



UNIVERSITÀ DEGLI STUDI DEL MOLISE
DIPARTIMENTO DI BIOSCIENZE E TERRITORIO
CORSO DI LAUREA IN INFORMATICA

Tesi di Laurea in
PROGRAMMAZIONE WEB E MOBILE

**Multiple Instance Learning per
l'identificazione di collusioni malevole in
Android**

Relatore
Ing. Francesco Mercaldo

Candidato
Andrea D'Aguanno
Matricola: 156708

Correlatore
Dott.ssa Rosangela Casolare

Anno Accademico 2019/2020

Ai miei genitori.

Grazie.

Abstract

Nell'ambito della sicurezza informatica attraverso il termine Malware, si vanno ad indicare tutti quei programmi sviluppati con l'intenzione di arrecare danno o mettere a rischio un sistema informatico. Un malware può agire a diversi livelli. Il termine stesso racchiude diverse categorie di software malevolo: Virus, Ransomware, Trojan, Worm, sono solo alcuni dei possibili tipi di malware. Android OS è un sistema operativo sviluppato per i dispositivi mobile. Ad oggi risulta essere il più utilizzato, questo perché è un sistema open source non legato ad un hardware specifico che lo ha portato quindi ad essere installato su molti device. Ma la diffusione di Android è dovuta anche alla moltitudine di applicazioni che oggi sono disponibili, queste però, se non installate da fonti sicure possono portare all'intrusione di codice malevolo. Uno dei metodi per nascondere il codice malevolo all'interno di applicativi Android avviene attraverso il "Colluding". In questo caso il codice malevolo è suddiviso tra diverse applicazioni che comunicano tra loro, scambiandosi ad esempio informazioni sensibili, all'interno di canali nascosti. In questo modo si riescono ad aggirare tutte quelle misure di sicurezza che utilizzano analisi statica per rilevare il comportamento malevolo di una sola applicazione malevola. Da qui, la motivazione che ci ha portato allo sviluppo di questa tesi, è legata alla classificazione di queste minacce, nello specifico partendo da file Apk colludenti. L'obiettivo è quello di utilizzare tecniche di machine learning, nello specifico di multiple instance learning per la classificazione di applicativi colludenti affetti da malware. I risultati raggiunti sono stati soddisfacenti permettendo, alla metodologia sviluppata, di effettuare con successo classificazioni attraverso diverse classi, di set di Apk colludenti.

Indice

Elenco delle figure	ix
1 Introduzione	1
1.1 Contesto applicativo	1
1.2 Motivazioni e obiettivi	2
1.3 Risultati raggiunti	2
1.4 Organizzazione della tesi	3
2 Android	5
2.1 Il sistema operativo	6
2.1.1 L'architettura	7
2.2 Le applicazioni Android	10
2.2.1 Struttura	10
2.2.2 Componenti	11
2.2.3 Gestione dei dati	14
2.3 Il package Apk	15
3 Sicurezza in ambiente Android	19
3.1 Malware	21
3.1.1 Colluding	22
4 La metodologia	25
4.1 I software e i dataset	26
4.1.1 WEKA	26
4.1.2 dataset.csv e dataset.arff	26

4.1.3	Dataset di applicativi android	28
4.2	Elaborazione dei file audio	30
4.2.1	Generazione dei file audio	30
4.2.2	Splitting dei file audio	30
4.3	Estrazione delle features	32
4.4	Il multiple instance learning	36
4.5	K-fold validation	39
4.6	Precision e Recall	40
5	Sperimentazioni	41
5.1	Classificazione multiclasse	41
5.2	Classificazione binaria Trusted - Malware	43
5.3	Classificazione binaria Apk_Get - Apk_Put	45
6	Conclusioni e sviluppi futuri	47
	Bibliografia	49

Elenco delle figure

2.1	Worldwide; Newzoo; 2016 to 2020	5
2.2	Mobile & Tablet Operating System Market Share Worldwide .	6
2.3	Android architecture	7
2.4	Android lifecycle	12
2.5	Apk package	16
3.1	Malware detection 2019 - G DATA	19
3.2	malware detection 2020 - Avira	20
3.3	Android OS Version - GlobalStats statcounter	21
4.1	Methodology of development	25
4.2	Input Mil package in weka	26
4.3	ARFF dataset for MIL	28
4.4	CSV file	28
4.5	Stolen resources	29
4.6	first split	32
4.7	second split	32
4.8	k th selected segment	39
4.9	K-fold in weka	39
5.1	Multi class dataset	42
5.2	Multi class classification	43
5.3	Binary class dataset	44
5.4	Get - Put class dataset	45
6.1	Output TLC model	48

Capitolo 1

Introduzione

1.1 Contesto applicativo

Nell'ambito della sicurezza informatica, attraverso il termine malware, abbreviazione di *malicious software*, si vanno ad indicare tutti quei "software malevoli" il cui scopo ultimo è quello di andare ad interferire con le operazioni svolte da un utente in un dispositivo elettronico. Un malware può agire a diversi livelli. Il termine stesso racchiude diverse categorie di software malevolo: Virus, Ransomware, Trojan, Worm, sono solo alcuni dei possibili tipi di malware. La caratteristica che differenzia un malware da un qualsiasi altro applicativo software risiede nell'intenzionalità dello sviluppatore di voler realizzare codice malevolo, quindi non rientrano in questo contesto tutti quei software che presentano un bug che mina alla sicurezza dell'applicativo.

È dunque fondamentale studiare i rischi ed i pericoli che riguardano il mondo dell'informatica e più nello specifico quello del mobile computing¹. Uno dei sistemi operativi più diffusi per i dispositivi mobile è *Android* che grazie alla sua flessibilità permette all'utente di eseguire particolari azioni liberamente all'interno del proprio ecosistema. Android ad oggi è leader del mercato degli smartphone, ed è quindi il sistema operativo più utilizzato a livello mondiale con 2,5 miliardi di utenti attivi mensilmente nella prima metà del 2019[1].

¹Con il termine mobile computing, si vanno ad indicare tutti quei dispositivi che non sono vincolati dall'utilizzo in una posizione fissa. Rientrano in questa categoria anche i dispositivi mobile come gli smartphone.

La sua diffusione è stata incentivata anche perché non è collegato con un produttore hardware specifico portando il sistema operativo di Google ad avere solo nel 2015 più di quattromila dispositivi diversi che utilizzassero Android [2].

1.2 Motivazioni e obiettivi

La possibilità di flessibilità dell'ecosistema Android apre le porte ad una maggior intrusione di codice malevolo all'interno degli applicativi, si pensi che oltre alla possibilità di installare applicazioni dallo store ufficiale *Google Play* è possibile scaricare ed installare applicazioni attraverso il reperimento di un qualsiasi file *.apk*² da un generico sito web. Gran parte dei malware Android si diffondono proprio attraverso l'utilizzo di quest'ultima pratica. Difatti, scaricando applicazioni da siti terzi non è garantita una preventiva scansione del download per rilevarne la dannosità come avviene attraverso l'utilizzo dello store ufficiale, più nello specifico attraverso l'utilizzo di *Google Play Protect* attivo di default in *Google Play*.

Da qui, la motivazione che ci ha portato allo sviluppo di questa tesi, è legata alla classificazione di queste minacce, nello specifico partendo da file *Apk* colludenti. L'obiettivo è quello di utilizzare tecniche di machine learning, nello specifico di multiple instance learning per la classificazione di applicativi colludenti affetti da malware.

1.3 Risultati raggiunti

Dato un set di file *Apk* colludenti, si è passati alla conversione del codice binario relativo in un file audio, nello specifico un file *Wav*³, di seguito si è proceduto ad una suddivisione, ricavando quindi da un singolo file *Wav* più file audio. Da questi ultimi, si è passati all'estrazione di alcune delle caratte-

²L'estensione *Apk* - Android Package indica un pacchetto contenente tutti i file relativi ad una applicazione Android.

³L'estensione *Wav* - Waveform Audio File Format è uno standard di formato di file audio, sviluppato da IBM e Microsoft, per memorizzare un flusso di bit audio. [3]

ristiche che li compongono. Creando un dataset ad hoc, abbiamo addestrato modelli di multiple instance learning per la classificazione di applicazioni sicure o applicazioni colludenti affette da malware.

1.4 Organizzazione della tesi

La tesi è sviluppata progressivamente nei seguenti capitoli, ai quali va ad aggiungersi il capitolo corrente relativo all'introduzione:

- **Capitolo 2** Android - in questo capitolo esporremo l'architettura del sistema operativo, i file che compongono un applicativo Android ed infine cosa sono i file Apk.
- **Capitolo 3** Sicurezza in ambiente Android - in questo capitolo, dopo una breve introduzione sui tipi di malware ci concentreremo sulle applicazioni colludenti e su cosa le caratterizza.
- **Capitolo 4** Metodologie - nel capitolo relativo le metodologie vengono esposti i software ed i dataset utilizzati per la classificazione, successivamente vengono descritti i passaggi che hanno permesso l'estrazione delle caratteristiche dalle applicazioni ed infine una panoramica sul multiple instance learning.
- **Capitolo 5** Sperimentazioni - in questo capitolo sono riportati i risultati delle sperimentazioni accompagnati da tabelle delle metriche.
- **Capitolo 6** Conclusione e Sviluppi futuri - nel capitolo vengono riassunti le conclusioni e gli sviluppi futuri.

Capitolo 2

Android

Nell'ultimo decennio, il numero di utenti che utilizza dispositivi mobile è progressivamente aumentato. Questo trend è dovuto ad un aumento dell'affidabilità e delle prestazioni, sia dell'hardware che del software. Ad oggi gli utenti che utilizzano uno smartphone in tutto il mondo superano il numero di tre miliardi di persone. Questo numero è destinato a salire di diverse centinaia di milioni nel corso dei prossimi anni. Una stima suggerisce che nel 2023 gli utenti che utilizzeranno questo strumento sarà di circa 4,3 miliardi [4], come si può osservare in figura 2.1.

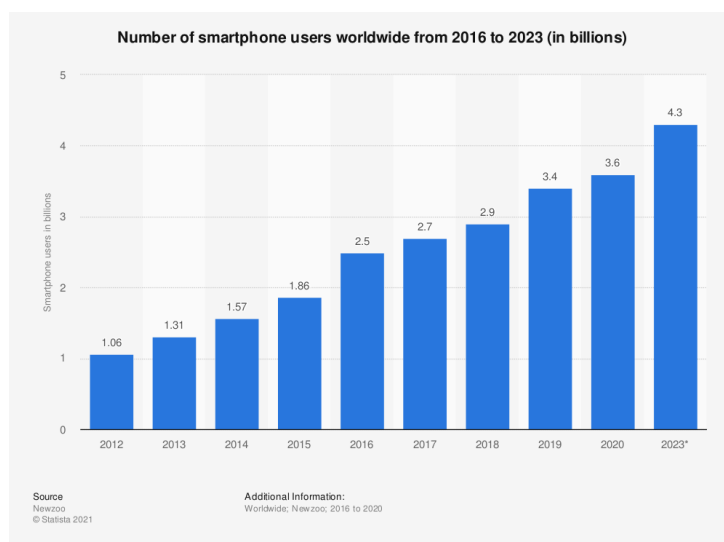


Figura 2.1: Worldwide; Newzoo; 2016 to 2020

Tra i sistemi operativi più utilizzati c'è il sistema operativo *Android*, basti pensare che, nel solo anno 2020, il 71.41% dei dispositivi mobile, nello specifico tablet e smartphone, venduti aveva come sistema operativo proprio *Android*, che insieme ad iOS, hanno coperto il 99.37% del mercato mobile come si può osservare in figura 2.2. L'ultima versione di questo sistema operativo *Android R - Android 11* è stata rilasciata nel settembre 2020 e la successiva versione Beta di *android 12* è prossima al rilascio.

