안드로이드 멀웨어 탐지의 지속가능성을 위한 K-Means 기반 분류 모델

정성원, 안석현, 조성제, 김동재, 황영섭

WDSC 2024

2024.08.20

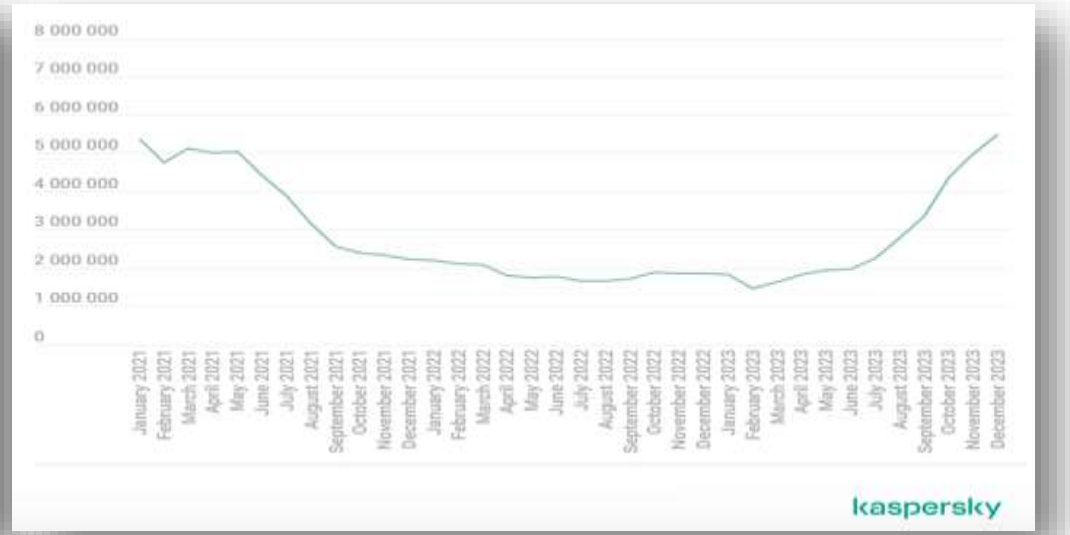
INDEX

01 연구배경 및 연구목적

연구배경 : Google Play & Kaspersky Statistics

Key Google Play Statistics

- Consumers spent $47 billion on Google Play apps and games in 2023
- Over 113 billion apps and games were downloaded on Google Play last year
- 2.61 billion apps and games are available to download on Google Play
- The top grossing app on Google Play in 2023 was Google One, a cloud storage service
- Instagram was the most downloaded app on Google Play last year, with 521 million downloads



- 애플리케이션 산업 정보 플랫폼 Business of Apps의 통계에 의하면, 2023년 한 해 동안만 Google Play Store에서 1,130억 개 이상의 앱이 다운로드 되었다고 밝혔다.
- 2023년에는 보안회사 Kaspersky의 제품이 총 330만 건의 공격을 차단했다고 밝혔다. 특히 2021년 2월까지 감소하던 모바일 기기에 대한 공격이, 2023년에는 2022년 대비 약 50%p 증가하였다.

기계학습 기반 멀웨어 탐지 연구와 K-Means 기법

❖ 모바일 기기를 대상으로 한 공격의 증가에 따라, 기계학습을 기반으로 API(Application Programming Interface) 호출, 권한(Permission) 그리고 인텐트(Intent) 등을 특징정보로 새롭고 다양한 패턴의 공격을 탐지하는 연구가 진행[3-7, 9-10, 14].

❖ ***[3] 조우상, 이호준, 이건용, 조성제, 우진운. API 호출정보를 이용한 안드로이드 악성 앱 탐지에서 난독화가 미치는 영향. Workshop on Dependable and Secure Computing, page 23-27, 2021***

❖ 하지만, 과거의 특징정보로 학습된 모델은 새로운 특징정보를 가진 악성 앱을 탐지하는데 효과적이지 않다는 것이 밝혀졌다.

→ 난독화 (Obfuscation)나 클로킹(Cloaking)등 진화하는 악성 코드 탐지기법의 필요성

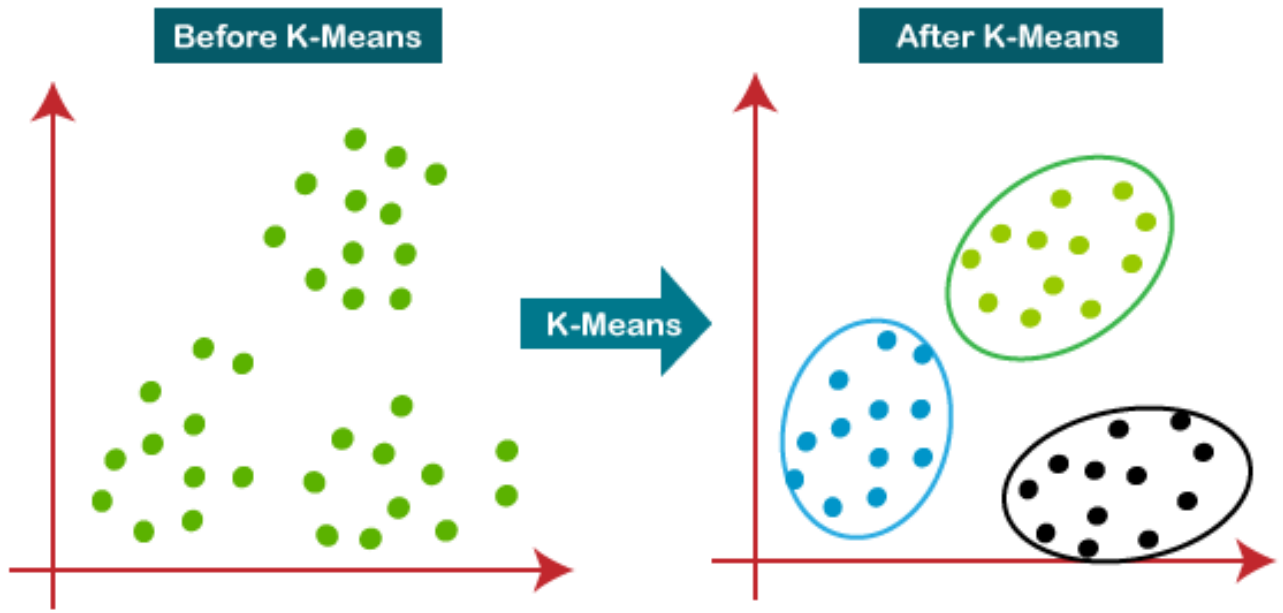
→ 과거 악성 앱에서 자주 사용된 API 호출이 정상 앱에서도 반복적으로 사용

K-Means Clustering을 적용한 분류 모델의 제안

❖ **따라서 본 논문에서는 악성 앱의 지속 메커니즘을 고려하기 위한 K-Means 기반 분류 모델을 제안**

❖ K-Means는 주어진 데이터들을 거리에 따라 유사성을 가지는 k개의 군집으로 분할

- K-Means는 직관적이고 간단한 구현, 대규모 데이터셋에서도 빠른 계산이 가능
- **제안된 모델은 정상/악성 여부와 관계 없이 API 호출 정보가 유사한 앱들을 L2 Distance를 기준으로 그룹화**



Research Questions

❖K-Means 방식으로 학습 및 예측을 했을 때, 정상/악성 앱들은 클러스터에서 어떤 분포를 가지는가?

