하여 모델 훈련을 진행한다.

```
In [8]: X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, 1:330], df['type'], test_size=0.20, random_state=42)
```

<그림 16> 훈련 및 테스트 세트 분리 코드

KNN 알고리즘, 결정트리, 랜덤포레스트, GaussianNB, SVM 모델을 사용하여 K-fold 로 검증하는 방법으로 진행되었다. 이때 K 는 10 으로 10 번 나누어 진행하여 평균값을 도출해낸다.

- KNN

```
k_fold = KFold(n_splits=10, shuffle=True, random_state=0)
clf = KNeighborsClassifier(n_neighbors = 13)
scoring = 'accuracy'
score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
print(score)
```

```
[0.96875    0.9375     0.90625    0.9375     0.84375    0.90625
 0.9375     0.90625    0.93548387 0.93548387]
```

```
In [24]: # kNN Score
         round(np.mean(score)*100, 2)
Out[24]: 92.15
```

<그림 17> KNN 모델 권한 데이터

- 결정트리

```
In [25]: clf = DecisionTreeClassifier()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
```

```
[0.84375    0.9375     0.96875    0.9375     0.96875    0.90625
 0.90625    0.90625    0.90322581 0.96774194]
```

```
In [26]: # decision tree Score
         round(np.mean(score)*100, 2)
Out[26]: 92.46
```

<그림 18> 결정트리 모델 권한 데이터

- 랜덤 포레스트

```
In [27]: clf = RandomForestClassifier(n_estimators=13)
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)

         [1.         0.90625    0.9375     0.9375     0.9375     0.9375
          0.90625    0.84375    0.93548387 0.96774194]

In [28]: # Random Forest Score
         round(np.mean(score)*100, 2)

Out[28]: 93.09
```

<그림 19> 랜덤 포레스트 모델 권한 데이터

- GaussianNB

```
In [29]: clf = GaussianNB()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)

         [0.875      0.875      0.90625    0.8125     0.90625    0.875
          0.8125     0.75       0.90322581 0.90322581]

In [30]: # Naive Bayes Score
         round(np.mean(score)*100, 2)

Out[30]: 86.19
```

<그림 20> GaussianNB 모델 권한 데이터

- SVM

SVM 모델은 하이퍼 파라미터(C 와 gamma)를 조절하여 최적의 모델을 찾는 방식이 필요하다. 다음은 각각 파라미터를 조절해서 얻은 값이다.

```
c = [10, 1, 0.1]
g = [10, 1, 0.1, 0.001]
for i in c:
    for j in g:
        svc = SVC(gamma=j, C=i, probability=True)
        scoring = 'accuracy'
        score = cross_val_score(svc, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
        print("C = ", i ,", gamma = ", j ," / Accuracy : ", round(np.mean(score)*100,2))

C =  10 , gamma =  10  / Accuracy :  77.96
C =  10 , gamma =  1  / Accuracy :  89.03
C =  10 , gamma =  0.1  / Accuracy :  92.47
C =  10 , gamma =  0.001  / Accuracy :  93.09
C =  1 , gamma =  10  / Accuracy :  77.96
C =  1 , gamma =  1  / Accuracy :  88.08
C =  1 , gamma =  0.1  / Accuracy :  92.78
C =  1 , gamma =  0.001  / Accuracy :  83.94
C =  0.1 , gamma =  10  / Accuracy :  68.56
C =  0.1 , gamma =  1  / Accuracy :  80.52
C =  0.1 , gamma =  0.1  / Accuracy :  91.22
C =  0.1 , gamma =  0.001  / Accuracy :  51.85
```

<그림 21> SVM 모델 권한 데이터

C 와 gamma 값에 따라 확률이 달라지는 것을 볼 수 있다. C 가 10 일 때 gamma 가 0.001 일 때 확률이 93.09 로 가장 높은 것을 확인할 수 있다.

5개의 모델 중 랜덤포레스트와 SVM이 93.09%로 가장 정확한 결과를 산출했다.

