UNIVERSITÀ DEGLI STUDI DI GENOVA
Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

RiskInDroid: analisi del rischio di applicazioni Android

Relatore:
Prof. Alessio MERLO

Candidato:
Gabriel Claudiu GEORGIU
Matricola n. 3618343

Anno Accademico 2015 – 2016

Simboli ed abbreviazioni

API

Application Programming Interface

CVE

Common Vulnerabilities and Exposures

CVSS

Common Vulnerability Scoring System

IDE

Integrated Development Environment

IPC

Inter Process Communication

NVD

National Vulnerability Database

SVM

Support Vector Machines

URL

Uniform Resource Locator

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Con quasi 300 milioni di smartphone venduti solamente nel secondo trimestre del 2016 [1], Android risulta essere il sistema operativo per dispositivi mobili più diffuso al mondo e, di conseguenza, anche un bersaglio appetibile per gli autori di *malware*, i quali possono sfruttare la sua popolarità per mettere in pericolo la privacy di milioni di utenti. È pertanto necessario prestare particolare attenzione agli aspetti di sicurezza legati alla piattaforma Android.

Con lo scopo di rendere l'utenza più consapevole del rischio in cui si incorre installando determinate applicazioni, questa tesi si prefigge l'obiettivo di proporre un metodo per l'analisi quantitativa del rischio delle app Android, volto a ricavare un punteggio numerico corretto ed affidabile che indichi quanto una certa app possa essere rischiosa. Dopo aver valutato due metodologie che si sono rivelate inadatte per una affidabile valutazione del rischio, viene ideato e proposto un nuovo metodo, denominato RiskInDroid, che si basa su tecniche di *machine learning* per costruire un modello di rischio partendo da un ampio campione statistico raccolto specificatamente per questo lavoro di laurea, costituito da 116 541 applicazioni benigne e 6 707 *malware*. Il funzionamento di RiskInDroid si fonda su un'analisi estensiva dei permessi delle applicazioni Android, e non soltanto sui permessi dichiarati (come nella maggior parte degli altri lavori scientifici), ma anche su quelli realmente utilizzati, richiesti ma non usati e infine utilizzati senza essere stati prima dichiarati.

Sebbene la letteratura scientifica che tratta argomenti riguardanti la sicurezza di Android sia molto vasta, finora si è focalizzata maggiormente sulla classificazione binaria di *malware* e non sono presenti molti lavori incentrati sulla creazione di un indice di rischio quantitativo. Le metodologie basate sulla classificazione binaria hanno il limite intrinseco di non poter fornire alcun valore quantitativo riferito alla pericolosità di un'app; per quanto riguarda invece le tecniche già proposte per la valutazione del rischio, alcune non sono applicabili in generale perché necessitano di conoscere la categoria di appartenenza dell'applicazione [2, 3], mentre in altri casi i valori di rischio non sono facilmente confrontabili perché non è stato fissato un intervallo limitato entro cui far variare tali valori [4]. RiskInDroid, il metodo proposto in questa tesi, prende spunto dalle metodologie basate sulla classificazione binaria di *malware*, integrando però le intuizioni più interessanti presenti nelle pubblicazioni riguardanti l'analisi di rischio delle applicazioni. In questo modo si

ottiene un approccio applicabile in maniera generale e un indice di rischio facilmente interpretabile e confrontabile con gli indici di rischio di altre app, superando dunque le limitazioni dei metodi proposti finora.

Per quanto riguarda la realizzazione pratica di RiskInDroid, si è deciso di impiegare il linguaggio di programmazione Python, scelta dettata soprattutto dalla volontà di utilizzare la libreria *scikit-learn* [5], che implementa le tecniche di *machine learning* necessarie al funzionamento di RiskInDroid. Una volta completato lo sviluppo e dopo aver valutato empiricamente l'efficacia del metodo, RiskInDroid è stato integrato come componente di Approver [6], un *tool* professionale per l'analisi completa delle applicazioni mobili sviluppato da Talos S.r.l.s (`http://www.talos-security.com/`) in collaborazione con il Computer Security Lab del dipartimento DIBRIS dell'Università di Genova.