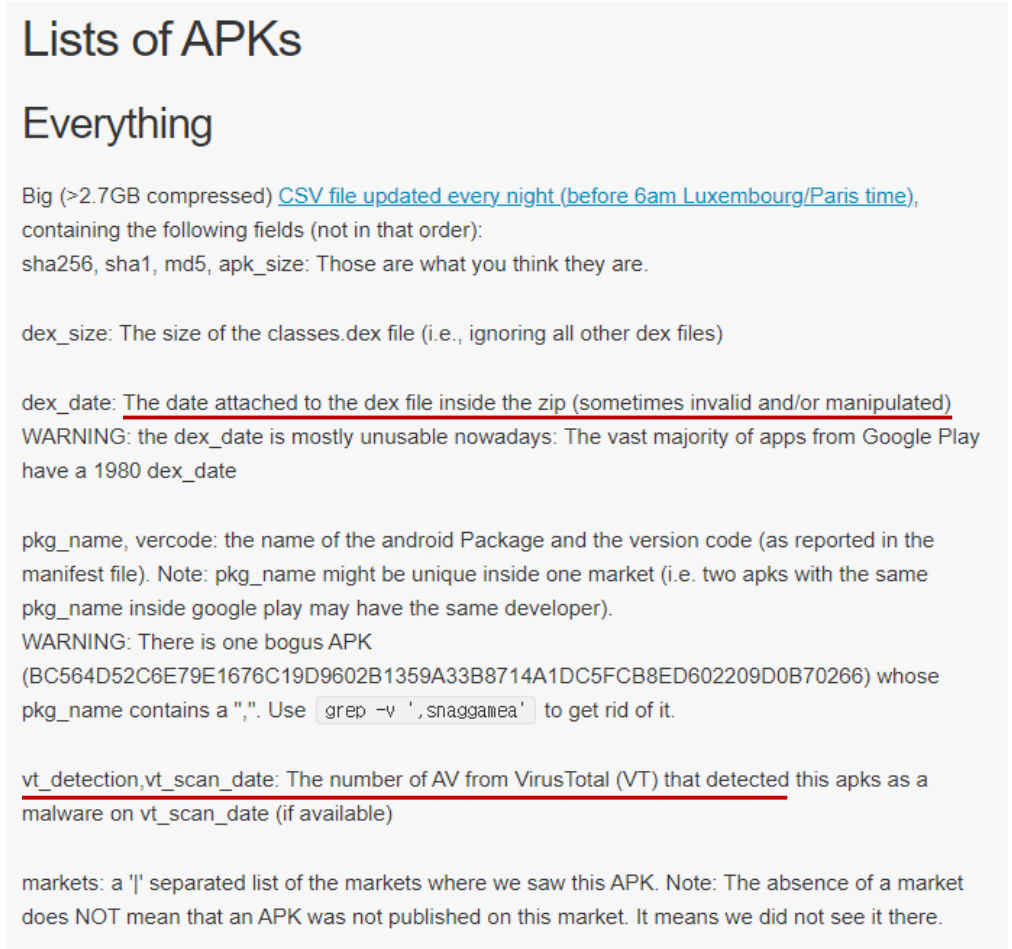
❖K-Means 기반 분류모델의 지속가능성을 어떻게 평가할 수 있는가?

❖기존의 방식과 비교했을 때 얼마나 지속가능성이 높은가?

02 안드로이드 멀웨어 탐지에 관한 기존 연구

안드로이드 멀웨어 탐지에 관한 기존 연구 : Androzoo Datasets

- ***Reference : [13] K. Allix, T. F. Bissyandé, J. Klein and Y. L. Trao n. AndroZoo: Collecting Millions of Android Apps for the Research Community. IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, page 468-471, 2016.***
- ***Reference : [14] AndroZoo. University of Luxembourg. [Online]. Available: https://androzoo.uni.lu/. [Accessed: Jul. 25, 2024]***



Lists of APKs

Everything

Big (>2.7GB compressed) CSV file updated every night (before 6am Luxembourg/Paris time), containing the following fields (not in that order):
sha256, sha1, md5, apk_size: Those are what you think they are.

dex_size: The size of the classes.dex file (i.e., ignoring all other dex files)

dex_date: The date attached to the dex file inside the zip (sometimes invalid and/or manipulated)
WARNING: the dex_date is mostly unusable nowadays: The vast majority of apps from Google Play have a 1980 dex_date

pkg_name, vercode: the name of the android Package and the version code (as reported in the manifest file). Note: pkg_name might be unique inside one market (i.e. two apks with the same pkg_name inside google play may have the same developer).
WARNING: There is one bogus APK (BC564D52C6E79E1676C19D9602B1359A33B8714A1DC5FCB8ED602209D0B70266) whose pkg_name contains a ",". Use `grep -v ',snaggamea'` to get rid of it.

vt_detection,vt_scan_date: The number of AV from VirusTotal (VT) that detected this apks as a malware on vt_scan_date (if available)

markets: a '|' separated list of the markets where we saw this APK. Note: The absence of a market does NOT mean that an APK was not published on this market. It means we did not see it there.

AndroZoo는 Google Play를 비롯한 다양한 출처에서 지속적으로 앱을 수집하고, APK 파일의 분석을 지원하며 현재까지 약 2,400만 개의 APK 데이터를 수집.

연도 분류

dex_date : AndroZoo에서 제공하는 연도 메타 데이터

정상/ 악성 앱 분류

정상 앱 : vt_detection = 0
악성 앱 : vt_detection ≥ 3

안드로이드 멀웨어 탐지에 관한 기존 연구 : How to Extract Effective Feature ?

Reference : [4] J. Jung, J. Park, S. J. Cho, S. Han, M. Park, and H. H. Cho. Feature engineering and evaluation for android malware detection scheme. Journal of Internet Technology, vol. 22, pages 423-440, 2021

❖ 정재민 등은 AndroZoo의 데이터셋과 랜덤 포레스트를 활용하여 Google Android의 공개 정보와 Arp[5] 및 Aafer[6]에 의해 **악성 소프트웨어 탐지에 효과적인 1, 848개의 공식 API 호출을 보고.**