테스트 환경 분석 – 권한

데이터를 분석하는 과정에서 사용된 모델의 방식은 랜덤 포레스트이며 10번의 교차 검증을 수행하여 훈련하였다. 하지만 k-fold로 교차 검증을 하는 방법은 적은 데이터셋으로도 더욱 정확하게 사용할 수 있는 장점이 있지만 시간이 더 오래 걸린다는 단점이 존재한다. 막대한 데이터를 다루는 서버에서는 이러한 방법을 적용했을 때 어떤 문제점이 있을지 분석한다.

```
round(np.mean(score)*100, 2)
```

executed in 47ms, finished 16:18:38 2020-06-15

```
[0.84375    0.9375     0.96875    0.9375     0.9375     0.875
 0.875      0.90625    0.90322581 0.96774194]
```

Out[7]: 91.52

In [8]:
```
clf = RandomForestClassifier(n_estimators=13)
scoring = 'accuracy'
score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
print(score)
# Random Forest Score
round(np.mean(score)*100, 2)
```

executed in 204ms, finished 16:18:38 2020-06-15

```
[1.         0.9375     0.96875    0.9375     0.84375    0.9375
 0.90625    0.84375    0.93548387 0.96774194]
```

Out[8]: 92.78

In [9]:
```
clf = GaussianNB()
scoring = 'accuracy'
score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
print(score)
# Naive Bayes Score
round(np.mean(score)*100, 2)
```

executed in 62ms, finished 16:19:36 2020-06-15

```
[0.875      0.875      0.90625    0.8125     0.90625    0.875
 0.8125     0.75       0.90322581 0.90322581]
```

Out[9]: 86.19

In [11]:
```
clf = SVC()
scoring = 'accuracy'
score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
print(score)
round(np.mean(score)*100,2)
```

executed in 165ms, finished 16:20:00 2020-06-15

```
[0.9375     0.90625    0.96875    0.9375     0.90625    0.90625
 0.90625    0.875      0.93548387 0.96774194]
```

<그림 22> 모델 별 실행 시간 비교 – 권한

위의 코드는 executed in 0ms를 통해 각 모델이 걸린 시간을 알 수 있다. 이를 통해 더 큰 데이터를 다룰 때 걸리는 시간을 분석한다. 권한 데이터셋의 경우 약400개의 샘플을 갖고 수행하였으며 랜덤포레스트를 사용하였을 때의 걸린 시간은 205ms이다. 그러나 의외로 교차검증을 사용하거나 사용하지 않고 일반적인 검증 방법을 사용했을 때와 별로 차이가 나지 않았다. 따라서 교차검증을 사용한 모델들끼리의 비교가 더 의미가 있다고

판단했다. 아래의 결과 값은 분석 결과와 다른 환경에서 실행되었기 때문에 미세한 확률의 오차가 있다.

모델	정확도	실행 시간
KNN 알고리즘	92.15%	168ms
결정트리	91.52%	47ms
랜덤 포레스트	93.09%	204ms
GaussianNB	86.19%	62ms
SVM	93.09%	165ms

샘플 데이터가 400 개밖에 되지 않기 때문에 이보다 더 많은 양의 데이터를 분석하게 되었을 때는 훨씬 더 오랜 시간이 소요될 것으로 예상된다. 따라서 현재는 단순히 최고의 정확도를 보여주는 랜덤포레스트 또는 SVM 모델이 선택되었지만 만약 이보다 더 많은 양의 데이터를 수행하게 된다면 정확도 차이도 별로 나지 않고 실행 시간도 1/5 밖에 소요되지 않는 결정트리 모델을 사용하는 것이 현명한 판단으로 분석된다.

네트워크 데이터 코드

데이터 전처리 – 네트워크 패킷

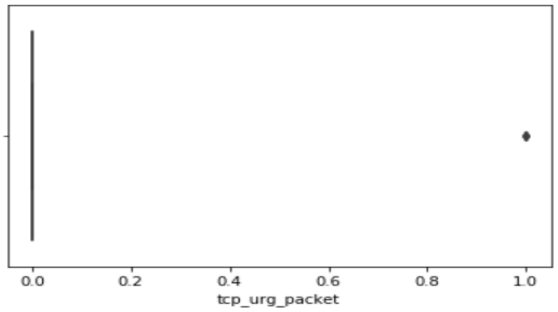
앞서 네트워크 데이터 분석에 언급했듯이 데이터 속성은 16개이지만 훈련 모델에 필요 없다고 판단되는 속성은 제거하여 전처리과정을 거쳤다.

```
In [16]:  # 데이터 전처리 과정
          # data에서 null인 부분
          data.isna().sum()

Out[16]:  name                    0
          tcp_packets             0
          dist_port_tcp           0
          external_ips            0
          vulume_bytes            0
          udp_packets             0
          tcp_urg_packet          0
          source_app_packets      0
          remote_app_packets      0
          source_app_bytes        0
          remote_app_bytes        0
          duracion             7845
          avg_local_pkt_rate   7845
          avg_remote_pkt_rate  7845
          source_app_packets.1    0
          dns_query_times         0
          type                    0
          dtype: int64

In [17]:  # durcaion, avg_local_pkt_rate, avg_remote_pkt_rate 제거
          data = data.drop(['duracion','avg_local_pkt_rate','avg_remote_pkt_rate'], axis=1).c
```