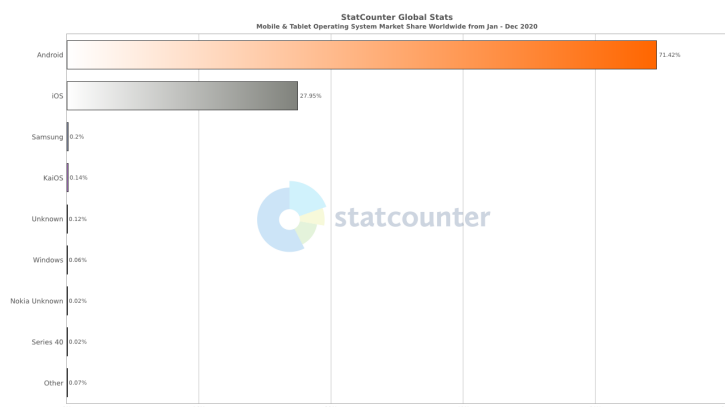


Figura 2.2: Mobile & Tablet Operating System Market Share Worldwide

La diffusione del sistema operativo *Android* è avvenuta per mezzo di smartphone e tablet, ma sono state sviluppate soluzioni ottimizzate per dispositivi specifici come *Wear Os* per smartwatch, *Android TV* per smart tv, *Android Auto* per l'integrazione tra smartphone ed auto, che ne aumentano ulteriormente il bacino di utenza.

Nei prossimi paragrafi esporremo l'architettura del sistema operativo *Android* ed i principali componenti che formano un applicativo *Android*. Infine descriveremo i file *Apk*.

2.1 Il sistema operativo

Come già detto in precedenza *Android* è un sistema operativo per dispositivi mobile, sviluppato dall'azienda *Android, Inc.* acquistata nel 2005 dalla statunitense *Google* che lo ha poi diffuso nel 2008. Il modello di sviluppo

è open source, questo consente, a chiunque sia interessato, di progettare e sviluppare componenti software ad esso dedicati, anche grazie alle librerie e alla documentazione fornita dal produttore. Inoltre, essendo distribuito con licenza *Apache 2.0*, consente a chiunque di modificare e distribuire il codice sorgente. Le applicazioni sviluppate per dispositivi Android sono scritte in linguaggio Java o Kotlin, linguaggi molto diffusi che consentono a molti sviluppatori di prendere parte alla progettazione di applicazioni, questo per far sì che le funzionalità di un dispositivo siano sempre molte ed aggiornate

2.1.1 L'architettura

L'architettura android può essere suddivisa in più livelli come in figura 2.3.

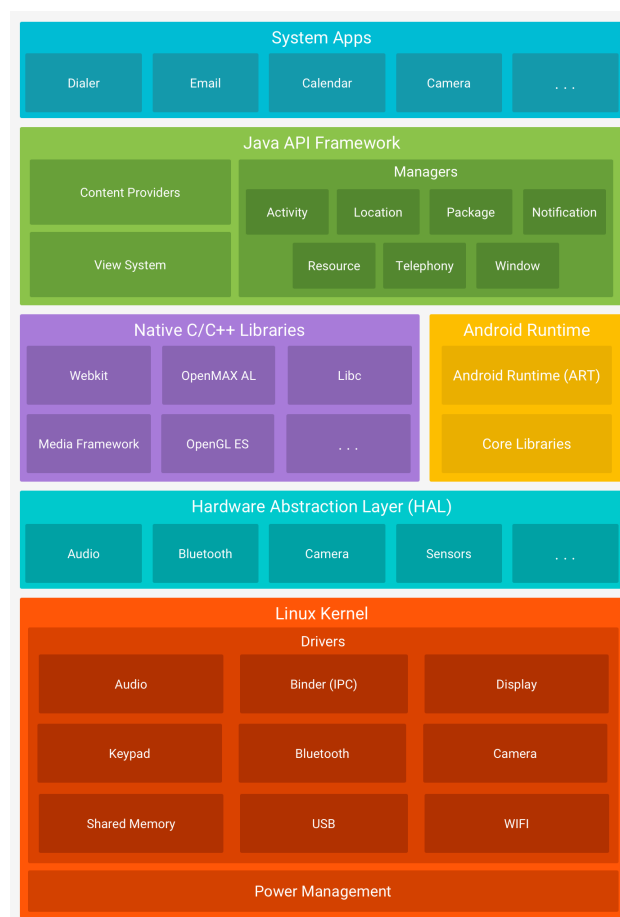


Figura 2.3: Android architecture

- Nel livello più basso troviamo il **kernel linux** al quale Android si appoggia per i servizi del sistema centrale quali come sicurezza e gestione della memoria. Inoltre presenta dei driver specifici per la gestione dell'hardware. Il Kernel funge quindi da strato di comunicazione tra hardware e software.

Driver Binder(IPC): questo componente consente al sistema operativo la comunicazione tra i processi, gestendo il passaggio dei dati tra le applicazioni che in Android sono eseguite ognuna in un processo differente.

Driver Shared Memory: consente la creazione, la mappatura ed il controllo della protezione della memoria condivisa tra i diversi processi.

Nonostante la scelta di utilizzare un kernel linux, per ottenere affidabilità e garantire sicurezza, Android è considerato come una distribuzione embedded Linux, sviluppata appositamente per ottimizzare al meglio le risorse, spesso esigue, di un dispositivo mobile e non è quindi una distribuzione Unix-like, ovvero non fa parte di tutti quei sistemi operativi simili e discendenti dal sistema operativo Unix.

- Salendo di livello troviamo una serie di **librerie native** sviluppate in C / C++ e librerie sviluppate in Java:

Media Framework: è la componente in grado di gestire diversi CODEC per i vari formati di acquisizione e riproduzione di audio e video. È basata sulla libreria open source OpenCORE. Supporta anche formati immagini come png e jpg.

Surface Manager: gestisce l'accesso alle funzionalità del display e coordina le diverse finestre che le applicazioni vogliono visualizzare sullo schermo, permettendo la visualizzazione contemporanea di grafica 2D e 3D dalle diverse applicazioni.

OpenGL ES: attraverso le API implementate in questa libreria si possono gestire le funzionalità 2D e 3D e l'utilizzo di un eventuale acceleratore grafico.

SQLite: libreria che implementa un database relazionale.

WebKit: browser utilizzato da Android per la visualizzazione di risorse

web.

Libc (System C library): implementazione della libreria standard C system (libc), per i dispositivi basati su Linux embedded ad esempio Android.

Secure Socket Layer (SSL): si occupa della sicurezza attraverso la gestione dei Secure Socket Layer. Sono protocolli crittografici che permettono una comunicazione sicura e una integrità dei dati su reti TCP/IP. I SSL cifrano la comunicazione dalla sorgente alla destinazione a livello di trasporto.

- **L'Android runtime (ART)** è un runtime system software che ha sostituito la Dalvik virtual machine (DVM). Quest'ultima infatti si occupava di compilare il codice ad ogni esecuzione di un'applicazione incidendo sulle prestazioni del dispositivo. Con ART invece la compilazione del codice avviene durante l'installazione dell'applicazione, ciò comporta tempi più lunghi per la procedura di installazione, ma una volta terminata, l'esecuzione dell'applicativo non necessita di compilazione. Per mantenere la compatibilità con le versioni precedenti, il bytecode utilizzato da ART per la compilazione è lo stesso utilizzato da DVM ovvero quello fornito dal file *.dex*¹

- **Java API Framework** consiste di Api e componenti sviluppate in Java per l'esecuzione di precise funzionalità di un'applicazione Android.

Activity Manager: responsabile dell'organizzazione delle schermate in uno stack. Rappresenta lo strumento attraverso il quale avviene l'interazione con l'utente.

Attraverso il **Content Provider** si vanno a gestire gli accessi ai dati archiviati dalla stessa applicazione o da terze, fornisce quindi meccanismi per la condivisione di dati tra applicazioni.

Notification Manager implementa i metodi per inviare una notifica al dispositivo ed avvisare l'utente che qualcosa è successo in background.

¹Le applicazioni Android sono comunemente scritte in Java e successivamente compilate in bytecode per la macchina virtuale Java, che viene quindi tradotto in bytecode Dalvik e archiviato in file *.dex* (Dalvik EXecutable) oppure *.odex* (Optimized Dalvik EXecutable).

Con l'utilizzo del **View System** si può renderizzare la GUI (graphical user interface) attraverso l'utilizzo di pulsanti, tabelle, aree di testo e così via in modo da poter gestire gli eventi associati ad ogni elemento. Il codice è contenuto in un file di markup .xml².

- Il primo livello, **System Apps**, contiene tutte le applicazioni native ed installate all'interno del dispositivo Android.

2.2 Le applicazioni Android

In questo paragrafo analizzeremo la struttura di una applicazione Android ed i componenti da cui è composta.

Lo sviluppo delle applicazioni Android può avvenire attraverso l'Ide³ *Android Studio* basato su JetBrains IDEA e rappresenta l'ide principale per lo sviluppo Android di Google. I linguaggi di programmazione utilizzati per sviluppare un applicativo Android come detto anche in precedenza sono Java e Kotlin, quest'ultimo inoltre sta diventando sempre più consigliato e diffuso in ambito Android. Lo sviluppo avviene utilizzando il kit di supporto software Android (SDK) che comprende documentazione dei metodi, un debugger, librerie software ed un emulatore, in Android Studio è possibile gestire ed utilizzare queste componenti da "SDK Manager". L'attività di *run* del codice può avvenire sia su un dispositivo fisico collegato tramite usb e dopo aver attivato la modalità "Debug Usb" oppure scaricando ed installando un'immagine di Android nel "AVD Manager - Android virtual devices".

2.2.1 Struttura

La struttura che compone un progetto Android può essere suddivisa in moduli. Tutto il progetto sarà contenuto in una cartella *app*. All'interno troveremo ulteriori directory che conterranno il codice e le risorse dell'applicativo:

²XML - eXtensible Markup Language, è un metalinguaggio per la definizione di linguaggi di markup dove, oltre all'utilizzo di tag prestabiliti è possibile definirne di propri.

³IDE - integrated development environment è un ambiente di sviluppo, un software che, in fase di programmazione, supporta i programmatori nello sviluppo e debugging del codice sorgente di un programma [5]

- **manifest:** le applicazioni Android dispongono di un file `AndroidManifest.xml`. In questo file vanno definite tutte le componenti sviluppate, la versione minima di API necessarie per eseguire l'applicazione, i permessi di cui necessita l'applicazione ed infine elenca le librerie esterne utilizzate nel codice.
- **java:** all'interno della directory `java` è contenuto il codice sorgente delle componenti e della logica che va a sviluppare l'applicativo, inoltre presenta anche dei file di test.
- **res:** le risorse che non sono codice, come immagini, layout xml, stringhe della GUI, sono contenute all'interno di subdirectory di `res`. Nello specifico troviamo la directory *drawable* che contiene le immagini, la directory *layout* al cui interno sono definiti tutti i file di `layout.xml` associati ad ogni singolo activity e/o fragment creato. Infine in *values* sono contenuti vari marcatori xml utili per la definizione di stringhe, dimensioni, colori e stili.

Il secondo modulo, la sezione *Gradle Scripts*, comprende tutti i file utilizzati per la build del progetto.

2.2.2 Componenti

Le componenti principali che compongono per l'appunto un applicativo sono cinque: Activity, Service, Content Provider, Broadcast Receiver e Intent.