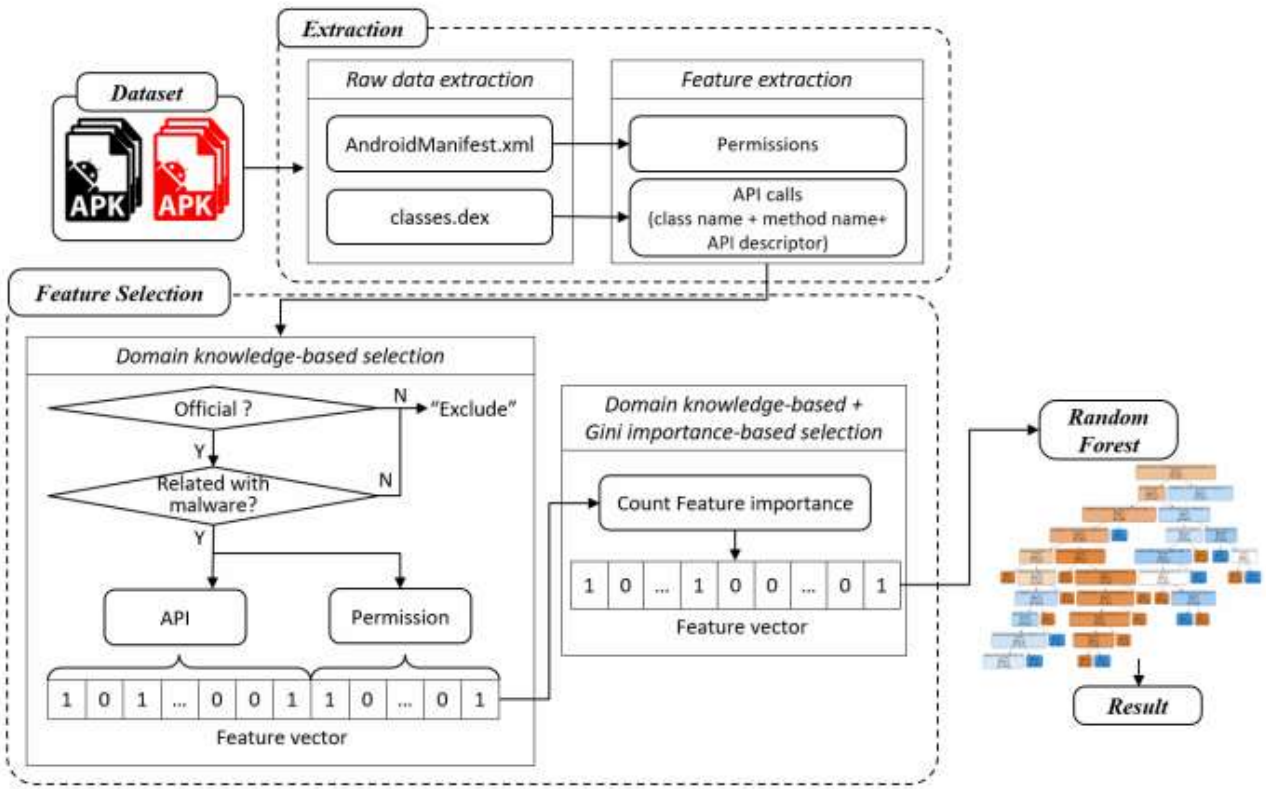


Table 1. A partial list of APIs selected using domain knowledge

	Some APIs (of the selected 1,848 APIs)
User account API	Landroid/accounts/AccountManager;.getAccounts:()[Landroid/accounts/Account;
	Landroid/accounts/AccountManager;.clearPassword:(Landroid/accounts/Account;)V
	Landroid/accounts/AccountManager;.getPassword:(Landroid/accounts/Account;)Ljava/lang/String;
Bluetooth API	Landroid/bluetooth/BluetoothAdapter;.enable:()Z
	Landroid/bluetooth/BluetoothAdapter;.isEnabled:()Z
GPS/Location API	Landroid/location/LocationManager;.addGpsStatusListener:(Landroid/location/GpsStatus$Listener;)Z
	Landroid/location/LocationManager;.requestLocationUpdates:(JFLandroid/location/Criteria;Landroid/app/PendingIntent;)V
Audio API	Landroid/media/AudioRecord;.startRecording:()V
SMS API	Landroid/telephony/SmsManager;.sendDataMessage:(Ljava/lang/String;Ljava/lang/String;S[BLandroid/app/PendingIntent;Landroid/app/PendingIntent;)V
	Landroid/telephony/SmsManager;.sendTextMessage:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;Landroid/app/PendingIntent;)V
Telephony API	Landroid/telephony/TelephonyManager;.getSimSerialNumber:()Ljava/lang/String;
	Landroid/telephony/TelephonyManager;.getDeviceId:()Ljava/lang/String;
Process API	Ljava/lang/Runtime;.exec:(Ljava/lang/String;)Ljava/lang/Process;
Notification API	Landroid/app/NotificationManager;.notify:(Ljava/lang/String;ILandroid/app/Notification;)V
String API	Landroid/lang/StringBuilder;.append:(Ljava/lang/CharSequence;)Ljava/lang/StringBuilder;

Table 2. A partial list of permissions selected using domain knowledge

	Some Permissions (of the selected 79 permissions)
User account Permission	GET_ACCOUNTS
	MANAGE_ACCOUNTS
	GET_ACCOUNTS_PRIVILEGED
Bluetooth Permission	BLUETOOTH
	BLUETOOTH_ADMIN
	BLUETOOTH_PRIVILEGED
Location permission	ACCESS_COARSE_LOCATION
	ACCESS_FINE_LOCATION
Camera permission	CAMERA
SMS permission	SEND_SMS
	READ_SMS
	WRITE_SMS
Phone info permission	READ_PHONE_STATE
	MODIFY_PHONE_STATE
	READ_PHONE_NUMBERS
Billing permission	Vending.BILLING
Launcher permission	com.android.launcher.permission.INSTALL_SHORTCUT
Overlay permission	android.permission.SYSTEM_ALERT_WINDOW

안드로이드 멀웨어 탐지에 관한 기존 연구 : What is Best Classification Model ?

Reference : [7] N. Peiravian, Z. Xingquan. Machine learning for android malware detection using permission and api calls. 2013 IEEE 25th international conference on tools with artificial intelligence. IEEE, pages 300-305, 2013.

❖ Peiravian [7]등은 권한(Permission)과 API 호출을 결합하여 SVM, 결정 나무(Decision Tree), 배깅(Bagging) 세 가지 접근 방식을 사용하여 악성 소프트웨어를 탐지, 그 중 **배깅 방식이 가장 좋은 성능을 제공함을 밝힘.**

Dataset	classifier	Accuracy	Precision	Recall	AUC
Perm	SVM	93.54	92.4	87.5	0.92
API	SVM	95.75	91.7	95.7	0.957
Com+	SVM	96.88	95.7	94.8	0.963
Perm	J48	92.36	89.8	86.6	0.917
API	J48	93.33	89.4	90.3	0.918
Com+	J48	94.46	90.6	92.8	0.936
Perm	Bagging	93.60	92.0	88.2	0.956
API	Bagging	94.89	93.6	90.7	0.986
Com+	Bagging	96.39	94.9	94.1	0.991

TABLE II
BENIGN VS. MALICIOUS DETECTION
{PERM:PERMISSION; API:API CALLS; COM+:PERMISSION+API CALLS}

안드로이드 멀웨어 탐지에 관한 기존 연구 : Sustainability

Reference : [9] H. Lee, S.-j. Cho, H. Han, W. Cho and K. Suh. Enhancing Sustainability in Machine Learning-based An droid Malware Detection using API calls. 2022 IEEE Fifth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), Laguna Hills, CA, USA, page 131-134, 2022

❖ 조우상[9] 등은 2014~2015년 데이터를 사용해 학습된 모델이 시간이 지나면서 지속가능하지 않음을 밝힘.

- **특히 2019~2020년의 악성 앱에서 자주 사용된 API 호출이 정상 앱에서도 반복적으로 사용하여 모델의 성능이 감소함을 확인**

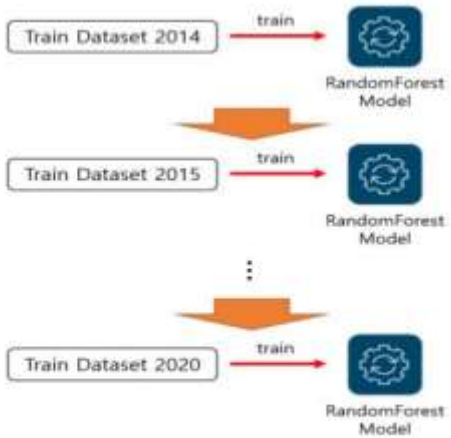


Fig. 2. Incremental Learning Method

Table III. Performance table of Model for Experiments to confirm sustainability

Year	Learning based on the data of 2014~2015		Models for each Machine Learning Method			
			Traditional Machine Learning (TML)		Tree-Keeping Incremental Learning (TKIL)	
	Accuracy	F1_score	Accuracy	F1_score	Accuracy	F1_score
2014	0.9933	0.9963	0.975	0.9859	0.971	0.9840
2015			0.973	0.9847	0.964	0.9801
2016	0.9896	0.9942	0.984	0.9910	0.976	0.9867
2017	0.9625	0.9796	0.993	0.9961	0.99	0.9944
2018	0.9643	0.9802	0.97	0.9831	0.971	0.9837
2019	0.302	0.373	0.975	0.9859	0.509	0.626
2020	0.2466	0.287	0.977	0.9870	0.505	0.6218
AUT	0.8052	0.8275	0.9784	0.9878	0.858	0.8956

안드로이드 멀웨어 탐지에 관한 기존 연구 : How to Measure Sustainability ?