<그림 23> 데이터 속성 제거

duracion, avg_local_pkt_rate, avg_remote_pkt_rate는 모두 값이 0이기 때문에 훈련 데이터에 필요없다고 판단되어 제거하였다.

```
In [21]: sns.boxplot(data.tcp_urg_packet)
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x2543ada5c88>
```




```
In [22]: # data.tcp_urg_packet이 0이상(위의 그래프에 나와있듯이 1)인 것은 2개이다.
         # 이는 향후 분석에 거의 의미가 없으므로 drop해준다.
         data.loc[data.tcp_urg_packet > 0].shape[0]
Out[22]: 2

In [23]: data = data.drop(columns=["tcp_urg_packet"], axis=1).copy()
         data.shape
Out[23]: (7845, 13)
```

<그림 24> tcp_urg_packet 시각화

"data.loc[data.tcp_urg_packet > 0].shape[0]" 코드를 통해 7845개의 데이터 중 2개(값은 1)을 제외한 나머지 데이터 값들은 0이므로 분석에 의미가 없다고 생각하여 제거하였다.

모델 훈련 및 평가 – 네트워크 패킷

훈련 세트와 테스트 세트를 분리하여 모델 훈련을 진행한다.

```
In [43]: X_train, X_test, y_train, y_test = train_test_split(scaledData.iloc[:,0:10], data.type.astype("str"), test_size=0.25, random_state=45)
```

<그림 25> 훈련 및 테스트 세트 분리 코드

앞서 권한에서 진행한 것과 같은 방법으로 KNN 알고리즘, 결정트리, 랜덤포레스트, GaussianNB, SVM 모델을 사용하여 K-fold로 검증하는 방법으로 진행되었다. 이때 K는 10으로 10번 나누어 진행하여 평균값을 도출해낸다.

- KNN

```
k_fold = KFold(n_splits=10, shuffle=True, random_state=0)
clf = KNeighborsClassifier(n_neighbors = 13)
scoring = 'accuracy'
score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
print(score)
```

```
[0.83843537 0.83843537 0.82142857 0.8537415  0.85008518 0.82964225
 0.85178876 0.85008518 0.82112436 0.85008518]
```

```
In [68]: # kNN Score
         round(np.mean(score)*100, 2)
Out[68]: 84.05
```

<그림 26> KNN 모델 네트워크 패킷

- 결정트리

```
In [69]: clf = DecisionTreeClassifier()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
```

```
[0.87244898 0.86734694 0.86904762 0.84013605 0.86541738 0.89097104
 0.85860307 0.86882453 0.86882453 0.84156729]
```

```
In [70]: # decision tree Score
         round(np.mean(score)*100, 2)
Out[70]: 86.43
```

<그림 27> 결정트리 모델 네트워크 패킷

- 랜덤포레스트

```
In [71]: clf = RandomForestClassifier(n_estimators=13)
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
```

```
[0.89285714 0.88435374 0.86904762 0.89285714 0.88245315 0.88756388
 0.9011925  0.90800681 0.87052811 0.87563884]
```

```
In [72]: # Random Forest Score
         round(np.mean(score)*100, 2)
Out[72]: 88.64
```

<그림 28> 랜덤포레스트 모델 네트워크 패킷

- GaussianNB

```
In [73]: clf = GaussianNB()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
```

```
[0.45918367 0.44897959 0.46088435 0.42687075 0.46848382 0.45655877
 0.44633731 0.47189097 0.46678024 0.45996593]
```

```
In [74]: # Naive Bayes Score
         round(np.mean(score)*100, 2)
Out[74]: 45.66
```