1.1 Organizzazione della tesi

La tesi è organizzata secondo l'ordine cronologico di svolgimento: si inizia con una presentazione complessiva del lavoro che si intende compiere, si prosegue illustrando metodologie e strumenti impiegati per concludere infine con l'analisi e il commento dei risultati ottenuti, indicandone eventuali sviluppi futuri.

La Sezione introduttiva offre una panoramica generale sugli argomenti trattati in questa tesi ed espone gli obiettivi prefissati.

La Sezione 2 presenta in maniera sintetica l'architettura del sistema operativo Android, con particolare attenzione al meccanismo di sicurezza basato sull'utilizzo di permessi e menzionando i principali canali di distribuzione delle app.

La Sezione 3 mostra i risultati più importanti ottenuti finora dalla comunità scientifica nell'ambito della classificazione binaria di *malware* e dell'analisi quantitativa del rischio delle applicazioni Android. Viene inoltre brevemente descritta la libreria di Python che implementa le tecniche di *machine learning* adoperate nel corso della tesi.

Nella Sezione 4 viene esposta nel dettaglio la metodologia adottata in questo elaborato di laurea: inizialmente viene proposto un approccio per l'analisi del rischio delle applicazioni basato sull'elenco CVE, che si rivela tuttavia poco funzionale; si valutano pertanto due ulteriori metodi, il secondo dei quali, denominato RiskInDroid, ha un carattere innovativo e risulta efficace nell'utilizzo pratico.

La Sezione 5 riporta i risultati ottenuti sperimentalmente applicando il metodo RiskInDroid su un campione statistico costituito da oltre 100 000 applicazioni Android. Le app valutate con un indice di rischio elevato vengono successivamente analizzate con una suite di antivirus online per determinarne la reale pericolosità.

Nella Sezione conclusiva vengono fatte alcune considerazioni sui risultati ottenuti durante lo svolgimento della tesi, illustrando infine alcuni spunti per proseguire ulteriormente il lavoro iniziato in questo elaborato di laurea.

2 Ecosistema Android

Android è un sistema operativo per dispositivi mobili *open source* basato su *kernel* Linux, sviluppato da Google e dall'Open Handset Alliance. È composto da un'architettura a più livelli di astrazione[1], come visibile in Fig. 1.