Activity

Un' **Activity** rappresenta l'Interfaccia utente. Ogni schermata è un'activity, all'interno della quale possono alternarsi dei *fragment*. Nel codice viene effettuato l'override del metodo "onCreate" in cui viene specificato il suo layout attraverso `setContentView(R.layout.activity)`. Le activity in Android seguono un ciclo di vita ben definito come si può osservare nella figura 2.4 tratta dalla documentazione ufficiale.

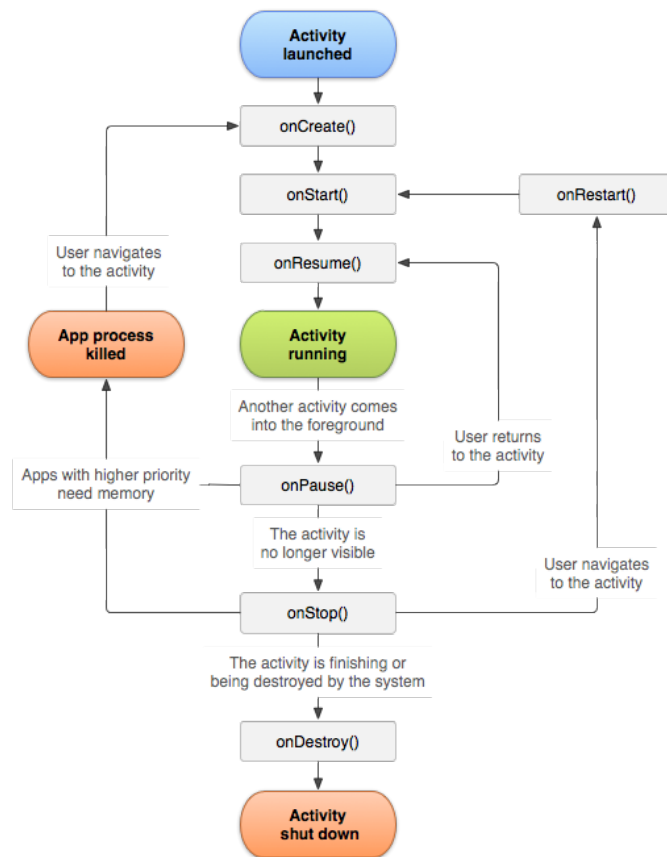


Figura 2.4: Android lifecycle

Nello specifico il metodo *onCreate()* viene invocato per stanziare e definire l'activity, questo viene avviato una sola volta per ciclo di vita. Successivamente quando l'activity è stata creata si invoca il metodo *onStart()* che rende l'activity visibile all'utente, in modo che passi in primo piano e diventi interattiva. Invocando poi il metodo *onResume()* l'activity si pone in "ascolto" di un evento da parte dell'utente, qui definiamo l'interattività con l'user. Rimarrà in questo stato finché non accade un evento che richiede un cambio di activity (ricezione chiamata, navigazione tra activity...), il metodo invocato a questo punto sarà *onPause()* che salva lo stato corrente dell'activity e dunque non sarà più possibile interagire con essa. Il passo successivo comprende l'invocazione di *onStop()* metodo che non rende più visibile l'activity. A questo punto l'utente può tornare all'activity ed il sistema richiamerà *onResume()*

oppure si invocherà *onDestroy()* che provvede alla pulizia e cessazione di un'activity.

Service

Un importante componente sono i **Service**. Questi svolge operazioni in background quindi anche al di fuori dell'utilizzo diretto dell'utente dell'applicazione, difatti non ha un'interfaccia grafica. È inoltre possibile eseguire comunicazione tra processi tramite IPC. I service vanno anch'essi definiti nel file `AndroidManifest.xml`.

Content provider

Come detto in precedenza, un altro componente principale, è il **Content Provider** che si occupa della gestione della condivisione in memoria dei dati salvati in database, su file oppure in rete tra le applicazioni. Risulta fondamentale per il controllo delle autorizzazioni per l'accesso ai dati.

Broadcast receiver

Il **Broadcast receiver** è un componente che, dato un messaggio a livello di sistema, in broadcast, consente la reazione all'evento.

Intent

L'ultimo componente in esame è l'**intent**, questo consente di notificare l'intenzionalità di un'applicazione nel voler richiedere una specifica funzionalità della stessa o da un'altra applicazione attraverso delle invocazioni anche di servizi di broadcast receiver. Il concetto di riuso delle componenti è fondamentale per avere un codice comprensibile e funzionale. I tipi di intent sono *intent espliciti* che specificano il componente da avviare ed *intent impliciti* ovvero coloro che dichiarano un'azione da eseguire⁴. Attraverso l'utilizzo di intent è possibile anche l'invio di parametri extra per lo scambio di informazioni tra oggetti.

⁴Ad esempio un intent implicito può essere la richiesta di scelta di quale applicazione utilizzare per la visualizzazione di un particolare file.

2.2.3 Gestione dei dati

Android permette il salvataggio dei dati delle applicazioni in diverse modalità. **Internal storage** ovvero lo spazio riservato esclusivamente all'uso dell'applicazione. In quest'area di memoria le altre applicazioni non possono accedere, per farlo, bisogna definire un meccanismo sicuro attraverso il content provider di cui abbiamo parlato precedentemente. Attraverso l'**External storage** le informazioni salvate sono invece accessibili a tutte le applicazioni del dispositivo, difatti rappresenta uno spazio condiviso tra le varie applicazioni. Se si vuole utilizzare questo metodo di salvataggio è buona norma controllare attraverso il file manifest che il dispositivo supporti l'estensione della memoria, inoltre se i dati sono sensibili bisogna provvedere all'utilizzo di cifrari per nascondere le informazioni. È possibile poi usufruire delle **Shared-preferences** ovvero il salvataggio di piccole raccolte di dati key-value facilmente definibili ed accessibili. Possono essere sia private che condivisibili. Di seguito è mostrato un esempio di utilizzo di shared preferences per il salvataggio di un orario tramite valori passati come parametri della funzione "doSave" 2.1.

```
1 public void doSave(int hour, int minutes) {
2     SharedPreferences sharedPreferences = CONTEXT.getSharedPreferences("
    DispensaSetting", Context.MODE_PRIVATE);
3     SharedPreferences.Editor editor = sharedPreferences.edit();
4     editor.putInt("hourpreferences", hour);
5     editor.putInt("minutepreferences", minutes);
6     editor.apply();
7 }
```

Listing 2.1: Shared Preference example

Un'ulteriore possibilità per il salvataggio delle informazioni è l'utilizzo di **database**. Android mette a disposizione un database relazionale open source, *SQLite*, scelto per la sua ottimizzazione nel consumo di risorse spaziali. *SQLite* legge e scrive direttamente da e su file su disco. Android utilizza uno schema ben definito per la memorizzazione di file su database. Per ogni applicazione che ne fa uso, esiste una cartella dedicata allo scopo di contenere i file relativi al database `"/data/data/packagename/databases"`. Ancora una volta, per gestire al meglio ed in sicurezza l'accesso alle informazioni in un

database, un ruolo fondamentale è ricoperto da content provider. Utilizzando la libreria ROOM si può usufruire di un livello di astrazione su SQLite che consente di avere un controllo in fase di compilazione sulle query SQL ma soprattutto una gestione semplificata del percorso di migrazione del database. L'api ROOM ha tre componenti: *database*, classe astratta che rappresenta il database e funge da punto di accesso per la connessione ai dati; *entity*, ovvero la classe che rappresenta una tabella del database; *dao*, interfaccia con i metodi per effettuare le operazioni di CRUD⁵. Di seguito un esempio di una interfaccia contenente i metodi dao 2.2.

```
1 @Dao
2 public interface ProdottoDao {
3     @Insert
4     public Long insertProdotto(ProdottoEntity prodottoEntity);
5     @Update
6     public void updateProdotto(ProdottoEntity prodottoEntity);
7     @Delete
8     public void deleteProdotto(ProdottoEntity prodottoEntity);
9     @Query("SELECT * FROM prodottoentity")
10    public List<ProdottoEntity> findAll();
11    @Query("SELECT * FROM prodottoentity WHERE category LIKE :callby")
12    public List<ProdottoEntity> findAllByCategory(String callby);
13 }
```

Listing 2.2: Dao example

Infine un'ulteriore possibilità è quella che sfrutta la connessione internet per eseguire il salvataggio dei dati in cloud.

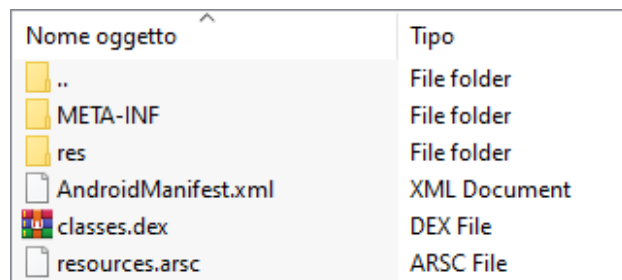
2.3 Il package Apk

Il .Apk è l'estensione che indica che si sta lavorando con una cartella compressa che racchiude tutti i file di un specifica applicazione Android. Sono una variante del formato .JAR⁶. In figura 2.5 possiamo osservare la composizione di un package Apk. Più nello specifico, un file Apk è un archivio

⁵CRUD - create, read, update, and delete sono le quattro operazioni di base della memorizzazione in un database.

⁶Un file con estensione JAR - Java Archive indica un archivio dati compresso usato per distribuire raccolte di classi Java.

che contiene i seguenti file: *AndroidManifest.xml*, *classes.dex* e *resources.arsc* inoltre comprende anche le cartelle *META-INF* e *res*.



Nome oggetto	Tipo
..	File folder
META-INF	File folder
res	File folder
AndroidManifest.xml	XML Document
classes.dex	DEX File
resources.arsc	ARSC File

Figura 2.5: Apk package

Classes.dex

Questo file è composto dall'insieme delle classi che compongono la logica dell'applicativo android. Il risultato sono classi compilate dall'Android SDK per essere eseguite dalla macchina virtuale Android RunTime (ART). Trovandoci in ambiente di sviluppo java, si ha che ogni classe sarà compilata in un file .class; questo passaggio avviene anche in Android a cui però segue la conversione di tutti i file .class in un unico file .dex. Dunque, questo file non conterrà il bytecode⁷ java, ma il bytecode DEX che è stato introdotto appositamente per il sistema android in quanto, tra le altre, va ad ottimizzare l'uso della memoria.

Res e resources.arsc

Nel package Apk troviamo una cartella *res*, questa contiene tutte le risorse dell'applicativo come le immagini ed il layout dei file xml. Mentre *resources.arsc* è un file di risorse in questo caso però compilate.

⁷Il bytecode è un linguaggio intermedio che si posiziona tra il linguaggio macchina e il linguaggio di programmazione. Viene adoperato per descrivere le operazioni che costituiscono un programma.

META-INF

Questa cartella contiene le informazioni del manifest ed altri metadati⁸. Nello specifico il file *MANIFEST.MF* contiene tutte quelle informazioni utilizzate da ART in fase di run-time come, ad esempio, qual è la classe principale e quali sono le politiche di sicurezza.

⁸Sistema di dati il cui scopo è la descrizione di altri dati, compresi gli archivi elettronici.

Capitolo 3

Sicurezza in ambiente Android

Android, come detto, è il leader mondiale del mercato mobile, questo ha portato all'aumento dello sviluppo di diversi tipi di applicazioni aumentando di conseguenza anche il numero degli attacchi attraverso l'utilizzo di applicazioni affette da malware. Un rapporto di Avira descrive come nella prima metà del 2020 si potessero contare quasi 2 milioni di applicazioni android affette da malware [6] figura 3.2, nell'anno precedente le rilevazioni effettuate da G DATA mostravano lo stesso trend [7] come si può osservare in figura 3.1.