<그림 29> GaussianNB 모델 네트워크 패킷

- SVM

앞서 C와 gamma를 조절한 것과 같이 네트워크 데이터 SVM 모델 또한 조절하는 방식이 필요하다.

```
c = [10, 1, 0.1]
g = [10, 1, 0.1, 0.001]
for i in c:
    for j in g:
        svc = SVC(gamma=j, C=i, probability=True)
        scoring = 'accuracy'
        score = cross_val_score(svc, X_train, y_train, cv=k_fold, n_jobs=1, scoring
        print("C = ", i ,", gamma = ", j ," / Accuracy : ", round(np.mean(score)*10
```

```
C =  10 , gamma =  10  / Accuracy :  84.12
C =  10 , gamma =  1  / Accuracy :  83.4
C =  10 , gamma =  0.1  / Accuracy :  78.67
C =  10 , gamma =  0.001  / Accuracy :  62.56
C =  1 , gamma =  10  / Accuracy :  83.42
C =  1 , gamma =  1  / Accuracy :  79.62
C =  1 , gamma =  0.1  / Accuracy :  75.78
C =  1 , gamma =  0.001  / Accuracy :  60.38
C =  0.1 , gamma =  10  / Accuracy :  77.0
C =  0.1 , gamma =  1  / Accuracy :  75.01
C =  0.1 , gamma =  0.1  / Accuracy :  66.45
C =  0.1 , gamma =  0.001  / Accuracy :  60.11
```

<그림 30> SVM 모델 네트워크 패킷

C와 gamma값을 조절하면서 많게는 84.12% 적게는 60.11%까지 정확도가 변동하는 것을 볼 수 있다. 이 때, C가 10, gamma가 10일 때 84.12%로 가장 정확도가 높은 것을 확인할 수 있다.

5개의 모델들 중 랜덤포레스트가 88.64%로 가장 정확한 결과를 산출해냈다.

테스트 환경 분석 – 네트워크 패킷

```
Out[19]: 84.39

In [20]: clf = DecisionTreeClassifier()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
         # decision tree Score
         round(np.mean(score)*100, 2)
         executed in 252ms, finished 16:29:24 2020-06-15

         [0.8792517  0.8622449  0.83503401 0.84353741 0.87393526 0.89097104
          0.86882453 0.88074957 0.879046   0.85008518]

Out[20]: 86.64

In [21]: clf = RandomForestClassifier(n_estimators=13)
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
         # Random Forest Score
         round(np.mean(score)*100, 2)
         executed in 793ms, finished 16:29:30 2020-06-15

         [0.88605442 0.88265306 0.88435374 0.88095238 0.88756388 0.89097104
          0.88586031 0.89267462 0.88245315 0.88415673]

Out[21]: 88.58

In [22]: clf = GaussianNB()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
         # Naive Bayes Score
         round(np.mean(score)*100, 2)
         executed in 81ms, finished 16:29:34 2020-06-15

         [0.43877551 0.43027211 0.46258503 0.41156463 0.4548552  0.44293015
          0.44122658 0.45996593 0.46337308 0.44974446]

Out[22]: 44.55

In [23]: clf = SVC()
         scoring = 'accuracy'
         score = cross_val_score(clf, X_train, y_train, cv=k_fold, n_jobs=1, scoring=scoring)
         print(score)
         round(np.mean(score)*100,2)
         executed in 7.15s, finished 16:29:47 2020-06-15

         [0.59693878 0.6037415  0.6037415  0.61394558 0.58432709 0.60988075
          0.5911414  0.60136286 0.58773424 0.61499148]
```

<그림 31> 모델 별 실행 시간 비교 - 네트워크

권한과 같은 방법으로 모델 별 실행시간과 정확도를 고려하여 분석을 진행하였다.