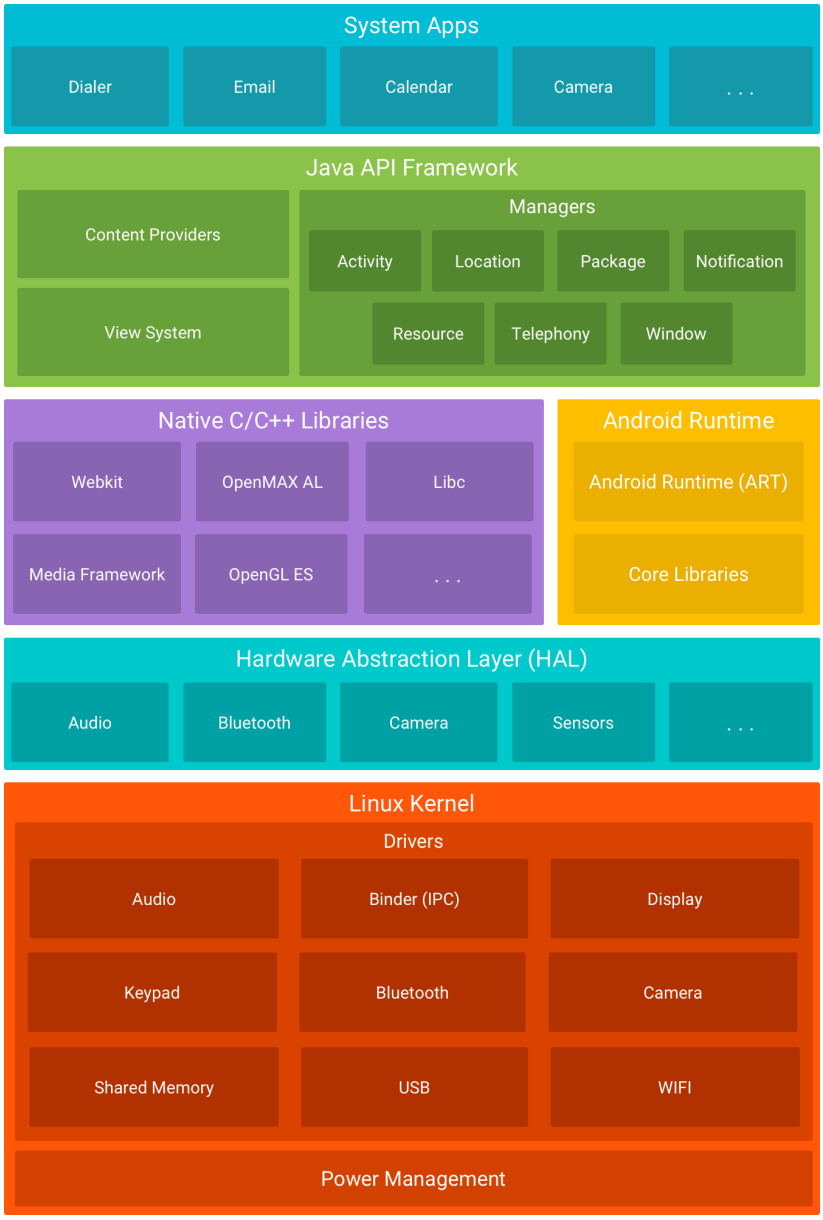


Figura 1: Architettura a livelli della piattaforma Android

In cima allo *stack* dell'architettura di Android si trovano le **applicazioni di sistema** che offrono le funzionalità di base (gestione email, calendario ecc.). Al livello

[1] `http://developer.android.com/guide/platform/index.html`

appena inferiore è presente il **Java API framework**, che fornisce un set di componenti modulari riutilizzabili per lo sviluppo di nuove applicazioni in linguaggio Java, come ad esempio funzioni per accedere alle risorse o per mostrare notifiche nella barra di stato dell'app. Il sistema Android rende inoltre disponibile l'accesso a **librerie native** scritte in C/C++ tramite l'impiego di Android NDK (*Native Development Kit*), nel caso si vogliano ottenere prestazioni più elevate nelle operazioni che coinvolgono l'impiego di grafica 2D e 3D. L'**Android Runtime** è il livello dove vengono eseguite più istanze di macchine virtuali (una per ogni applicazione) ottimizzate per avere un basso consumo di risorse. In particolare, ogni macchina virtuale esegue il `DEX` *bytecode* generato a partire dall'app scritta in Java. L'**Hardware Abstraction Layer** è formato da un insieme di librerie che rende possibile l'utilizzo delle risorse hardware da parte del livello superiore (Java API framework). Alla base di tutta l'architettura è presente il **Kernel Linux**, che astrae l'hardware del dispositivo e fornisce funzionalità elementari, ad esempio per la gestione dei processi, della memoria o dell'*Inter Process Communication (IPC)*.

Il sistema operativo Android assegna ad ogni applicazione un Linux `user ID` univoco in fase di installazione della stessa. In combinazione al fatto che ogni app viene eseguita in una propria istanza di macchina virtuale, ciò contribuisce alla creazione di una *sandbox* che garantisce l'isolamento di ogni applicazione installata rispetto alle altre e rispetto al sistema operativo. Android implementa dunque il principio del minimo privilegio[2], pertanto ogni app che voglia accedere a risorse protette del dispositivo (fotocamera, memoria esterna ecc.) oppure a dati di altre applicazioni deve richiedere esplicitamente l'autorizzazione all'utente. Questo meccanismo di autorizzazione viene garantito in Android tramite l'utilizzo di permessi. Nelle versioni di Android precedenti alla 6.0, i permessi sono richiesti in fase di installazione dell'app e l'utente deve accettarli tutti affinché l'installazione venga completata; nelle versioni successive i permessi sono invece gestiti dinamicamente (ovvero sono richiesti quando vengono utilizzati)[3].