❖ ***Reference : F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. 28th USENIX Security Symposium (USENIX Security 19), 2019***

❖ 위 논문에서는 안드로이드 멀웨어 분류가 해결된 문제인지에 대해 논의

- F1-score가 0.99에 달하는 경우가 있지만, **두 가지의 만연한 실험적 편향(Experimental Bias)으로 결과가 부풀려졌다고 주장**
  - 공간적 편향 (Spatial Bias) : 현실의 도메인을 고려하지 않고 정상/악성 앱의 비율을 실험에 적용해 생기는 편향
  - 시간적 편향 (Temporal Bias) : 시간적 편향은 미래의 지식을 학습시켜 시간적으로 일관되지 않은 결과를 제공하는 편향
- **현실을 모방하는 시간적 및 공간적 조건에서 악성 앱 탐지를 수행하여** 부풀려진 성능을 제한하는 전략과 **지속가능성을 평가할 수 있는 AUT(Area Under Time)를 제안.**

❖ **F. Pendlebury는 안드로이드 악성 앱 탐지 분야에서 위 실험적 편향들은 모델의 성능을 저하시키는 주요 원인으로 꼽았다.**

03 K-Means 기반 분류 모델을 이용한 멀웨어 탐지

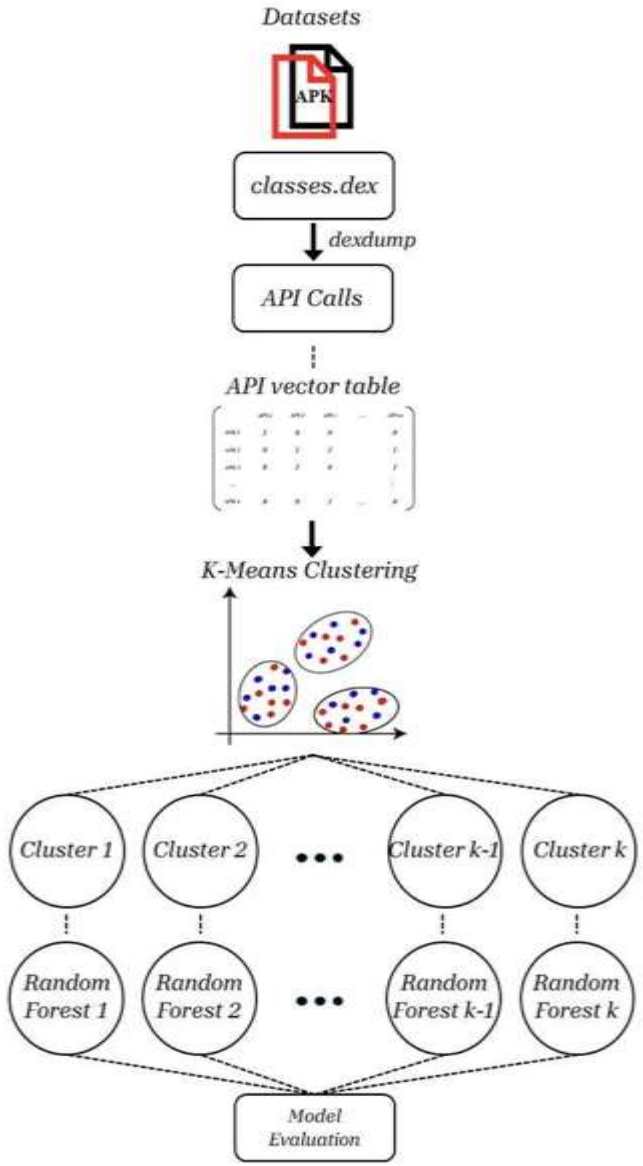
K-Means 기반 분류 모델의 구성

❖ K-Means 클러스터링

- 학습 및 예측 시, 정상/악성 클래스에 관계 없이 앱들을 k개의 클러스터에 할당.

❖ 랜덤 포레스트 (Random Forest) 분류

- 각 클러스터에 데이터들이 할당되면, 랜덤 포레스트 **분류기를 클러스터 별로 학습 및 예측**.
  - **테스트 데이터 예측 시 기학습된 랜덤 포레스트는 정상/악성에 대한 이진 분류 진행**
- 배깅 방식의 랜덤 포레스트는 일반적으로 효과적인 악성 앱 탐지 기법으로 알려져 있음.



K-Means 기반 분류 모델을 이용한 멀웨어 탐지

훈련 데이터 및 테스트 데이터 분할

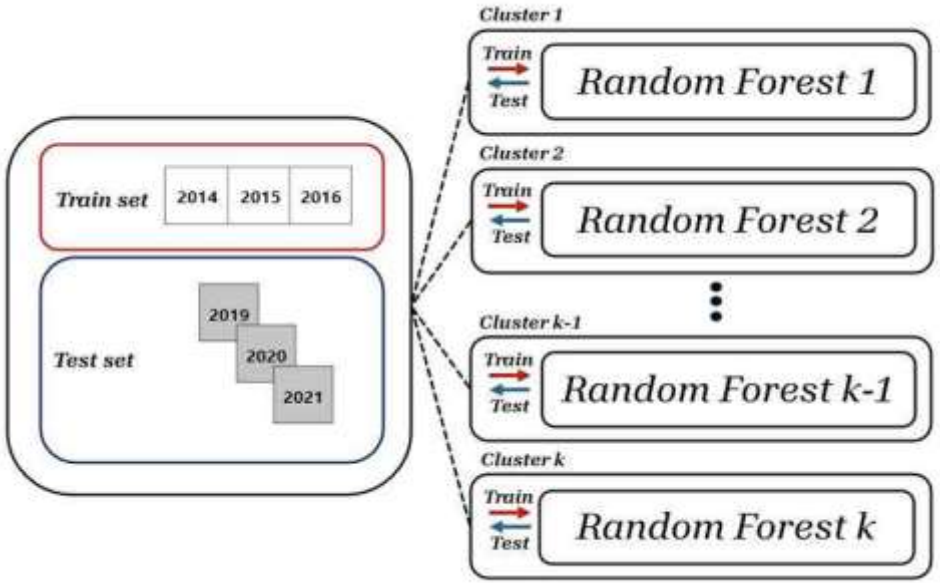
AndroZoo 데이터셋에서 총 48,000개의 APK를 사용

❖ 훈련 데이터

- K-Means 클러스터링과 랜덤 포레스트 분류기 학습 데이터
- **2014~2016년까지** 연도별 6,000개의 정상/악성 앱을 1:1 비율로 사용

❖ 테스트 데이터

- K-Means 클러스터링과 랜덤 포레스트 분류기 예측 데이터
- **훈련 데이터보다 미래 시점의 데이터 → 모델의 강건성 평가**
- **테스트는 1년 단위로 진행**
  - **각 테스트 슬롯의 훈련/테스트 데이터는 9:1의 비율을 가진다**
- **2019~2021년까지** 연도별 2,000개의 정상/악성 앱을 1:1 비율로 사용



dex_date	정상 앱	악성 앱
2014	6,000	6,000
2015	6,000	6,000
2016	6,000	6,000
2019	2,000	2,000
2020	2,000	2,000
2021	2,000	2,000
Total	24,000	24000

평가 메트릭 : Confusion Matrix

❖본 모델의 성능을 평가 지표는 혼동행렬(Confusion Matrix)을 기반으로 하며, 이 때 임의의 클러스터 C_i의 혼동행렬 요소는 아래와 같이 정의한다.