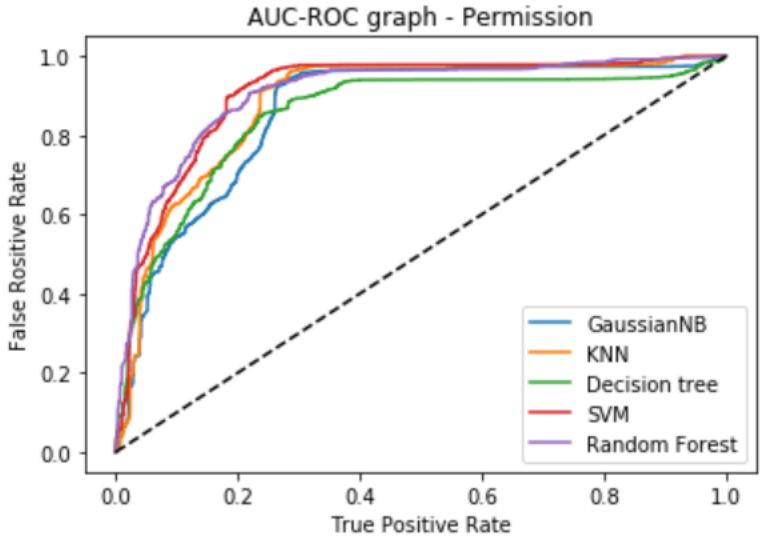
모델	정확도	실행 시간
KNN 알고리즘	84.39%	518ms
결정트리	86.64%	252ms
랜덤 포레스트	88.58%	793ms
GaussianNB	44.55%	81ms
SVM	84.12%	7.15s

네트워크를 분석한 데이터는 8 천개의 샘플과 분석에 필요한 속성 값도 권한에 비해 많기 때문에 전체적으로 권한을 분석하는 시간보다 오래 걸렸다. 그러나 각 모델을

확연한 차이를 보이는데 우선 가장 높은 정확도를 가지는 랜덤포레스트는 795ms 로 역시 권한 데이터를 분석했던것과 같이 정확도는 미세하게 우세하지만 걸리는 시간은 결정트리보다 더 느리게 진행되었다. 가장 짧은 시간을 가지는 GaussianNB 모델은 81ms 로 매우 빠른 속도로 분석이 진행되었지만 44.55%라는 매우 낮은 정확도를 가졌기 때문에 모델로서의 성능은 떨어진다고 분석된다. SVM 또한 84.12%이라는 비교적 낮은 정확도와 8 천개의 샘플을 대상으로 분석하는 시간이 7 초 이상 걸렸기 때문에 이 역시 매우 방대한 데이터를 대상으로 진행된다면 고려할 모델에서 제외될 것이다.

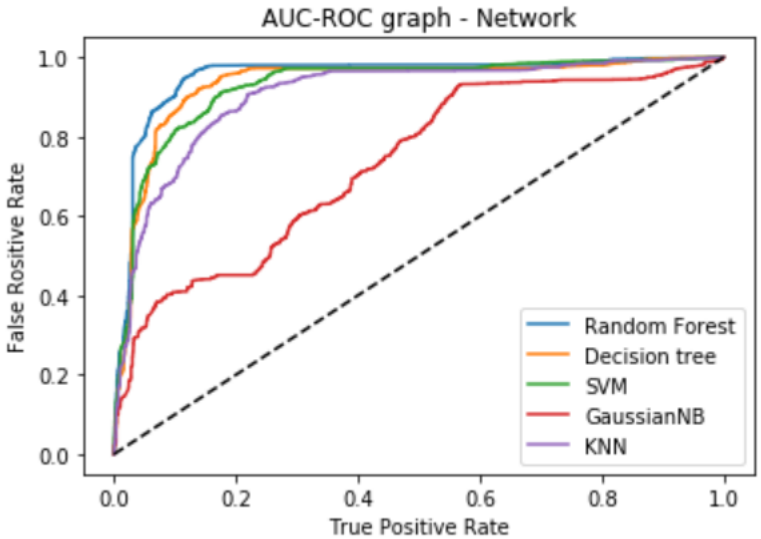
실제 많은 데이터를 대상으로 분석할 때 모델을 선택해야 할 때 실행 시간이 우선적으로 고려된다면 결정트리, 정확도가 우선적으로 고려된다면 랜덤 포레스트 모델을 사용하는 것이 좋다고 판단된다.