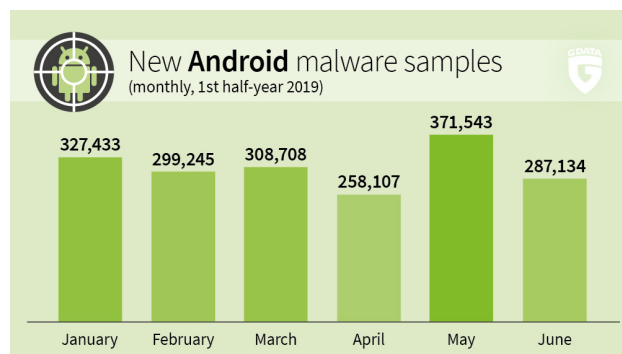


Figura 3.1: Malware detection 2019 - G DATA

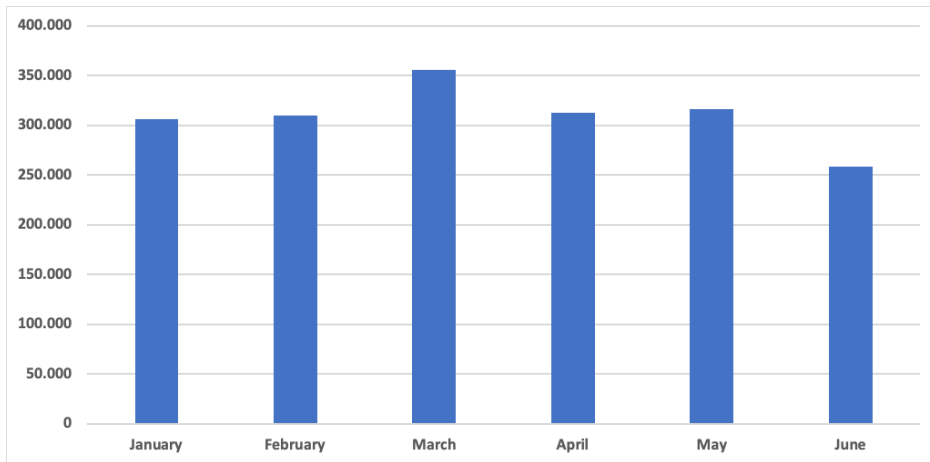


Figura 3.2: malware detection 2020 - Avira

Il malware inoltre è stato la categoria di attacco che più è stata rilevata come minaccia in ambiente android, quasi 3/4 delle rilevazioni riguardavano appunto un malware [6]. Questo è stato reso possibile dalla frammentazione del mercato. Essendo un sistema operativo in continua evoluzione e diffusione, diventa sempre più complesso riuscire a garantire la sicurezza di tutte le versioni in circolazione. Difatti un problema per la sicurezza è legato alla versione android installata sul proprio device. Nel 2017 si stimava ci fossero circa 1 miliardo di dispositivi android che non fossero aggiornati o non avrebbero ricevuto aggiornamenti, rendendo i dispositivi sempre più obsoleti e vulnerabili [8].

In figura 3.3, nell'intervallo che va dal Gennaio 2019 al Gennaio 2021 possiamo infatti osservare come le due versioni più diffuse siano Android 9 (2017) e Android 8 (2018).

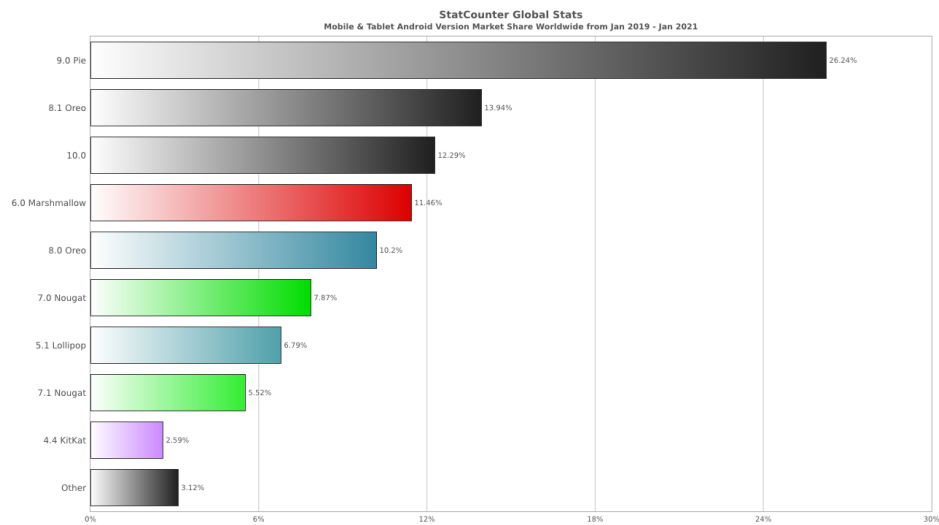


Figura 3.3: Android OS Version - GlobalStats statcounter

Il rischio rimane quindi molto alto e da non sottovalutare. Nel prossimo paragrafo vedremo cosa si intende per malware.

3.1 Malware

Un Malicious software, abbreviato Malware, è un termine che va ad indicare tutti quei programmi che mettono a rischio un sistema informatico. La maggior diffusione avviene attraverso internet e più nello specifico attraverso le e-mail. In ambiente mobile però le app dannose possono nascondersi anche all'interno di applicazioni che all'apparenza sembrano non rappresentare una minaccia, questo avviene soprattutto se ci si affida per il download a store non ufficiali. I tipi di malware più diffusi sono:

- *Virus*: programmi presenti in applicazini che una volta eseguite diffondono il codice malevole ad altri programmi del sistema;
- *Trojan*: codice che solitamente è nascosto in applicazioni che risultano utili all'utente, ma che una volta istallate consentono agli attaccanti di ottenere l'accesso al dispositivo;

- *Ransomware*: impediscono all'utente di accedere al proprio dispositivo cifrando i suoi file, spesso sono seguiti da una richiesta di riscatto per riottenere l'accesso al dispositivo;
- *Worm*: si diffondono nei dispositivi di una rete danneggiandoli mediante la distruzione di dati e file;

Da un'Indagine condotta da AV-Test, i trojan sono risultati il mezzo preferito dai criminali informatici per introdurre codice malevolo rappresentando il 93.93% di tutti gli attacchi di malware sui sistemi Android. Il ransomware si è classificato al secondo posto, con il 2,47% [9].

Detection

La detection di malware Andorid avviene attraverso l'utilizzo di diverse tecniche, le quali possono essere classificate in due gruppi principali:

- *Analisi statica*: alcune funzionalità del file binario, nel nostro caso il file `classes.dex` dell'applicazione, vengono estratte ed analizzate utilizzando diversi approcci come, ad esempio, l'utilizzo del machine learning;
- *Analisi dinamica*: il malware viene rilevato durante l'esecuzione andando a monitorare delle tracce di funzionalità utilizzate per arrecare danno.

Spesso ci si sofferma sul controllo degli accessi alle risorse fornite ad un'applicazione per stabilire il livello di rischio. Android tuttavia non impone nessun controllo, una volta forniti i permessi su come le applicazioni scambino le informazioni.

3.1.1 Colluding

Andorid utilizza diversi sistemi per la sicurezza e la privacy dell'utilizzatore. Un'applicazione per richiedere l'accesso a delle informazioni sensibili o all'utilizzo di una componente hardware (come fotocamera, microfono...) deve chiedere un'autorizzazione all'utente. Queste autorizzazioni vanno definite

nel file manifest. Inoltre è implementato anche un meccanismo di sandboxing, facendo in modo quindi che l'applicazione utilizzi solamente le risorse di cui necessita e che l'esecuzione avvenga in un ambiente chiuso. Per applicazioni colludenti si intendono tutte quelle applicazioni che implementano codice malevolo e che utilizzano covert channel¹ per eseguire azioni malevole. Le applicazioni colludenti sono dunque in comunicazione tra di loro attraverso dei canali nascosti e questo dà la possibilità di poter sviluppare il codice malevolo in più parti (snippet), codice che sarà inserito in applicazioni differenti. In questo modo si possono eludere tutti quei meccanismi di analisi statica che ricercano malware all'interno di ogni singola applicazione, in quanto la rilevazione di questo tipo di applicazioni oltre a dover valutare la minaccia alla sicurezza di ogni applicazione, comporta anche verificare l'esistenza di una comunicazione tra processi di app differenti durante un attacco informatico. Le applicazioni colludenti possono essere suddivise in due gruppi: nel primo troviamo tutte quelle app utilizzate per testare il rilevamento di malware ed il grado di protezione del dispositivo, nel secondo abbiamo tutte quelle app che vengono utilizzate per esplorare tutti i diversi covert channel [10]. Un attacco che avviene attraverso il *Colluding* può eseguire qualsiasi tipo di attacco (DoS, furto di informazioni...), inoltre può svolgere un'attività di supporto ad un attacco in atto andando ad aumentarne l'impatto. Il principale obiettivo degli attacchi di Colluding è evitare le restrizioni imposte dall'ambiente sandbox di Android in modo da rendere più complessa la rilevazione dell'attacco in atto. Supponendo quindi di avere due applicazioni colludenti che utilizzino covert channel, per effettuare la detection di questo tipo di attacco, bisogna combinare le due applicazioni colludenti in modo da racchiudere gli snippet che implementano la comunicazione delle componenti, all'interno di uno stesso Apk. L'applicazione risultante è valida ai fini dell'analisi statica in quanto tutti i componenti (activity, services, manifest...) sono riuniti in un unico pacchetto e dunque i canali di comunicazione utilizzati per l'attacco ora riguardano la stessa applicazione. Tuttavia questo

¹Un covert channel - canale nascosto - è un tipo di attacco informatico che sfrutta canali di comunicazione nascosti per scambiare oggetti tra processi che nella norma non dovrebbero essere in grado di comunicare.

potrebbe portare ad un'errata istallazione su un dispositivo [10].

Per una specifica degli attacchi del dataset utilizzato in questo lavoro si rimanda al paragrafo 4.1.3

Capitolo 4

La metodologia

In questo capitolo esporremo la metodologia utilizzata per la classificazione delle applicazioni Android, attraverso file audio. Nello specifico partiremo dai software e dai tipi di dati utilizzati per poi muoverci verso l'estrazione delle feature, la generazione dei dataset e la classificazione attraverso una particolare tecnica di machine learning chiamata multiple instance learning (MIL) come mostrato in figura 4.1.

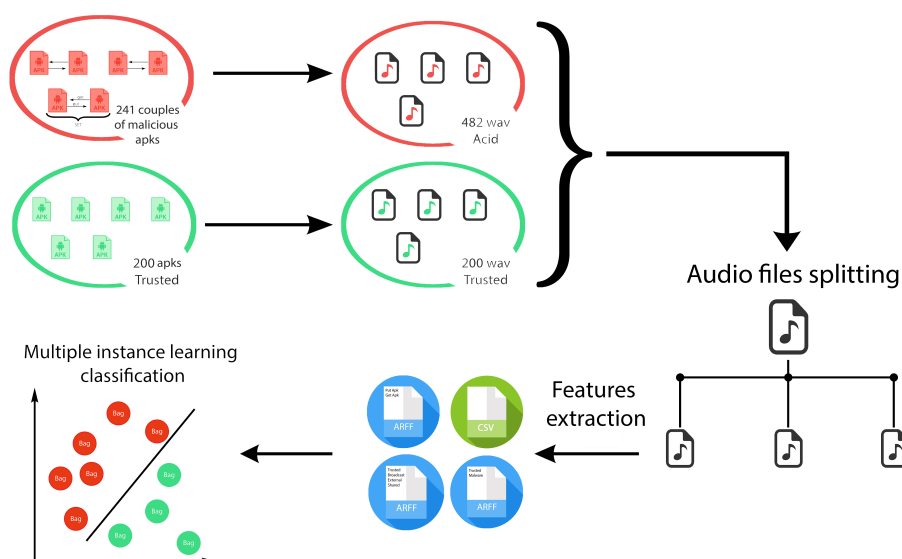


Figura 4.1: Methodology of development

4.1 I software e i dataset

In questo paragrafo esporremo dapprima tutti i software utilizzati nell'analisi e una breve panoramica sulle caratteristiche principali dei dataset utilizzati.