❖ TP_i : 악성 앱을 악성 앱으로 탐지한 수

❖ TN_i : 정상 앱을 정상 앱으로 탐지한 수

❖ FP_i : 정상 앱을 악성 앱으로 탐지한 수

❖ FN_i : 악성 앱을 정상 앱으로 탐지한 수

		Predicted	
		Negative **(N)** -	Positive **(P)** +
Actual	Negative -	True Negative **(TN)**	**False Positive (FP) Type I Error**
	Positive +	**False Negative (FN) Type II Error**	True Positive **(TP)**

평가 메트릭 : Micro Average 방식

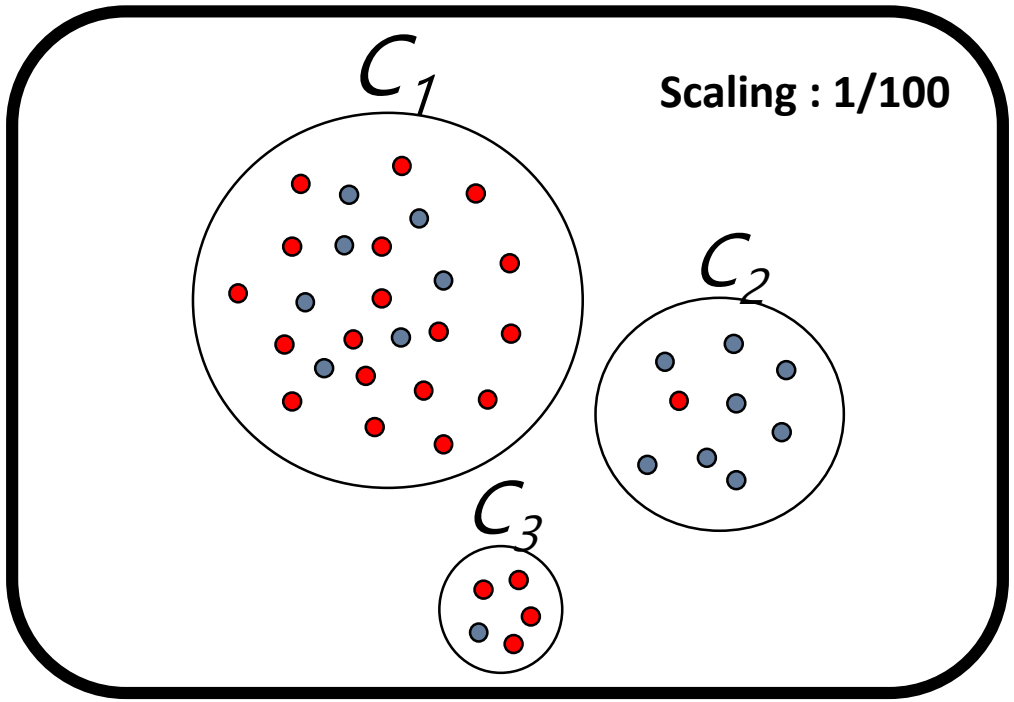
❖ ***Reference : Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. Inf. Process. Manage. 45, 4 (July, 2009), 427–437***

❖4.5절에 따르면, 클러스터 별 개체수, 정상/악성 앱 불균형을 확인할 수 있다.

- 따라서 본 실험에서는 **모든 클러스터의 예측 결과를 하나의 혼동행렬로 합산한 후** 정밀도(Precision), 재현율(Recall), F1-score를 계산하는 마이크로 평균(Micro-Average) 방식을 사용한다

[표 1] 2014~2016년 K-Means 학습 결과

	Cluster	개체수	비율	악성 앱 개체수	클러스터 내 악성 앱 비율
k=2	C_1	28,388	0.7886	14,235	0.5014
	C_2	7,612	0.2114	3,765	0.4946
k=3	C_1	22,563	0.6268	12,246	0.5427
	C_2	9,956	0.2765	3,483	0.3498
	C_3	3,481	0.0966	2,271	0.6524
k=5	C_1	16,884	0.4690	10,156	0.6015
	C_2	2,938	0.0816	1,921	0.6538
	C_3	10,114	0.2809	3,685	0.3643
	C_4	5,853	0.1626	2,083	0.3559
	C_5	211	0.0586	155	0.7346
k=7	C_1	14,263	0.3962	8,911	0.6248
	C_2	3,403	0.0945	1,281	0.3764
	C_3	5,615	0.1560	1,841	0.3279
	C_4	106	0.0029	86	0.8113
	C_5	880	0.0244	604	0.6864
	C_6	9,287	0.2580	3,721	0.4007
	C_7	2,446	0.0679	1,556	0.6361



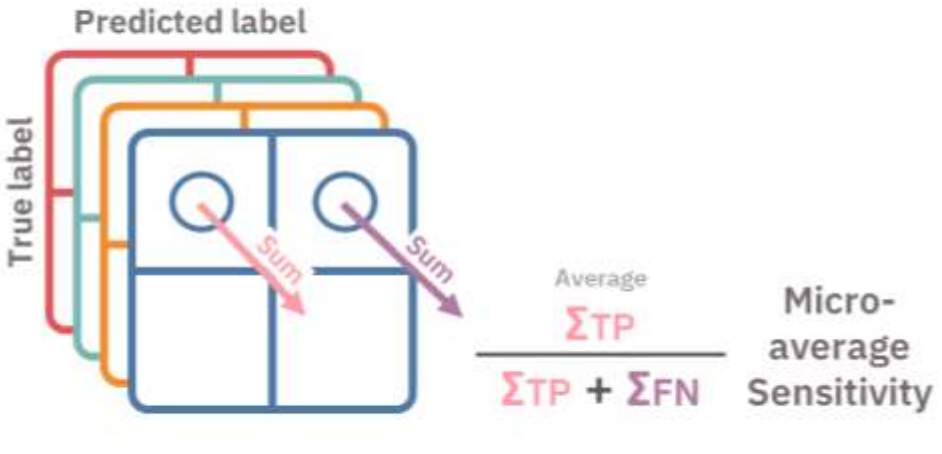
❖Micro-Average

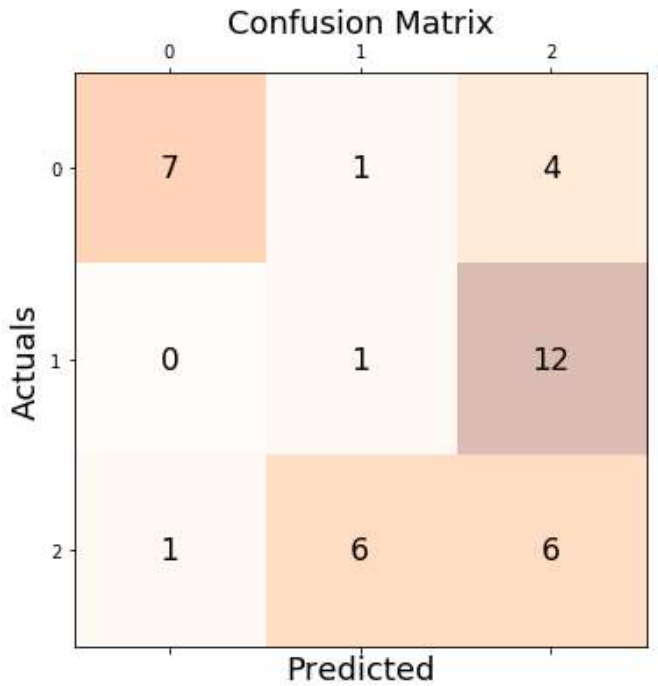
- **모든 클래스의 수치를 합산하여 누적된 TP,FN,TN.FP를 구한 후, 이를 기반으로 성능 지표를 계산하는 방식**
  - Micro average에서 “average”라는 용어는 다중 클래스/레이블 분류에서 성능 지표를 집계(aggregate)하여 **단일 값으로 평균화하는 과정에서 유래**
  - **다중 레이블 분류에서** 마이크로 평균 방식의 정확도(Accuracy) 지표는 일반적으로 사용하지 않는다[15].