AUROC 그래프



<그림 32> 권한 AUR 그래프

그림 32 은 권한에서 각 모델의 AUR 그래프이다. 각 모델의 성능은 앞서 보았다 싶이 10 개 성능의 평균을 낸 것이다. 모든 모델의 성능 대다수가 90% 이상으로 기계학습 모델에 뛰어난 성능을 보였다. 이는 데이터를 제대로 전처리 한 후 패턴을 추출하여 기계학습 알고리즘 모델에 잘 적합 되었다는 것을 의미한다.



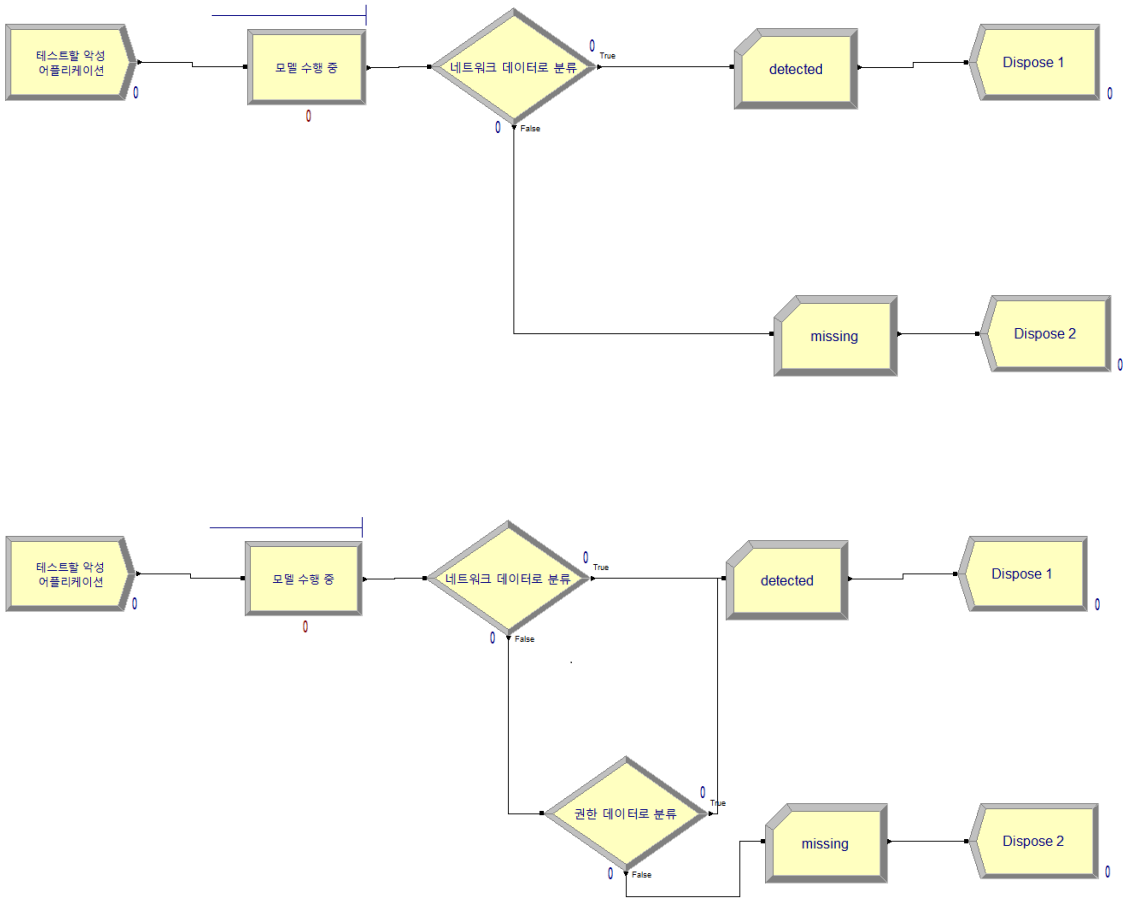
<그림 33> 네트워크 AUR 그래프

그림 33 은 네트워크에서 각 모델의 AUR 그래프이다. 앞서 GaussianNB 은 44.55%로 낮은 성능을 보였다. 따라서 그래프에서도 낮은 면적을 보여주고 있다. 나머지 모델들은 85%정도로 대다수 면적이 비슷하여 비슷한 성능을 가지고 있음을 알 수 있다. 엄청 뛰어난 성능은 아니지만 비교적 좋은 성능을 보여주고 있다.