4.1.1 WEKA

Acronimo di "Waikato Environment for Knowledge Analysis", è un software open source per il machine learning. Partendo da un dataset¹ è possibile applicarvi dei metodi di apprendimento automatico e di analizzarne il risultato. È inoltre possibile, attraverso l'utilizzo di questi metodi, avere una previsione su nuovi set di dati. Per poter utilizzare gli algoritmi di classificazione del multiple instance learning bisogna importare i relativi package, attraverso il tool "Package Manager" già presente di default nella schermata iniziale di weka. Figura 4.2.

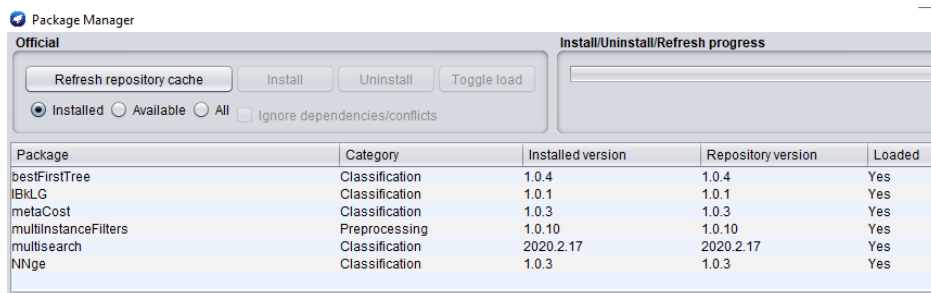


Figura 4.2: Input Mil package in weka

Una particolarità di questo software è l'utilizzo di dataset .arff.

4.1.2 dataset.csv e dataset.arff

I dataset utilizzati nel progetto sono dataset con estensione **.csv - Comma Separated Values**, questo è un formato di file di testo in cui ogni riga rappresenta un record della tabella. Ogni colonna invece rappresenta dei valori associati ad ogni record. Le colonne sono separate da virgole, da qui il nome. In figura 4.4 è possibile osservarne un esempio.

¹Collezione di dati organizzati, la grandezza è data dal numero di righe.

Il secondo tipo di dataset che abbiamo utilizzato sono file dati con estensione **.arff - Attribute Relationship File Format** come suggerisce il nome, questo formato di file organizza i dati seguendo una logica relazionale. La formattazione del dataset utilizzata è stata realizzata in un'ottica di classificazione attraverso algoritmi di multiple instance learning, dunque si è reso necessario dover organizzare i dati in bag. L'inizializzazione del file, per una classificazione MIL² può essere suddiviso in cinque componenti [11]:

1. Nella prima va sempre definita la relazione che lega i dati attraverso l'attributo **@relation** ed un nome che descriva quello che vogliamo predire.
2. Successivamente andranno inseriti gli identificativi delle bag, ovvero utilizzando **@attribute bag_id {...}** si vanno ad inserire nelle parentesi graffe la lista di tutti gli identificativi delle istanze della bag. Ogni identificativo va separato dall'altro tramite l'utilizzo di una virgola.
3. Dopo si definiranno gli attributi della bag, per farlo si utilizza **@attribute bag relational** per definire l'inizio della bag ed **@end bag** per definire la fine della bag. All'interno, tra i due attributi, vanno specificati gli attributi che compongono le istanze di una bag, ovvero le caratteristiche dei dati. Per farlo si utilizza ancora una volta l'identificativo **@attribute nome_caratteristica tipo_caratteristica**. Il tipo di caratteristica può essere *numeric* se il valore del dato è un numero intero, altrimenti *real* se il tipo di attributo è un numero reale. Se invece il tipo di attributo è una stringa (nel caso di un attributo booleano), scriveremo i valori ammissibili tra parentesi graffe es. {yes, no}.
4. A questo punto dobbiamo definire la classe che rappresenta una istanza. Per farlo inseriremo i valori tra parentesi graffe definendo l'attributo class come, **@attribute class {class1, class2}**
5. Infine a capo della key **@data** inseriremo il dataset correttamente formattato nel seguente modo: `iesima bag_id + ','` poi bisogna definire

²Mil - Multiple instance learning

tutte le istanze della bag. Una bag si definisce all'interno delle virgolette "...". Inseriamo all'interno tutte le istanze ognuna della quali sarà separata dal carattere speciale '\n'. A sua volta ogni istanza è rappresentata dai diversi attributi, tanti quanti ne abbiamo definiti in precedenza. Ogni attributo d'istanza è separato dall'altro tramite ','. Infine, dopo la chiusura delle virgolette, inseriamo la virgola e va definita la classe della bag. Ogni riga ha quindi la seguente formattazione:

```
bag_id , " attr1Ist1, attr2Ist1, attr3Ist1 \n attr1Ist2, attr2Ist2, attr3Ist2 \n attr1Ist3, attr2Ist3, attr3Ist3 " , classe
```

[illegible][illegible]

Figura 4.4: CSV file

Figura 4.3: ARFF dataset for MIL

4.1.3 Dataset di applicativi android

Siamo partiti da due dataset di applicazioni Apk. Il primo contenente un set di 200 applicazioni android non affette da malware definito come dataset "trusted", il secondo dataset invece contiene 241 coppie di Apk colludenti, abbiamo definito questo dataset come "Acid". Il dataset Acid è stato realizzato attraverso l'utilizzo del tool ACE (Collusion Application Engine)[10]. Quest'ultimo dataset, come detto, comprende 241 set ognuno dei quali include due package Apk. Nello specifico, queste due applicazioni contenenti il codice malevolo, comunicano tra loro attraverso lo scambio nascosto di informazioni

e vengono quindi dette *colludenti*. Difatti un'applicazione è di tipo "PUT" e l'altra è di tipo "GET". Per un totale di 482 applicazioni colludenti, si hanno 241 applicazioni di tipo "GET" e 241 di tipo "PUT". Ad esempio un applicativo di tipo "GET" recupera le informazioni dei contatti di un dispositivo attraverso uno snippet, le concatena in una stringa e provvede all'invio, ad esempio, attraverso l'utilizzo di external storage, all'applicativo di "PUT". Questo sarà in grado di estrarle e di provvedere all'invio delle informazioni ad un server remoto, portando al termine così un furto di dati. In figura 4.5 è possibile osservare le risorse attaccate e sottratte dalle applicazioni generate.

Snippet	Description
Accounts	Returns device accounts information
Bookmarks	Returns the browser bookmarks
Call log	Returns the call log
Contacts	Returns the list of contacts
History	Returns the navigation history
IMEI	Returns the device IMEI
Microphone	Records audio for 5 seconds
WiFi	List of WiFi SSID networks
Tasks	List of processes being executed

Figura 4.5: Stolen resources

I canali utilizzati per la comunicazione dalle applicazioni del dataset sono:

- *Broadcast_Intent*: in questo channel un'applicazione avvia un intent. La seconda applicazione attraverso un broadcast receiver intercetta l'intent.
- *Shared_Preferences*: le shared preferences non sono da considerare per una comunicazione inter-application, tuttavia se definiti i flag "WORLD_READABLE" o "WORLD_WRITABLE" è possibile procedere al salvataggio delle copie key-value accessibili globalmente. La prima app si occupa di salvare i valori e la seconda di andare a leggerli ed inoltrarli ad un web-server attraverso una richiesta HTTP POST.

- *ExternalStorage*: le applicazioni utilizzano canali di external storage per effettuare lo scambio di informazioni.

Le informazioni relative ad ogni set, il tipo di attacco e la risorsa utilizzata, sono descritti in un file `description.txt`.

4.2 Elaborazione dei file audio

In questo paragrafo descriveremo l'elaborazione dei file audio, ovvero la loro generazione/conversione e la successiva suddivisione.

4.2.1 Generazione dei file audio

Lo script realizzato per la conversione di un applicativo android in un file audio, una volta inserito nella directory che contiene i due dataset di Apk, va dapprima ad individuare tra i tutti i file che compongono un dataset i soli package .apk, successivamente va a decomprimere ogni pacchetto e ad estrarre da ognuno il file `classes.dex`. Questo file viene elaborato e convertito in un file audio con estensione Wav. L'output è salvato in una cartella creata ad hoc per raccogliere tutti i file audio generati, suddividendo i file audio generati da applicativi "trusted" da quelli di tipo "Acid", creando una directory di output differente a seconda del tipo.

4.2.2 Splitting dei file audio

Il secondo script sviluppato chiede in input la durata dello split in secondi, che deve essere maggiore di 0. Controllando che non sia stata creata in precedenza, va ad inizializzare una cartella "Splitted" tramite una funzione "createNewDirectory" [Listing:4.1].

```

1 def createNewDirectory(path=os.getcwd(), nameNewDirectory=""):
2     pathNewDirectory = path + "\\ " + nameNewDirectory
3     if not os.path.exists(pathNewDirectory):
4         try:
5             os.mkdir(pathNewDirectory)
6             print("Directory created successfully")
7             return pathNewDirectory

```



```

8         except OSError as error:
9             print("Directory can not be created")
10    else:
11        return pathNewDirectory

```

Listing 4.1: Create new directory function

Proseguendo, va ad iterare su tutti i file audio Wav presenti nelle cartelle "trusted" ed "Acid" generate precedentemente dalla conversione delle applicazioni. [Listing:4.2]

```

1 for wav_file in os.listdir(unsplit_audio_folder):
2     if wav_file.endswith(".wav"):
3         wavSplittedDirectory = createNewDirectory(
4             Path_SubDirectory, str(wav_file))
5         pathToOriginalWav = unsplit_audio_folder + "\\\" +
6         str(wav_file)
7         splittingWav(pathToOriginalWav,
8             wavSplittedDirectory)

```

Listing 4.2: Iteration over whole audios

Per ogni file Wav trovato si va a creare una cartella in "*Splitted\nameFileAudio*" dove il nome della directory è dato dal file .wav da splittare. Questa nuova cartella sarà poi popolata con i file splittati risultanti. In questo modo, ogni file .wav avrà una sua directory che conterrà gli split di output. Lo split avviene in una funzione "splittingWav" che va a calcolare il numero degli split in cui suddividere il file audio originale. Il calcolo avviene tramite una divisione per eccesso, in questo modo si includono anche suddivisioni più piccole del numero dato in input. Successivamente, iterando sul numero di splitting da effettuare, crea la suddivisione partendo, di volta in volta, dal file della durata intera, come si può osservare in figura 4.7 e 4.6.