[2] `http://developer.android.com/guide/components/fundamentals.html`
[3] `http://developer.android.com/training/permissions/requesting.html`

2.1 Sistema dei permessi in Android

Il sistema dei permessi è uno dei principali meccanismi di sicurezza su cui si basa Android. Ogni app viene infatti eseguita in un processo isolato ed è grazie ai permessi che le vengono concessi che può accedere a specifiche risorse del sistema. I permessi sono dichiarati nel file `AndroidManifest.xml`, che contiene anche altre informazioni fondamentali riguardanti l'applicazione come il ad esempio il suo nome, le versioni di Android compatibili e tutti i componenti presenti nell'app. Al momento della stesura della tesi, il sistema operativo Android mette a disposizione più di 130 permessi[4]; gli sviluppatori possono tuttavia definire nuovi permessi (ad esempio per esporre in maniera controllata funzionalità delle proprie app verso l'esterno) o richiedere permessi specifici dichiarati da altre applicazioni. I permessi inclusi nel sistema Android sono classificati in 4 categorie[5] in base al loro rischio potenziale:

1) `normal`: tipologia di permesso utilizzata di default che viene concessa all'applicazione direttamente dal sistema senza notificare l'utente. Si tratta di permessi per accedere a risorse a basso rischio;

2) `dangerous`: categoria di permessi più rischiosi del normale, generalmente non necessari per le applicazioni, ma richiesti per accedere ai dati privati e all'hardware del dispositivo (ad esempio, per leggere i contatti o per effettuare chiamate). Tali permessi possono essere concessi solamente tramite la conferma esplicita dell'utente;

3) `signature`: questi permessi possono essere concessi solamente ad applicazioni firmate con lo stesso certificato di quelle che hanno dichiarato i permessi e, in caso la firma coincida, sono concessi senza bisogno di notificare l'utente;

4) `signatureOrSystem`: tipologia di permesso speciale garantita in modo automatico alle applicazioni incluse nell'immagine di sistema Android.

2.2 *Application Store*

Il successo e la diffusione di un sistema operativo sono fortemente influenzati dalla quantità di programmi e/o applicazioni disponibili. Rispetto alla concorrenza,

[4] `http://developer.android.com/reference/android/Manifest.permission.html`
[5] `http://developer.android.com/guide/topics/manifest/permission-element.html`

Android è avvantaggiato dal fatto di essere *open source* e perché, modificando una semplice impostazione, permette di installare app da fonti non ufficiali. Per questo motivo, oltre al Google Play Store, sono presenti svariati *application store* alternativi da dove è possibile scaricare ed installare app per il proprio dispositivo mobile, come ad esempio Aptoide e Uptodown. In questa tesi vengono dunque presi in considerazione sia canali di distribuzione ufficiali (ad esempio Google Play Store, Samsung Store ecc.) sia fonti non ufficiali (come siti web, *market* non ufficiali ecc.) per disporre di un campione statistico sufficientemente ampio per una valutazione affidabile dell'approccio proposto in questa tesi per l'analisi del rischio di app Android.

3 Stato dell'arte

Al giorno d'oggi Android è un sistema operativo installato sull'86.2% dei dispositivi mobili [1], con centinaia di migliaia di applicazioni disponibili (e non sempre provenienti da fonti ufficiali), pertanto il problema della sicurezza su questa piattaforma è molto attuale. Negli ultimi anni la comunità scientifica, e non solo, ha proposto diverse metodologie allo scopo di individuare le caratteristiche ed i comportamenti che contraddistinguono i *malware* (app malevole) dalle app benigne. In particolare, esistono due approcci generali per il rilevamento di applicazioni malevole: analisi statica e dinamica. Le tecniche basate sull'analisi statica consistono nell'esaminare il codice sorgente decompilato, senza eseguire l'applicazione; in questo modo è possibile estrarre informazioni per individuare, ad esempio, i permessi, i *broadcast receiver* oppure la presenza di stringhe sospette. Nel caso di analisi dinamica, l'app viene invece eseguita in un ambiente di test controllato per monitorarne il comportamento, ed è possibile ottenere dati riguardanti il traffico di rete generato o le risorse di sistema utilizzate. Generalmente, entrambi i tipi di analisi prevedono la creazione di un modello di classificazione a partire dalle informazioni osservate, in modo da riuscire a distinguere le applicazioni benigne da quelle maligne in base alle loro caratteristiche, codificate come *feature*.