아레나 분석

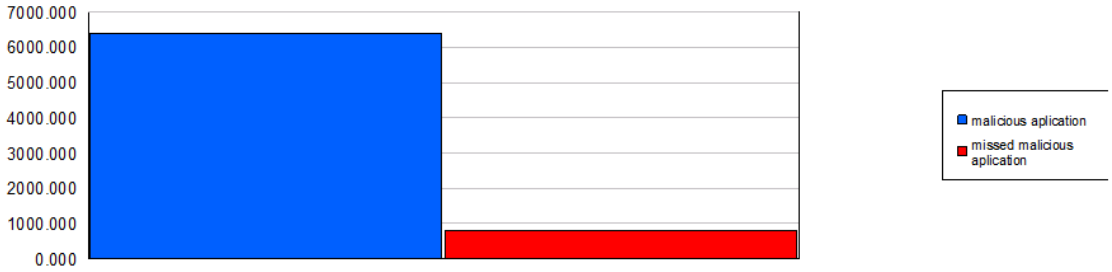
분석한 두 모델은 서로 다른 데이터를 갖고 분석하기 때문에 만약 한 어플리케이션에서 권한과 네트워크의 정보 모두를 획득할 수 있다면 두 가지 방법을 모두 사용하여 어플리케이션에 대한 머신 러닝을 수행할 수 있을 것이다. 따라서 이러한 시뮬레이션을 진행하기 위하여 아레나 시뮬레이션을 사용하여 구체적으로 어떤 결과를 가지는지 분석하였다.

Create에서는 비정상적인 어플리케이션을 생성하며 네트워크 통신 데이터를 갖고 이것이 악성 어플리케이션인지 아닌지 판단한다. 판단하기 위한 기준으로는 가장 높은 정확도를 가지는 랜덤 포레스트를 사용하였으며 그 정확도는 88.58이다. 2번째 decide에서는 권한을 갖고 판단을 하는 머신러닝을 가정하여 진행하였으며 사용된 모델은 역시 가장 높은 정확도를 가지는 랜덤 포레스트 방식을 사용하였으며 정확도는 92.78이다. 비교를 위해서 네트워크만 분석한 결과를 가지는 모델과 네트워크와 권한 두 가지 방법을 모두 사용하여 분석하였다.

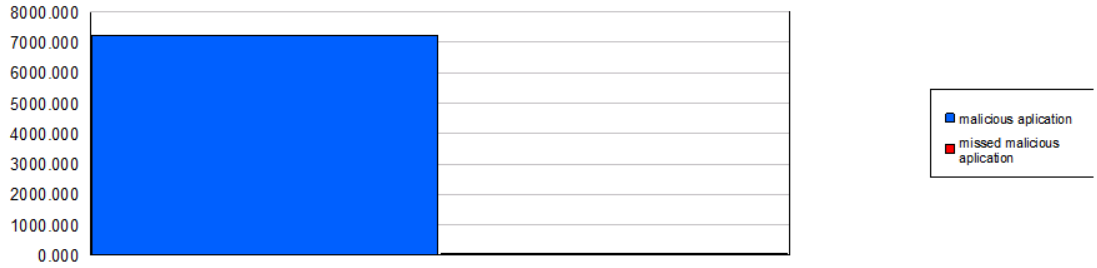


<그림 34> 아레나 시뮬레이션 모델

Count	Average	Half Width	Minimum Average	Maximum Average
malicious aplication	6405.00	108.60	6355.00	6436.00
missed malicious aplication	808.33	51.18	789.00	830.00



Count	Average	Half Width	Minimum Average	Maximum Average
malicious aplication	7240.33	107.99	7201.00	7287.00
missed malicious aplication	54.6667	10.04	51.0000	59.0000