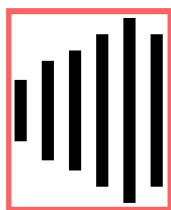


Figura 4.6: first split

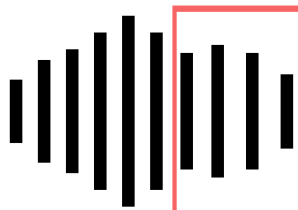


Figura 4.7: second split

La libreria utilizzata per effettuare lo splitting dei dati è "pydub"[12]. Nel codice le variabili @t_start e @t_stop rappresentano per ogni iterata il punto d'inizio e quello di fine dell'i-esimo split del file. Il nome dei file splittati corrispondono all'intervallo dell'i-esimo split (in millisecondi).

```

1 def splittingWav(pathToOriginalWav , pathToSplittedWav):
2     #Load
3     originalAudio = AudioSegment.from_wav(pathToOriginalWav)
4     numberOfSplit = math.ceil(originalAudio.duration_seconds /
5     splittingduration)
6     t_start = 0
7     for i in range(numberOfSplit):
8         t_stop = (t_start + (splittingduration * 1000))
9         splittedAudio = originalAudio[t_start:t_stop]
10        splittedAudio.export(out_f=pathToSplittedWav + "\\ " + str(t_start) +
11        " - " + str(t_stop) + '.wav',format="wav")
12        t_start = t_stop

```

Listing 4.3: Splitting Wav function

4.3 Estrazione delle features

Al fine di poter effettuare una classificazione abbiamo bisogno di attributi che possano rappresentare un dato oggetto, nel nostro caso un audio digitale. Il suono in natura è un segnale continuo, dunque per essere memorizzato deve essere campionato ottenendo, in questo modo, un segnale digitale rappresentato da valori numerici che vadano ad approssimare il più possibile la forma dell'onda. Ogni campione è formato da un dato numero di bit e viene prelevato con ritmo costante dal suono. Una frequenza di campionamento rappresenta il numero di campioni prelevati in un secondo ed è un numero

che di solito oscilla tra gli 8000 ed i 44100 samples al secondo.

Lo scopo dello script per l'estrazione delle feature è quello di andare a generare tre dataset:

- *data.arff*: un dataset .arff, che viene utilizzato per la classificazione, contenente tutte le feature estratte per ogni singolo audio splittato, organizzando le istanze per la classificazione tramite multiple instance learning come descritto nel paragrafo [4.1.2]. In questo dataset l'attributo **class** assume uno dei valori, "trusted", "broadcast_intent", "shared_preferences" o "external_storage". Per estrarre questi attributi si va a leggere il contenuto del file di accompagnamento "description.txt" all'interno di ogni set Acid. Nel caso si stia analizzando un file audio generato da un Apk trusted, l'attributo verrà semplicemente impostato come "trusted".

```
1      with open(f'{acidDatasetFolder}\\{setFold}\\{description.txt}',  
2              'r') as reader:  
3          malwareType = reader.read().split(":")[1]  
4          classe = malwareType
```

Listing 4.4: Get android resources

- *dataBinary.arff*: questo dataset, utilizzato per la classificazione, contiene tutte le feature estratte, ancora una volta, per ogni singolo audio splittato, organizzando i dati in bag. La differenza consiste nell'andare ad inserire come attributi della **class** solamente i valori, "trusted", "malware".
- *data.csv*: il dataset.csv comprende solamente le applicazioni provenienti dal dataset "Acid". Le tre colonne che compongono la tabella sono riempite nel seguente modo. Nella prima colonna si inseriscono i nomi delle bag_id, nella seconda colonna le features delle varie istanze della bag ed infine, nella terza colonna, l'attributo **class** che può assumere uno dei valori, "trusted", "broadcast_intent", "shared_preferences" o "external_storage". Questo dataset sarà utilizzato per generare un quarto dataset "dataGetPut.arff" come vedremo nel paragrafo 4.3.

Lo script realizzato per estrarre le features, inizializza dapprima i file dataset di cui sopra che saranno poi utilizzati per organizzare le features. Successivamente, lo script provvede ad iterare nelle cartelle "Acid_Splitted" e "Trusted_Splitted" il cui contenuto sono le subdirectory con gli audio split-tati generati dallo script precedente [4.2.1]. Per ogni audio contenuto nelle subdirectory si procede quindi all'estrazione delle features con l'ausilio della libreria "librosa"[13]. Attraverso:

```
1 y, sr = librosa.load(splittedAudioPath, mono=True, duration=30)
```

Si va a caricare il file audio in formato mono. La variabile "y" rappresenta la serie temporale dell'audio, dunque $y[t]$ corrisponde all'ampiezza della forma dell'onda al campione t -esimo. Mentre la variabile "sr" rappresenta la frequenza di campionamento di "y". Successivamente attraverso le istruzioni:

```
1 chroma_stft = librosa.feature.chroma_stft(y=y, sr=sr)
2 spec_cent = librosa.feature.spectral_centroid(y=y, sr=sr)
3 spec_bw = librosa.feature.spectral_bandwidth(y=y, sr=sr)
4 rolloff = librosa.feature.spectral_rolloff(y=y, sr=sr)
5 zcr = librosa.feature.zero_crossing_rate(y)
6 mfcc = librosa.feature.mfcc(y=y, sr=sr)
```

- `chroma_stft`: calcola a partire dall'onda un cromogramma e ritorna un valore normalizzato di ogni frame.
- `spectral_centroid`: calcola il centroide spettrale. L'intensità di un suono in funzione del tempo e della frequenza può essere rappresentato graficamente da uno spettrogramma. Ogni fotogramma di uno spettrogramma viene normalizzato e ne viene estratta la media (centroide).
- `spectral_bandwidth`: calcola la frequenza della larghezza di banda di ogni frame.
- `spectral_rolloff`: calcola per ogni frame la frequenza di rolloff.
- `zero_crossing_rate`: calcola i passaggi per lo zero in una serie temporale.
- `mfcc`: genera una sequenza di coefficienti cefalici mfcc da una serie temporale.

Per ognuno di questi valori è stata estratta la media ed il risultato inserito nella stringa che poi comporrà una riga del dataset. All'interno del codice sono inseriti diversi controlli per far sì che la stringa rispetti la formattazione dei file di tipo ARFF. Infine la stringa, composta dal nome dell'audio (integro), dalle feature di tutti i singoli file audio (splittati) e dalla classe d'appartenenza, viene scritta in una nuova riga del file dataset corrispondente.

```

1         .....
2         if (featureOfBag == 1):
3             featureOfBag += 1
4             to_append_arff = f'{{audioTitle}}".wav"{{","}}\{{np.mean(chroma_stft)
5             }}{{","}}\{{np.mean(spec_cent)}}{{","}}\{{np.mean(spec_bw)}}{{","}}\{{np.mean(rolloff)
6             }}{{","}}\{{np.mean(zcr)}}{{","}}\{{
7
8         indexMfcc = 1
9         for e in mfcc:
10             if (indexMfcc == len(mfcc)):
11                 to_append_arff += f'{{np.mean(e)}}\{{
12                 to_append += f' {{np.mean(e)}}\{{
13             else:
14                 to_append_arff += f'{{np.mean(e)}}{{","}}\{{
15                 to_append += f' {{np.mean(e)}}{{","}}\{{
16             indexMfcc += 1
17         if splittedAudio != latestSplitter:
18             to_append += "\n"
19
20         .....
21
22         fileArff = open('results\\data.arff', 'a', newline='')
23         fileArff.writelines(to_append_arff)
24         fileArff.close()

```

Listing 4.5: Arff formatting string

Ogni dataset così generato è composto da 682 istanze che individuano le varie bag, con annesse istanze. Ogni bag corrisponde ad una applicazione.

Creazione del quarto dataset

Per generare il quarto dataset **dataGetPut.arff** abbiamo realizzato uno script che lavora su un dataset "smistamento.csv" contenente lo smistamento delle applicazioni. Ovvero in questo dataset sono elencati i nomi di ogni applicativo appartenente al dataset "Acid" e per ognuno è specificato se l'applicativo è di tipo "PUT" o "GET".

Lo script quindi va a creare un nuovo file ARFF in cui verranno inserite le feature *dataGetPut.arff*. Questo dataset sarà utilizzato per la classificazione. Di seguito dato in input il dataset "data.csv", creato attraverso lo script precedente [4.3], va a controllare per ogni istanza del dataset (data.csv) il tipo di classe di appartenenza nel secondo dataset (smistamento.csv) attraverso la ricerca di un'uguaglianza del nome dell'applicativo. Infine va a scrivere nel datasetGetPut.arff creato le istanze, le features e la tipologia rilevata.

```
1     for index, row in dataset_csv.iterrows():
2         typeWav = row[2]
3         nameWav = row[0]
4         if typeWav == "trusted":
5             continue
6         else:
7             type = str(getTypeOfApp(str(row[0]).split(".")[0])).replace(" ", "_")
8         stringa = f'{nameWav}{"", "}"\{row[1]}\{"", "}"\{type}\n'
9
10        fileArff = open("results\\dataGetPut.arff", 'a', newline='')
11        fileArff.writelines(stringa)
12        fileArff.close()
```

Listing 4.6: Application Get or Put control

4.4 Il multiple instance learning

In questo paragrafo faremo una breve panoramica sul multiple instance learning. Prima però descriveremo cos'è il machine learning.

Machine learning

Il machine learning è quella branca dell'intelligenza artificiale che si occupa di individuare schemi nei dati al fine di addestrare dei modelli per eseguire

successivamente delle previsioni attraverso l'utilizzo di nuovi dati. I dati sono proprio il punto centrale di questa tecnologia, in quanto il progressivo aumento di disponibilità degli stessi ha reso necessario lo sviluppo di algoritmi che fossero in grado di catturarne la conoscenza. I tipi di machine learning sono tre:

1. **Apprendimento supervisionato:** questa tipologia si basa sull'addestramento di un modello. All'algoritmo viene dato in input il dataset da analizzare che comprende l'output atteso. I compiti possono dividersi in due sottocategorie: classificazione e regressione.

- *Classificazione:* l'obiettivo è, sulla base di osservazioni svolte in precedenza, quello di andare ad effettuare una predizione delle label di classi di nuove istanze. Le label sono un particolare valore discreto, che rappresenta in qualche modo un gruppo di dati. Dunque, il ruolo svolto dalla classificazione è quello di discriminare l'appartenenza di un set di dati ad una delle classi. A sua volta la classificazione può essere suddivisa in altre sottocategorie. Nel caso in cui le classi su cui effettuare una predizione è composta da due label, si dice che stiamo effettuando una *classificazione binaria*, altrimenti se le classi sono più di due ci troviamo di fronte ad una *classificazione multiclasse*.
- *Regressione:* in questo caso sia l'input che l'output sono continui, difatti, dato in input un dataset, l'output sarà dato da un valore continuo. Ad esempio, attraverso la regressione lineare, data una variabile predittiva x ed una variabile risposta y , si va a calcolare la retta che va a minimizzare la distanza fra i punti e la retta [14].

2. **Apprendimento non supervisionato:** in questa categoria non sono note le classi del training set, la struttura dei dati può essere ignota. Per l'estrazione delle informazioni, dunque, non si utilizza una variabile nota né una funzione di ricompensa come vedremo nella prossima categoria. Un esempio è il *clustering* in cui si vanno a ricercare dei sottogruppi che individuino relazioni tra i dati.

3. **Apprendimento con rinforzo:** in questa categoria, l'algoritmo di apprendimento sviluppa un sistema che migliora le prestazioni andando ad imparare dagli errori che ha effettuato in precedenza. Quindi il focus è quello di andare a massimizzare la ricompensa attraverso un approccio di tipo esplorativo.

Come si è potuto notare tutti gli approcci necessitano di un "ingrediente fondamentale": i dati. Uno dei passi primari è il pre-processing dei dati come ad esempio la normalizzazione. Un ulteriore passaggio che va effettuato sul dataset, per la maggior parte degli algoritmi, è la suddivisione di quest'ultimo in un *training set* ed un *test set*. Il primo viene utilizzato per la fase di addestramento del modello predittivo, il secondo per valutare la capacità di quanto il modello sia in grado di effettuare una predizione utilizzando nuovi dati, viene detta attività di testing.