In questo elaborato di laurea ci si concentra solamente sui metodi di analisi statica, in particolare su quelli basati sui permessi relativi alle app Android, senza indagare su metodologie che ricorrono all'analisi dinamica (ad esempio [7]): come dimostra infatti la letteratura scientifica (Sezioni 3.1 e 3.2), impiegando metodologie incentrate sull'analisi statica si ottengono spesso buoni risultati e non è necessario investire risorse in dispositivi mobili e/o emulatori (indispensabili per l'analisi dinamica). Nello specifico, si è partiti esaminando i lavori presenti nella letteratura scientifica riguardanti la classificazione binaria di *malware* e quelli relativi alla creazione di un punteggio di rischio per le applicazioni Android. Dopodiché, dall'analisi dello stato dell'arte di queste due branche di ricerca è stato proposto un approccio nuovo (RiskInDroid) per la generazione di un indice di rischio per app Android.

3.1 Classificazione binaria di *malware*

Le pubblicazioni scientifiche che trattano il problema della classificazione binaria di *malware* in ambito Android sono numerose. Vengono analizzate di seguito solo

quelle più rilevanti e promettenti ai fini dell'obiettivo di questa tesi.

Uno dei primi lavori proposti allo scopo di individuare i *malware* in maniera semplice ed efficiente è Kirin [8]. Si tratta di un elenco di regole di sicurezza che descrivono pattern considerati pericolosi a partire dalle informazioni reperibili nel file `AndroidManifest.xml` delle app Android. Ad esempio, è considerata pericolosa un'app che richieda contemporaneamente i permessi `RECEIVE_SMS` e `WRITE_SMS`. Questa tecnica viene valutata su un totale di 311 applicazioni popolari scaricate dal Google Play Store nel 2009, individuando 10 app che violano la tutte le regole predefinite, di cui 5 risultano effettivamente sospette anche dopo un'analisi manuale più approfondita.

Gli autori di [9] suggeriscono una metodologia basata sul *machine learning* per riconoscere i *malware*, utilizzando come algoritmi *k-means* e gli alberi decisionali. Da ogni applicazione analizzata viene estratto un vettore di *feature* formato da 0 e 1, riguardante soprattutto i permessi richiesti dall'app, dove 1 indica la richiesta dello specifico permesso e 0 l'assenza; questo modo di costruire le *feature* viene adottato anche in RiskInDroid. Il metodo sviluppato in [9] viene validato con un *dataset* di 500 applicazioni ottenendo buoni risultati.

In [10] viene presentato un lavoro simile a [9], in quanto si utilizzano tecniche di *machine learning* su insiemi di *feature* estratte dal file `AndroidManifest.xml` delle applicazioni Android (in particolare i permessi dichiarati). Con a disposizione un *dataset* composto da 249 *malware* e 357 app benigne, in [10] si valutano le prestazioni di diversi classificatori nell'individuare le app malevole, fra cui alberi decisionali, *Random Forest* e *Naive Bayes*.

Gli autori di [11] sviluppano una metodologia basata non solo sui permessi dichiarati dalle app Android, ma anche su quelli effettivamente utilizzati. Inoltre, per la creazione dei vettori di *feature*, non vengono considerati solamente i singoli permessi, ma anche le coppie di permessi (sia dichiarati sia realmente utilizzati). La tecnica proposta è articolata in due fasi sequenziali in cui si controllano inizialmente i permessi richiesti e poi le coppie di permessi effettivamente utilizzati. Empiricamente si ottengono ottimi risultati, impiegando come classificatore gli alberi decisionali e

conducendo i test su un insieme di 20 548 applicazioni benigne e 1 136 *malware*.