<그림 35> 아레나 결과 그래프 비교

위의 그래프는 네트워크 데이터만 갖고 분석한 결과이며 아래의 그래프는 두 가지 방법을 모두 사용하여 분석한 결과이다. 한 눈에도 두 가지 방법을 모두 사용하여 수행한 결과값이 훨씬 더 성능이 좋은 것을 확인할 수 있었다. 그러나 이것은 단순히 네트워크에서 분석한 머신 러닝에서의 결과값에서 다시 권한 분석을 수행한 것이기 때문에 실제로 사용했을 때는 네트워크에서 분석할 때 탐지하지 못한 데이터는 권한 데이터에서 어떤 성능을 나타낼 지는 실제로 수행해보기 전까지는 확신할 수 없다. 따라서 정확한 실험 환경을 구축하기 위해서는 한 어플리케이션에서 권한 데이터와 네트워크 관제 데이터 두 가지 모두를 가지고 있는 확실한 데이터셋들이 필요한데 아쉽게도 그러한 데이터를 가지려면 악성 어플리케이션에 대한 정보를 쉽게 획득할 수 있는 환경을 가져야 하지만 현재 환경에서는 그러지 못해 불가능했다. 그러나 동적 분석과 정적 분석의 데이터를 머신 러닝을 수행할 때 혼합하여 수행한다면 단순히 한가지 분석 방법을 사용했을 때보다 성능이 더 좋게 나올 것이라는 예측은 할 수 있었다.

결론

이번 프로젝트의 목표는 머신러닝을 활용하여 악성코드를 탐지하는 것이다. 프로젝트에서 제안한 방법은 두 가지인데 하나는 권한을 기반으로 나머지 하나는 네트워크 트래픽을 이용하였다. 두 방법 모두 모바일 어플리케이션이라면 가지고 있는 데이터이기 때문에 어떤 어플리케이션이라도 데이터를 추출하여 악성 어플리케이션인지 탐지할 수 있다. 하지만 권한 같은 방식의 경우 정상적인 어플리케이션이 위험 권한 또는 많은 권한을 요구할 수도 있고 네트워크 방식의 경우 정상 네트워크 패킷이 악성 네트워크 비슷할 수도 있다. 이러한 부분들은 예외적인 상황이지만 정확도를 높이기 위해서 고려해봐야한다.

본 프로젝트에서 사용한 모델은 KNN, 결정 트리, 랜덤 포레스트, GaussianNB, SVM 총 5가지 모델을 사용하였다. 권한과 네트워크 모두 5가지 모델들 중 랜덤 포레스트 모델이 각각 93.09%, 88.64%의 가장 높은 정확성을 보였다. 아무래도 다수의 결정트리를 학습하는 앙상블 방법이다 보니 높은 정확성을 보인 것 같다. 걸린 시간은 정확도가 어느 수준까지 높은 모델 중 결정트리 모델이 각각 47ms, 252ms로 가장 낮았다. 따라서 실제 빅데이터를 분석할 때는 정확도와 걸린 시간의 우선 순위를 정하여 각 우선 순위에 맞는 모델을 사용하면 될 것이다.

향후 프로젝트를 더 진행할 기회가 생긴다면 두 가지를 더 보완하고 싶다. 앞서 언급한 것처럼 정상 어플리케이션이 실제로 기능이 많아 위험 권한과 많은 권한을 요구한다면 아마 머신러닝을 통한 결과는 악성 코드로 탐지될 것이다. 이러한 결과를 막기 위해, 데이터 셋을 더욱 분석하여 예외적인 상황에서도 더 정확한 결과를 얻을 수 있도록 보완하고 싶다. 다른 하나는 높은 정확성을 가진 모델도 있었지만 그렇지 않은 모델들도 있었다. 모델들의 정확성을 더욱 높이기 위해 데이터 속성을 바꾸거나 모델들의 알고리즘을 변경 및 추가하고 싶다.

이번 프로젝트를 통해 직접 실습을 해봄으로써 빅데이터와 머신러닝이 어떤 방향 또는 방식으로 작동하는지 이해하는 데 좋은 기회가 되었다.

참고 문헌

1. 캐글 안드로이드 권한 데이터 셋

https://www.kaggle.com/xwolf12/datasetandroidpermissions

2. 캐글 안드로이드 네트워크 데이터 셋

https://www.kaggle.com/xwolf12/network-traffic-android-malware

3. 머신러닝을 이용한 권한 기반 악성코드 탐지

http://www.ndsl.kr/ndsl/search/detail/article/articleSearchResultDetail.do?cn=JAKO201820765436671

4. 안드로이드 SVM 을 통한 악성코드 탐지

https://nmlab.korea.ac.kr/publication/published.papers/2013/2013.12-SVM%20based%20android%20malware%20detection-KCIC2013.pdf