Multiple instance learning

Nel machine learning supervisionato tradizionale, ogni dataset è composto da delle righe che rappresentano le istanze i cui valori sono associati a degli attributi, le colonne. Ad ogni istanza è associata una label. Per alcune tipologie di dati l'etichetta può essere assegnata solamente ad un gruppo di istanze, non a tutte singolarmente. In questi casi bisogna utilizzare MIL³. Il Multiple instance learning rientra nell'apprendimento supervisionato. L'utilizzo di questa metodologia ha come caratteristica principale l'organizzazione delle istanze avviene tramite particolari set che vengono chiamati *bag*. Ad ognuna di queste bag viene associata una label, dunque un'etichetta riguarda tutte le istanze della bag [15]. Ad un algoritmo di apprendimento automatico vengono date in input più bag, ognuna delle quali è composta da più istanze. L'obiettivo del MIL è quello di prevedere la label di nuove bag non utilizzate nella fase di addestramento del modello. Lo sviluppo di questa metodologia si sta ampliando in quanto sono aumentati i dati disponibili e classificarli singolarmente, come avviene in un problema di classificazione standard, richiede uno sforzo maggiore. Attraverso il MIL si allevia questo onere [16]

³Abbreviazione di multiple instance learning

organizzando appunto i dati in bag contenenti le istanze ed assegnandole una label. Un apprendimento tramite MIL consente di affrontare sia problemi di classificazione che di regressione.

In questo lavoro le istanze che vanno a comporre una bag sono tutte le feature estratte da ogni audio splittato. Questo significa che ogni bag rappresenta una singola applicazione ed il suo contenuto è composto dall'insieme delle feature, ognuna estratta dal relativo file audio splittato dall'audio integro frutto della conversione dell'Apk in .wav. La classe di una bag è invece data dal tipo di applicazione, se trusted o malware come specificato al paragrafo 4.3

4.5 K-fold validation

La k-fold validation è una tecnica che va a dividere il dataset in k sezioni, utilizzando k-1 sezioni per il training e la k-ma sezione per il testing. La procedura viene poi ripetuta andando ad utilizzare una nuova sezione delle k calcolate precedentemente per il testing e le restanti k-1 per il training. Il tutto si ripete per k volte andando di volta in volta ad utilizzare un segmento differente per la fase di testing. Infine, tra tutte le osservazioni, si va a prendere la media. In WEKA l'algoritmo viene chiamato una k+1 esima volta sull'intero testi di dati [17]. Questa tecnica è utile per dataset non molto popolati.

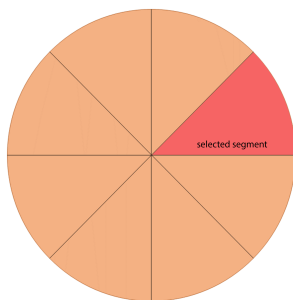


Figura 4.8: k th selected segment

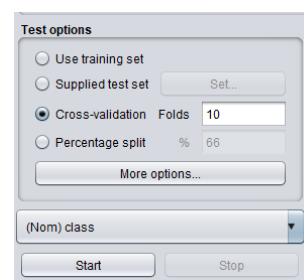


Figura 4.9: K-fold in weka

4.6 Precision e Recall

Per valutare la bontà di una predizione da parte di un modello di machine learning si utilizzano varie metriche. Tra le più utilizzate troviamo la precision e la recall. Distinguiamo in un processo predittivo i false positive, ovvero quelle istanze etichettate come appartenenti ad una classe, ma che in realtà non lo sono ed i true positive, ovvero quelle istanze etichettate come appartenenti ad una classe e che lo sono realmente.

- **Precision:** è data dal rapporto tra il numero di true positive ed il numero totale di elementi etichettati come appartenenti ad una classe, dunque la somma di veri e false positive. Un modello preciso genera pochi false positive, perché quando prevede l'appartenenza ad una classe raramente sbaglia, mentre i falsi negativi potrebbero essere molti non prevedendo quindi tutte le istanze che dovrebbero appartenere ad una classe. Dunque la precision peggiora se vi sono tanti false positive.
- **Recall:** questa metrica è data dal rapporto tra il numero di true positive ed il numero di istanze che realmente vi appartengono. Un alto valore di recall indica che il modello recupera tutte le istanze che appartengono alla classe. Tuttavia potrebbero esserci molti false positive. Dunque la recall peggiora se vi sono tanti falsi negativi.

Spesso accade che al migliorare della precisione si assiste ad un peggioramento della recall e viceversa. Bisogna dunque trovare un modello che cerchi un giusto compromesso.

Capitolo 5

Sperimentazioni

In questo capitolo verranno riportati i risultati osservati nella fase di classificazione dei dataset ottenuti dalle features dai file audio splittati, ottenuti a partire dalle applicazioni Andorid. Dapprima vedremo come sono composti i dataset ed in seguito quali sono i risultati ottenuti dalle classificazioni. Il processo di addestramento di modelli di multiple instance learning è avvenuto attraverso l'utilizzo del software WEKA [4.1.1]. La bontà delle classificazioni effettuate si è calcolata dalle due metriche precision e recall sopra descritte [4.6]. Nel processo di training di ogni modello di classificazione utilizzato si è impiegato l'utilizzo della K-Fold Validation [4.5], nel dettaglio la divisione e ripetizione è avvenuta sul 10% del dataset, impostando il valore $K = 10$, di conseguenza i valori di precision e recall rappresentano la media aritmetica delle 10 iterazioni eseguite durante l'addestramento.

5.1 Classificazione multiclasse

Come detto in precedenza, al paragrafo 4.1.3, le applicazioni utilizzate provengono da due dataset "trusted" e "Acid". Le applicazioni Acid sono di diverso tipo, "broadcast_intent", "shared_preferences" o "external_storage". Dopo aver effettuato una prima conversione delle applicazioni Android in file audio di tipo Wav, lo splitting degli stessi è stato effettuato in due passaggi. Nella prima operazione di splitting, la suddivisione è stata effettuata in

intervalli di circa 35 minuti (2092 secondi), ricavando quindi, dall'estrazione delle feature da ogni file audio suddiviso, un dataset "*data_2092.arff*" di 682 bag, il maggior numero delle quali composte da circa 8 istanze. Nella seconda operazione di splitting, la suddivisione è stata effettuata in intervalli di circa 17 minuti (1046 secondi) in questo modo il numero di bag che vanno a comporre il dataset "*data_1046.arff*" è come prima di 682 bag, la maggior parte delle quali popolata però da 16 istanze. In figura 5.1 si può osservare la suddivisione delle classi.

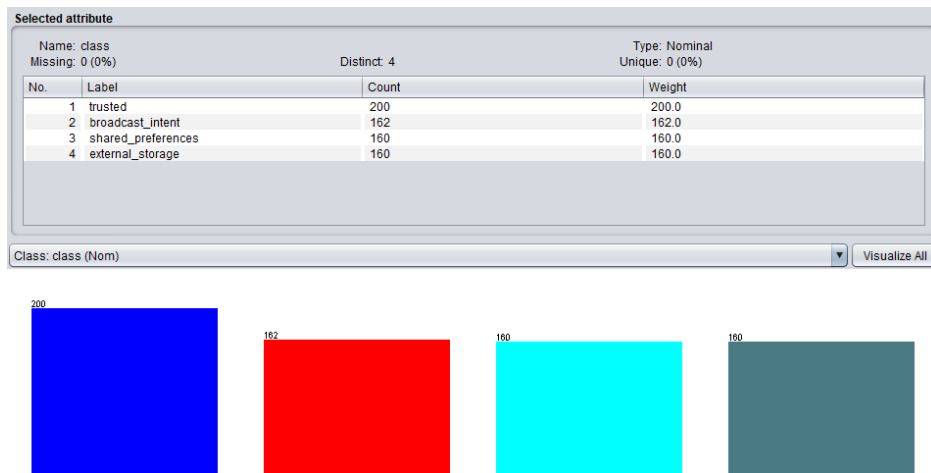


Figura 5.1: Multi class dataset

La classificazione è stata eseguita adottando il criterio della K-Fold validation [4.5]. Nello specifico l'algoritmo utilizzato è stato il TLC che ha restituito per il dataset "**data_2092.arff**":

- Precision: 0.847
- Recall: 0.845

mentre per il dataset "**data_1046.arff**" i valori osservati sono:

- Precision: 0.844
- Recall: 0.845

In figura 5.2 si può osservare come le istanze del test set sono state classificate. Nello specifico sulle ascisse è riportata la classe predetta dal modello,

mentre sulle ordinate, la classe di appartenenza originale. Alle istanze è stata applicata Jitter per una migliore visualizzazione. Le bag rappresentate da una 'x' sono state predette correttamente, ovvero la classe coincide con la classe predetta dal modello, mentre le bag rappresentate dal quadratino sono quelle erroneamente classificate dal modello, la cui classe d'appartenenza è la corrispondente sull'asse delle ordinate che dà anche il colore alle istanze, mentre la predetta è la classe corrispondente sulle ascisse.

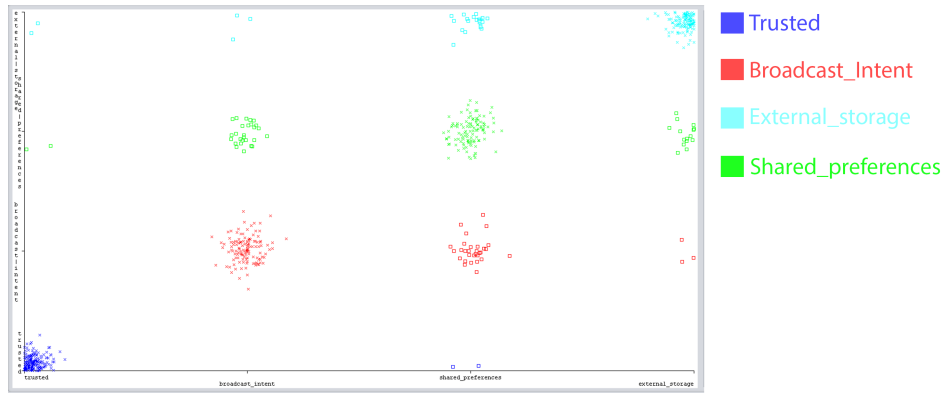


Figura 5.2: Multi class classification

5.2 Classificazione binaria Trusted - Malware

La seconda sperimentazione effettuata riguarda la classificazione dei dataset le cui classi d'appartenenza sono "trusted" e "malware". I dataset popolati da 682 bag sono stati ricavati da istanze di audio splittati in circa 35 minuti (2092 secondi) da cui si è ottenuto il dataset "dataBinary_2092.arff" e da audio suddivisi in intervalli di circa 17 minuti (1046 secondi) da cui abbiamo ricavato il dataset "dataBinary_1046.arff". Le classi delle bag sono composte da 200 bag con label "trusted" e 482 bag con label "malware" come si può osservare dall'istogramma in figura 5.3.