In [12] viene proposto DREBIN, un metodo efficiente per il riconoscimento di *malware* tramite analisi statica. A partire dal file `AndroidManifest.xml` e dal codice decompilato delle app, vengono estratti insiemi di *feature* sottoposti ad un classificatore formato da *Support Vector Machines* (SVM). Questo metodo viene testato su un insieme di 123 453 applicazioni benigne e 5 560 *malware*, ottenendo risultati che indicano un'accuratezza del 94%, con solo l'1% di falsi positivi. Questa metodologia può essere impiegata anche direttamente sui dispositivi mobili, con tempi di esecuzione nell'ordine di alcuni secondi. Per ogni app esaminata, DREBIN fornisce inoltre una descrizione delle *feature* sospette individuate. La collezione di *malware* utilizzata in [12] viene resa disponibile come DREBIN *dataset* e viene impiegata anche per le prove empiriche utilizzate per la validazione sperimentale di RiskInDroid.

3.2 Analisi del rischio

La letteratura scientifica riguardante l'analisi del rischio delle applicazioni Android contiene un numero inferiore di pubblicazioni rispetto a quella relativa alla classificazione binaria di *malware*; pertanto, nel seguito di questa sezione sono presentati i principali lavori scientifici che prevedono metodi quantitativi per la valutazione del rischio delle app Android.

In [2] è descritto un metodo per rilevare segnali di rischio per le applicazioni in base a quanto raramente vengono richiesti permessi (o coppie di permessi) critici, dove con permesso critico si intende un permesso con un impatto significativo per la sicurezza e/o per la privacy dell'utente. In particolare, se un'app richiede un permesso che è poco presente nelle altre app della stessa categoria, tale app è considerata rischiosa. Sempre in [2] viene indicato come ottenere alcune funzioni per il calcolo del rischio delle applicazioni Android, basandosi su modelli probabilistici bayesiani; tuttavia, per poter applicare questi metodi, è necessario conoscere anche la categoria di appartenenza dell'app (informazione reperibile nello *store* da cui viene scaricata l'app, come ad esempio Giochi, Musica, Produttività ecc.). Vengono inoltre menzionate 3 proprietà interessanti che dovrebbe avere ogni funzione per la generazione di un punteggio di rischio:

1) **monotonicità**, rimuovendo un permesso richiesto da un'applicazione, il valore di rischio della stessa dovrebbe diminuire;

2) **coerenza**, le app maligne dovrebbero avere un indice di rischio alto;

3) **facilità di comprensione**, in modo che il valore di rischio di un'applicazione sia facilmente interpretabile e confrontabile con altre applicazioni.

Le prove empiriche in [2] sono eseguite su un *dataset* composto da due insiemi di applicazioni benigne costituiti rispettivamente da 71 331 e 136 534 campioni, raccolti nel 2011 e nel 2012, e da una collezione di 808 *malware*.

In [3] viene suggerito un approccio per calcolare il rischio delle app Android basandosi sulla loro categoria di appartenenza. Nello specifico, una volta creati i vettori di *feature* a partire dai permessi richiesti dalle applicazioni, per ogni categoria si determinano quali e quanti sono i permessi più richiesti dalle app benigne presenti in quella categoria, individuando in tal modo i pattern di permessi per ogni categoria. A questo punto è possibile calcolare un indice di rischio per una nuova applicazione andando a misurare quanto i permessi richiesti dall'app si discostano dai pattern rilevati in precedenza. Nonostante i buoni risultati ottenuti empiricamente su un *dataset* formato da 7 737 app benigne e 1 260 *malware*, il limite maggiore di questo metodo consiste nella necessità di conoscere la categoria di appartenenza delle applicazioni, informazione non sempre disponibile o affidabile.

In [13] viene descritto un *framework* di valutazione del rischio per le applicazioni composto di 3 livelli interdipendenti: analisi statica, analisi dinamica e analisi comportamentale. Ogni livello si occupa di aspetti specifici dell'applicazione ed è il *framework* proposto a combinare i risultati ricavati dai singoli moduli in modo da ottenere il rischio associato all'applicazione, il quale è composto da informazioni dettagliate come un punteggio di rischio numerico e una lista di fattori determinanti che rendono l'app rischiosa. Il *framework* presentato è descritto nel dettaglio ma solamente in maniera teorica, in quanto in [13] non vengono riportate prove empiriche a sostegno della metodologia proposta.