$$\mathrm{Pr}^{\mu} = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} (TP_i + FP_i)}$$

$$Re^{\mu} = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} (TP_i + FN_i)}$$

$$F_1^{\mu} = \frac{2 \cdot Pr^{\mu} \cdot Re^{\mu}}{Pr^{\mu} + Re^{\mu}}$$





Modified AUT(Area Under Time)

❖ ***F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. 28th USENIX Security Symposium (USENIX Security 19), 2019***

❖지속가능성의 측정단위를 연도시간으로 수정한 AUT를 아래 수식과 같이 제안한다.

$$AUT^{+}(f, N, n) = \frac{1}{n-1} \sum_{k=1}^{n-1} \frac{[f(x_{k+1}) + f(x_k)] \cdot (N/n)}{2}$$

❖여기서 $f(x_k)$는 k지점에서 평가된 성능 값이며, F1-Score, Recall, Precision 등이 사용된다.

❖ N과 n은 각각 데이터 수집기간과 테스트 슬롯을 의미하며 슬롯 수는 N/n으로 표현된다.

❖ 2019년부터 2021년도까지 N은 3Y 이므로 (F_1^{μ}, 3Y, 3)을 사용한다.

04 실험 결과

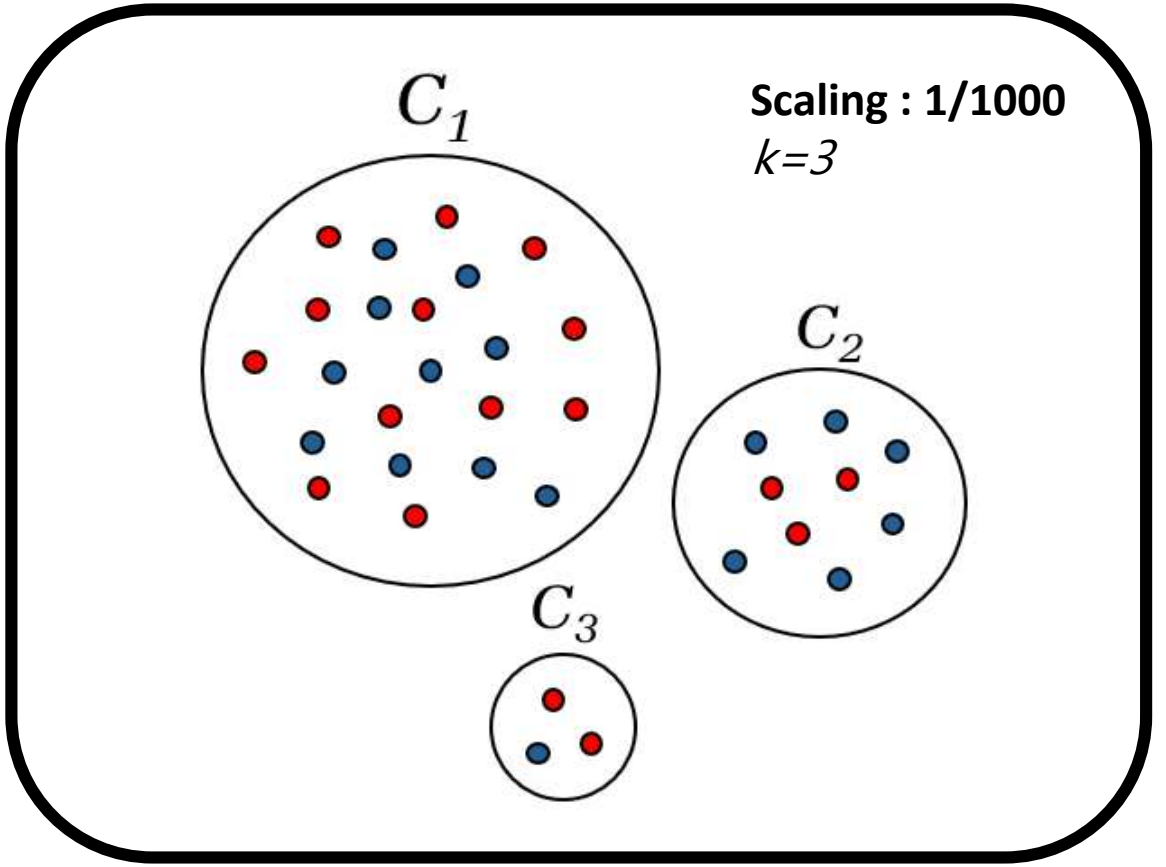
2014~2016년 K-Means 학습 결과

❖ 클러스터의 수(k)에 상관없이 하나의 거대 클러스터(C_1)가 생성되는 것을 확인할 수 있다.

❖또한 클러스터의 수(k)가 증가하면 각 클러스터의 **개체수 불균형은 완화되지만**, C_1을 비롯한 각 클러스터 내의 정상/악성 앱들의 **클래스 불균형은 심화된다.**

[표 1] 2014~2016년 K-Means 학습 결과

	Cluster	개체수	비율	악성 앱 개체수	클러스터 내 악성 앱 비율
k=2	C_1	**28,388**	**0.7886**	**14,235**	**0.5014**
	C_2	7,612	0.2114	3,765	0.4946
k=3	C_1	**22,563**	**0.6268**	**12,246**	**0.5427**
	C_2	9,956	0.2765	3,483	0.3498
	C_3	3,481	0.0966	2,271	0.6524
k=5	C_1	**16,884**	**0.4690**	**10,156**	**0.6015**
	C_2	2,938	0.0816	1,921	0.6538
	C_3	10,114	0.2809	3,685	0.3643
	C_4	5,853	0.1626	2,083	0.3559
	C_5	211	0.0586	155	0.7346
k=7	C_1	**14,263**	**0.3962**	**8,911**	**0.6248**
	C_2	3,403	0.0945	1,281	0.3764
	C_3	5,615	0.1560	1,841	0.3279
	C_4	106	0.0029	86	0.8113
	C_5	880	0.0244	604	0.6864
	C_6	9,287	0.2580	3,721	0.4007
	C_7	2,446	0.0679	1,556	0.6361