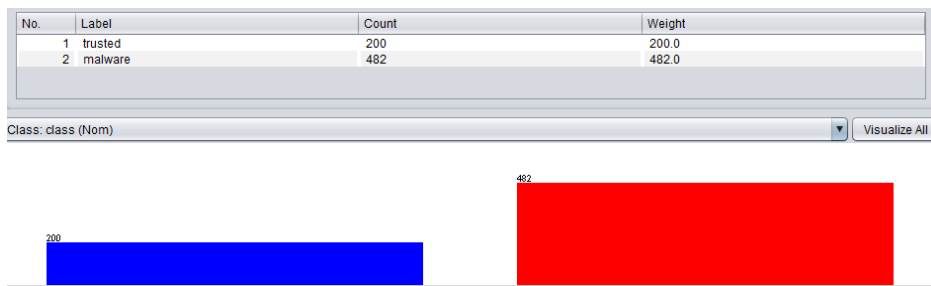


Figura 5.3: Binary class dataset

Le classificazioni hanno riguardato diversi algoritmi, i risultati, valutati attraverso i valori di precision e recall sono stati:

Dataset dataBinary_2092		
ALG	PRECISION	RECALL
MIEMDD	0.999	0.999
MIDD	0.997	0.997
QUICK DD	0.996	0.996
TLC	0.993	0.993
MITI	0.981	0.981
MIRI	0.980	0.979
MILR	0.949	0.946
CitationKNN	0.924	0.915
MDD	0.852	0.818
MISVM	0.806	0.733
TLD	0.79	0.299

Dataset dataBinary_1046		
ALG	PRECISION	RECALL
MIEMDD	0.997	0.997
TLC	0.996	0.996
QUICK DD	0.991	0.991
MITI	0.913	0.902
MIRI	0.913	0.902
MILR	0.981	0.981
CitationKNN	0.916	0.905
MDD	0.915	0.905
MISVM	0.802	0.724

Dunque il risultato migliore si è ottenuto con l'algoritmo MIEMDD in entrambi i dataset.

5.3 Classificazione binaria Apk_Get - Apk_Put

In questa sperimentazione, i dataset comprendono solo le applicazioni affette da malware provenienti dal dataset "Acid". Le bag di ogni dataset sono quindi in totale 482, suddivise in due classi Apk_Get e Apk_Put come descritto nel paragrafo 4.1.2. Nella figura 5.4 possiamo osservare l'istogramma relativo alle due classi sopra descritte.

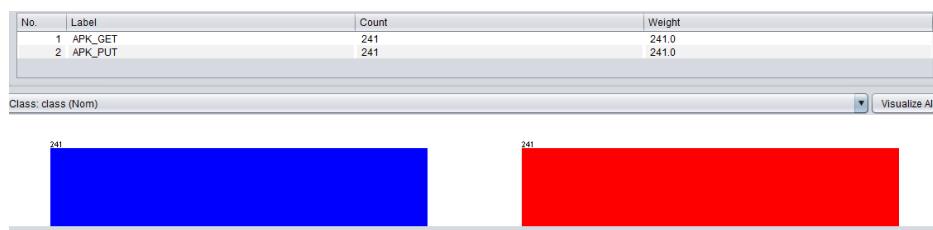


Figura 5.4: Get - Put class dataset

Anche in questo caso la classificazione ha ricoperto l'utilizzo di più algoritmi. La classificazione è stata eseguita sui due dataset "dataGetPut_2092.arff" e "dataGetPut_1046.arff" e la valutazione eseguita sulla bontà dei risultati di precision e recall. I valori osservati sono stati i seguenti:

Dataset dataGetPut_2092		
ALG	PRECISION	RECALL
TLC	0.979	0.979
MITI	0.971	0.971
MIRI	0.965	0.965
MIDD	0.878	0.861
MDD	0.804	0.712
TLDSIMPLE	0.755	0.519
MIEMDD	0.751	0.502
QUICKDD	0.729	0.726
MILR	0.599	0.562
BOOST	0.498	0.498
WRAPPER	0.498	0.498
CitationKNN	0.497	0.498
SIMPLEMI	0.494	0.498

Dataset dataGetPut_1046		
ALG	PRECISION	RECALL
TLC	0.992	0.992
MITI	0.979	0.979
MIRI	0.975	0.975
MIDD	0.913	0.913
MDD	0.805	0.703
TLDSIMPLE	0.754	0.515
MIEMDD	0.700	0.618
QUICKDD	0.869	0.838
MILR	0.574	0.573
BOOST	0.498	0.498
WRAPPER	0.498	0.498
CitationKNN	0.497	0.498
SIMPLEMI	0.494	0.498

Nel classificare questi dataset il risultato migliore è stato ottenuto con l'utilizzo dell'algoritmo TLC.

Capitolo 6

Conclusioni e sviluppi futuri

Conclusioni

L'obiettivo di classificare le applicazioni colludenti attraverso metodi di multiple instance learnign, come si può osservare dal capitolo [5] relativo alle sperimentazioni, ha prodotto risultati soddifacenti. Nello specifico, possiamo vedere come uno degli algorimti che ha prodotto risultati più soddifacenti sia stato il TLC (two-level classification). In figura 6.1 possiamo osservare parte del modello generato per il dataset "dataGetPut1046.arff".

```

=== Classifier model (full training set) ===

Partition Generator:

J48 pruned tree
-----

bag_mfcc3 <= -47.668896: APK_PUT (179.66)
bag_mfcc3 > -47.668896
|   bag_mfcc18 <= 0.029074
|   |   bag_mfcc4 <= -36.702797: APK_PUT (129.79)
|   |   bag_mfcc4 > -36.702797
|   |   |   bag_mfcc5 <= -36.423157: APK_PUT (88.0/1.0)
|   |   |   bag_mfcc5 > -36.423157
|   |   |   |   bag_mfcc5 <= -36.421864
|   |   |   |   bag_mfcc18 <= 0.025458
|   |   |   |   |   bag_mfcc5 <= -36.422176: APK_PUT (7.02/1.0)
|   |   |   |   |   bag_mfcc5 > -36.422176: APK_GET (79.27)
|   |   |   |   |   bag_mfcc18 > 0.025458: APK_GET (80.27)
|   |   |   |   bag_mfcc5 > -36.421864
|   |   |   |   |   bag_mfcc1 <= -42.610081
|   |   |   |   |   bag_mfcc17 <= -22.454451
|   |   |   |   |   bag_mfcc3 <= -46.919662
|   |   |   |   |   |   bag_mfcc20 <= -1.339326: APK_PUT (58.2)
|   |   |   |   |   |   bag_mfcc20 > -1.339326
|   |   |   |   |   |   |   bag_mfcc4 <= -35.941315
|   |   |   |   |   |   |   |   bag_chroma_stft <= 0.12741: APK_PUT (5.02)
|   |   |   |   |   |   |   |   bag_chroma_stft > 0.12741: APK_GET (8.03/1.0)
|   |   |   |   |   |   |   |   bag_mfcc4 > -35.941315
|   |   |   |   |   |   |   |   |   bag_mfcc17 <= -22.67732
|   |   |   |   |   |   |   |   |   |   bag_mfcc1 <= -42.648254: APK_GET (13.04/1.0)
|   |   |   |   |   |   |   |   |   |   bag_mfcc1 > -42.648254: APK_PUT (3.01)
|   |   |   |   |   |   |   |   |   |   |   bag_mfcc17 > -22.67732: APK_GET (145.49/5.02)
|   |   |   |   |   |   |   |   |   |   |   bag_mfcc3 > -46.919662: APK_GET (164.55/6.02)
|   |   |   |   |   |   |   |   |   |   bag_mfcc17 > -22.454451
|   |   |   |   |   |   |   |   |   |   |   bag_mfcc13 <= -14.072556: APK_PUT (179.07/2.01)
|   |   |   |   |   |   |   |   |   |   |   bag_mfcc13 > -14.072556
|   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc1 <= -63.343884
|   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc2 <= 165.682449
|   |   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc13 <= -11.998972: APK_PUT (70.71)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc13 > -11.998972
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc13 <= -11.973404
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc1 <= -78.887489: APK_GET (159.54)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc1 > -78.887489: APK_PUT (9.03)
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   bag_mfcc13 > -11.973404

```

Figura 6.1: Output TLC model

Sviluppi futuri

Gli sviluppi futuri saranno quelli di andare ad effettuare degli studi per la ricerca dell'istanza della bag che la rende la bag classificata come malware in modo da capire se è possibile verificare quale tra tutte le istanze della bag contengono le feature relative allo snippet malevolo. Altri studi riguarderanno la scelta delle features del modello, cercare di individuare quale feature ricorre più spesso per la classificazione.

Bibliografia

- [1] VentureBeat - Emil Protalinski, “Android passes 2.5 billion monthly active devices,” May 7, 2019, [Online; in data 11-maggio-2021]. [Online]. Available: <https://venturebeat.com/2019/05/07/android-passes-2-5-billion-monthly-active-devices/>
- [2] blog.Google, “S’more to love across all your screens,” Sep 29, 2015, [Online; in data 11-maggio-2021]. [Online]. Available: <https://blog.google/products/android/smores-to-love-across-all-your-screens/>
- [3] Wikipedia contributors, “Wav — Wikipedia, the free encyclopedia,” 2021, [Online; accessed 10-May-2021]. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=WAV&oldid=1020662001>
- [4] Statista, “Number of smartphone users worldwide from 2016 to 2023,” 2020. [Online]. Available: <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>
- [5] Wikipedia, “Integrated development environment — wikipedia, l’enciclopedia libera,” 2021, [Online; in data 10-maggio-2021]. [Online]. Available: https://it.wikipedia.org/w/index.php?title=Integrated_development_environment&oldid=118335792
- [6] Avira. Rapporto sulle minacce malware: statistiche e tendenze del secondo trimestre 2020. [Online]. Available: <https://www.avira.com/en/blog/malware-threat-report-q2-2020-statistics-and-trends>
- [7] G. C. AG. Rapporto malware mobile: nessun rallentamento con il malware android. [Onli-

- ne]. Available: <https://www.gdatasoftware.com/news/2019/07/35228-mobile-malware-report-no-let-up-with-android-malware>
- [8] SOFTPEDIA NEWS, “One billion android devices are out of date and won’t get further updates,” Nov 14, 2017, [Online; in data 11-maggio-2021]. [Online]. Available: https://news.softpedia.com/news/one-billion-android-devices-are-out-of-date-and-won-t-get-further-updates-518533.shtml#sgal_0
- [9] Statista - Av-Test - Joseph Johnson, “Distribution of android malware 2019,” Jan 25, 2021, [Online; in data 11-maggio-2021]. [Online]. Available: <https://www.statista.com/statistics/681006/share-of-android-types-of-malware/>
- [10] T. M. C. Jorge Blasco, “Blasco, j., chen, t.m. automated generation of colluding apps for experimental research. j comput virol hack tech 14, 127–138 (2018).” 2018. [Online]. Available: <https://doi.org/10.1007/s11416-017-0296-4>
- [11] Eibe Frank, Mark A. Hall, and Ian H. Witten (2016). Morgan Kaufmann, Fourth Edition, 2016., “The weka workbench. online appendix for data mining: Practical machine learning tools and techniques,” 2016, [Online; in data 11-maggio-2021]. [Online]. Available: https://waikato.github.io/weka-wiki/multi_instance_classification/
- [12] James Robert, <http://jiaaro.com>, “pydub,” 2011, [Online; in data 12-maggio-2021]. [Online]. Available: <https://github.com/jiaaro/pydub#installation>
- [13] McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto., “librosa: Audio and music signal analysis in python.” in proceedings of the 14th python in science conference, pp. 18-25.” 2015, [Online; in data 12-maggio-2021]. [Online]. Available: <https://librosa.org/doc/latest/index.html#id1>
- [14] S. RASCHKA, *Machine Learning con Python, costruire algoritmi per generare conoscenza*. Apogeo, 2019.

- [15] Wikipedia contributors, “Multiple instance learning — Wikipedia, the free encyclopedia,” 2020, [Online; accessed 13-May-2021]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multiple_instance_learning&oldid=972908596
- [16] M.-A. Carbonneau, V. Cheplygina, E. Granger, and G. Gagnon, “Multiple instance learning: A survey of problem characteristics and applications,” *Pattern Recognition*, vol. 77, p. 329–353, May 2018. [Online]. Available: <http://dx.doi.org/10.1016/j.patcog.2017.10.009>
- [17] WekaMOOC. Data mining with weka (2.5: Cross-validation). [Online]. Available: <https://www.youtube.com/watch?v=V0eL6MWxY-w>