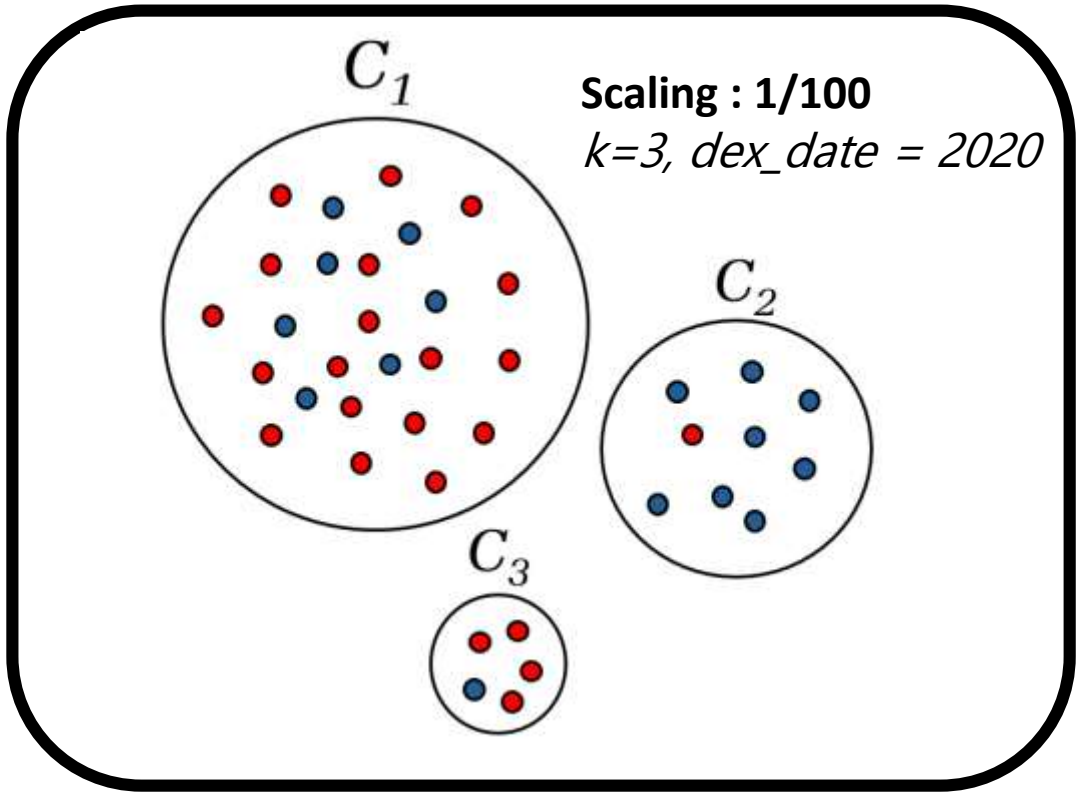
2019~2021년 K-Means 예측 결과 : 연도별 (dex_date)

❖ 클러스터의 수(k)가 3일 때를 기준으로 [표 1]과 비교했을 때, 가장 크기가 큰 C_1은 평균 악성 앱 비율이 약 20%p 증가했다.

❖또한 두 번째로 큰 C_2는 정상 앱의 비율이 증가하였다. C_3은 테스트 데이터마다 악성 앱의 비율이 변동하지만, C_2보다는 높은 악성 앱 비율을 보였다.

[표 2] 연도에 따른 K-Means 예측 결과 (k=3)

	Cluster	유형	개체수	클러스터 내 비율
k=3 (2019)	C_1	정상	717	0.2792
		악성	1,851	0.7208
	C_2	정상	793	0.9308
		악성	59	0.0692
	C_3	정상	490	0.8448
		악성	90	0.1552
k=3 (2020)	C_1	정상	544	0.2290
		악성	1,831	0.7709
	C_2	정상	756	0.9594
		악성	32	0.0406
	C_3	정상	700	0.8363
		악성	137	0.1634
k=3 (2021)	C_1	정상	418	0.2739
		악성	1,108	0.7260
	C_2	정상	829	0.9431
		악성	50	0.0569
	C_3	정상	753	0.4721
		악성	842	0.5279



2019~2021년 K-Means 예측 결과 : 클러스터의 개수 별 (k)

❖ 클러스터의 수(k)가 증가함에 따라 개체수 불균형은 완화되고,**정상/악성 클래스 불균형은 심화되는 경향은** [표 1]의 결과와 유사하다.

❖조우상 등은 2014~2015년 데이터를 사용해 학습된 모델이 시간이 지나면서 지속가능하지 않음을 밝힘.

- 특히 2019~2020년의 악성 앱에서 자주 사용된 API 호출이 정상 애플리케이션에서도 반복적으로 사용하여 모델 의 성능이 감소함을 확인

[표 1] 2014~2016년 K-Means 학습 결과

	Cluster	개체수	비율	악성 앱 개체수	클러스터 내 악성 앱 비율
k=2	C_1	28,388	0.7886	14,235	0.5014
	C_2	7,612	0.2114	3,765	0.4946
k=3	C_1	22,563	0.6268	12,246	0.5427
	C_2	9,956	0.2765	3,483	0.3498
	C_3	3,481	0.0966	2,271	0.6524
k=5	C_1	16,884	0.4690	10,156	0.6015
	C_2	2,938	0.0816	1,921	0.6538
	C_3	10,114	0.2809	3,685	0.3643
	C_4	5,853	0.1626	2,083	0.3559
	C_5	211	0.0586	155	0.7346
k=7	C_1	14,263	0.3962	8,911	0.6248
	C_2	3,403	0.0945	1,281	0.3764
	C_3	5,615	0.1560	1,841	0.3279
	C_4	106	0.0029	86	0.8113
	C_5	880	0.0244	604	0.6864
	C_6	9,287	0.2580	3,721	0.4007
	C_7	2,446	0.0679	1,556	0.6361

[표 3] k에 따른 K-Means 예측 결과 (dex_date=2019)

	Cluster	유형	개체수	클러스터 내 비율
k=2 (2019)	C_1	정상	1,141	0.3776
		악성	1,881	0.6224
	C_2	정상	859	0.8783
		악성	119	0.1216
k=3 (2019)	C_1	정상	717	0.2792
		악성	1,851	0.7208
	C_2	정상	793	0.9308
		악성	59	0.0692
	C_3	정상	490	0.8448
		악성	90	0.1552
k=5 (2019)	C_1	정상	353	0.1637
		악성	1,803	0.8363
	C_2	정상	323	0.8874
		악성	41	0.1126
	C_3	정상	704	0.9084
		악성	71	0.0916
	C_4	정상	488	0.9295
		악성	37	0.0704
	C_5	정상	132	0.7333
		악성	48	0.5687

랜덤 포레스트 분류 및 혼동행렬 합산 결과

❖클러스터의 수(k)가 3인 경우, 각 클러스터의 랜덤 포레스트 분류 결과를 합산하여 연도별로 나타낸다.

- 클러스터별 TP와 FP의 비율이 압도적으로 높은 비율을 차지했다. 이는 랜덤 포레스트 모델이 **2014~2016년 훈련 데이터에 대하여 미래 시점의 테스트 데이터를 예측할 때, 악성 앱으로 탐지되는 경향이 높음을 나타낸다.**
- 특히 2020년의 경우 2019년에 비해 F_1 μ가 약 1%p 소폭 감소했고, 2019년과 2021년의 결과는 미세한 차이를 보였다.

[표 4] 혼동행렬 합산 결과 (k=3)

k=3	예측 / 실제	정상 앱	악성 앱
2019	정상 앱	434 (TN)	1,566 (FP)
	악성 앱	25 (FN)	1,975 (TP)
2020	정상 앱	360 (TN)	1,640 (FP)
	악성 앱	22 (FN)	1,978 (TP)
2021	정상 앱	447 (TN)	1,553 (FP)
	악성 앱	76 (FN)	1,924 (TP)

클러스터 별 분류 결과

❖[표 5]는 [표 4]에서 **k=3, dex_date=2019**일 때 클러스터 별 혼동행렬을 나타낸다.

- 테스트 데이터의 2,000개 악성 앱 중 C_1에서 악성 앱 1,833개의 TP가 검출되었다.
- 또한 세 연도 모두에서 FP의 개체수가 균등하게 분포하는 것으로 나타났다
- 하지만 C_1의 정밀도(Pr_1)는 약 79%의 상대적으로 높은 비율임에 반해 정상 앱의 비율이 93%였던 C_2와 가장 크기가 작은 C_3의 정밀도(Pr_2, Pr_3)는 각각 7.5%, 17%만을 보였다

  → **이는 랜덤 포레스트 모델이 각 클러스터의 개체 수와 정상/악성 클래스 불균형을 고려하지 않고 많은 수의 정상 앱을 악성 앱으로 탐지했음을 의미**

[표 5] 클러스터별 분류 결과 (k=3, dex_date=2019)

	예측 / 실제	정상 앱	악성 앱
C_1	정상 앱	223 (TN)	494 (FP)
	악성 앱	18 (FN)	1,833 (TP)
C_2	정상 앱	147 (TN)	646 (FP)
	악성 앱	6 (FN)	53 (TP)
C_3	정상 앱	64 (TN)	426 (FP)
	악성 앱	1 (FN)	89 (TP)

최종 $F_1{}^{\mu}$, AUT^{+} 산출 결과

❖Micro-Average 방식으로 F1-Score을 계산 후 AUT^{+} 를 산출하여 기존의 랜덤 포레스트 방식과 비교했다

- 전반적으로, 제안된 모델이 기존의 방식보다 F1-Score와 AUT^{+}의 면에서 더 좋은 성능을 보임
- 특히, 2019~2021년에 걸쳐 AUT^{+}가 가장 높은 클러스터 수(k)는 15였으며, 연도별 $F_1{}^{\mu}$의 평균을 토대로 AUT^{+} 를 계산한 값은 0.7182였다.
  - 이는 K-Means를 적용하지 않은 기존의 랜덤 포레스트 방식보다 3.4%p 높은 수치이다.

	Traditional Machine Learning(RF)	Micro Avg.							
		k=2	k=3	k=5	k=7	k=11	k=13	k=15	Avg.
2019	**0.6927**	0.7004	0.7129	0.7082	0.7351	0.7330	0.7346	0.7399	**0.7234**
2020	**0.6807**	0.6895	0.7042	0.6966	0.7239	0.7286	0.7245	0.7367	**0.7148**
2021	**0.6826**	0.6876	0.7026	0.7070	0.7305	0.7428	0.7273	0.7416	**0.7199**
AUT^{+}	**0.6842**	0.6917	0.6949	0.7021	0.7284	0.7332	0.7277	0.7387	**0.7182**

04 결론 및 향후 연구

결론

❖K-Means 클러스터링 결과

- 학습 및 예측 시, 클러스터의 개수(k)를 늘릴수록 클러스터 내 개체수의 불균형은 줄어들었지만, 정상/악성 클래스 간 불균형이 심화되는 경향을 보였음.

❖지속가능성 평가

- Micro-Average 방식으로 지속가능성 평가지표를 산출한 결과, 기존 랜덤 포레스트 방식보다 3.4%p 향상된 수치를 얻었음.

❖한계점

- 랜덤 포레스트는 클러스터 내 불균형을 고려하지 않아 정상 앱을 악성 앱으로 탐지하는 비율이 높다는 한계점이 드러남.

향후 연구 계획

❖ 최적 클러스터의 개수(k) 탐색

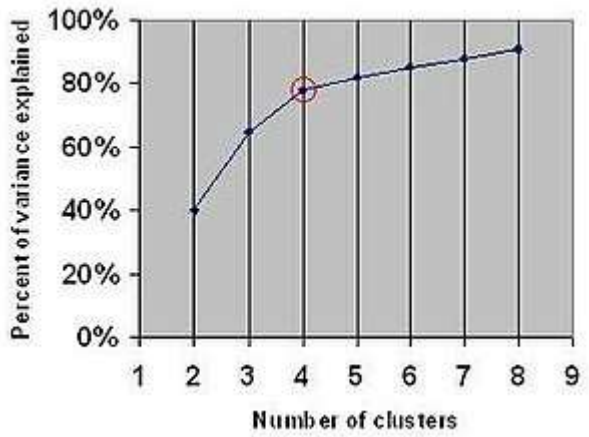
- Elbow Method등을 이용하여 데이터의 균형과 정확도를 모두 고려할 수 있는 최적의 클러스터 개수를 탐색

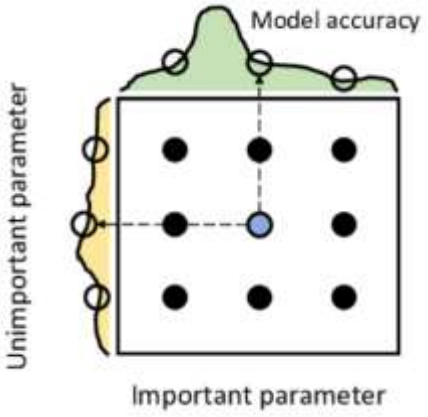
❖ 각 클러스터의 특성을 고려한 랜덤 포레스트 최적화

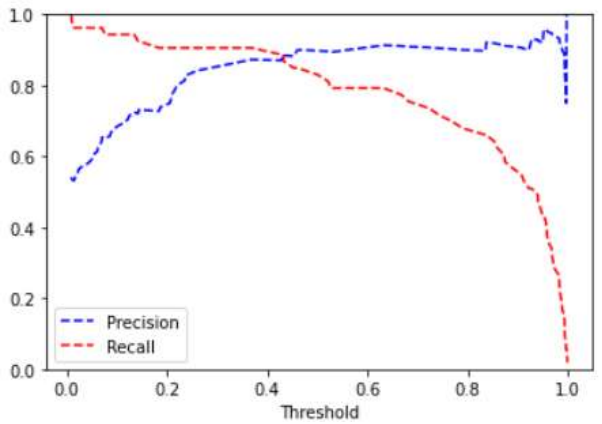
- 클러스터 별 랜덤 포레스트 분류기에 그리드 탐색을 이용한 초매개변수(Hyper Parameter) 튜닝 진행

❖ 임계값 설정

- 악성 앱 탐지 시 오탐 및 미탐 비율을 줄이기 위해 균형 잡힌 임계값 설정을 목표







Acknowledgements

이 논문은 산업통상자원부(MOTIE)와 한국에너지기술평가원(KETEP)의 지원을 받아 수행한 연구과제임. (No. 20212020800120)

또한,

이 연구는 2021년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임.(no. 2021R1A2C2012574)

Thank you

Q&A

❖ Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. Inf. Process. Manage. 45, 4 (July, 2009), 427–437.

❖ **큰 클래스에 더 민감하다 → 본 실험의 전체 데이터셋은 정상 / 악성의 비율이 1:1**

❖ Micro-Average 방식은 각 샘플을 개별적으로 려하여 평균을 계산하는 방식으로, 정확도를 제외한 모든 지표에 적합 → **3개 이상의 Class/Label에서는 Accuarcy의 정의와 계산 방식**이 Micro averaging의 접근 방식과 맞지 않기 때문.

$$\mathrm{Pr}^{\mu} = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} (TP_i + FP_i)}$$

$$Re^{\mu} = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} (TP_i + FN_i)}$$

$$F_1^{\mu} = \frac{2 \cdot Pr^{\mu} \cdot Re^{\mu}}{Pr^{\mu} + Re^{\mu}}$$