Gli autori di [4] realizzano DroidRisk, un metodo quantitativo per il calcolo di un indice di rischio per le applicazioni Android. Disponendo di una collezione

formata da 27 274 app benigne e 1 260 app maligne, in [4] vengono presentate alcune statistiche relative ai permessi delle applicazioni nel *dataset* e successivamente viene descritta una formula probabilistica, costituita dal prodotto fra la probabilità di richiesta e fra l'impatto di ogni permesso (inteso come il danno potenziale che si può causare utilizzando tale permesso). Questo metodo viene approfondito più nel dettaglio nella Sezione 4.4 di questa tesi.

3.3 Libreria *scikit-learn*

Scikit-learn [5] (`http://scikit-learn.org/`) è una libreria *open source* sviluppata in linguaggio Python che contiene le implementazioni dei principali algoritmi di *machine learning* per la risoluzione di problemi di classificazione, regressione e *clustering*. Lo scopo di questa libreria non è concentrarsi sul numero di funzionalità offerte, ma di fornire algoritmi implementati allo stato dell'arte, con particolare attenzione alla qualità del codice sorgente sviluppato. In [14] si può trovare una descrizione più approfondita della struttura interna della libreria; qui di seguito ci si limita a discutere le motivazioni che rendono *scikit-learn* adatta per l'obiettivo di questa tesi.

Il motivo principale per la scelta di impiegare *scikit-learn* è la presenza della funzione `predict_proba` nella maggior parte dei classificatori resi disponibili dalla libreria. Si tratta di una funzione che, invece di restituire solamente il nome della classe predetta, indica in output la probabilità con cui ogni classe viene predetta dai metodi di classificazione. Per alcuni classificatori, come *Naive Bayes*, ottenere le probabilità di ogni classe è immediato perché sono i classificatori stessi ad essere probabilistici; in altri casi è invece possibile ricavare le probabilità delle classi predette utilizzando le tecniche descritte in [15], come viene fatto ad esempio per le *Support Vector Machines*.

Scikit-learn è dunque una libreria potente e versatile, ma anche relativamente semplice da utilizzare per gli utenti meno esperti. Presenta tuttavia dei limiti per quanto riguarda l'esecuzione parallela di codice che sfrutti la presenza di più processori di calcolo; inoltre non è stata ancora realizzata una soluzione stabile e duratura per il salvataggio in maniera persistente dei modelli di classificazione, una volta eseguito il *training* (operazione che può rivelarsi costosa in termini di tempo).

4 Metodologia

Per l'implementazione dei metodi proposti in questo elaborato di laurea vengono utilizzati principalmente due strumenti di sviluppo. Inizialmente, per il lavoro di tesi viene impiegato l'Integrated Development Environment (IDE) IntelliJ IDEA 2016[6] e il linguaggio di programmazione Java, con cui viene scritto il codice relativo alla Sezione 4.2. Successivamente, per poter far uso della libreria di *machine learning* chiamata *scikit-learn*, si passa al linguaggio Python, adoperando come *editor* Visual Studio Code[7]. Trattandosi di strumenti multi-piattaforma, durante lo svolgimento della tesi vengono utilizzati sia il sistema operativo Windows 10 sia Ubuntu 16.04 (eseguito in una macchina virtuale).

4.1 Raccolta delle applicazioni

Per la valutazione della qualità dei metodi proposti e delle relative implementazioni, è prima di tutto necessario costruire una base di dati che contenga un buon numero di campioni da analizzare.

Fonte	Numero di applicazioni
Raccolta *malware*	6 707
Google Play Store[8]	101 730
Aptoide[9]	7 609
Uptodown[10]	7 202
Totale	123 248

Tabella 1: Composizione del *dataset* di applicazioni

Nella Tabella 1 è visibile l'origine delle applicazioni prese in esame durante questo elaborato di laurea. L'insieme di *malware* è costituito principalmente dal DREBIN *dataset* [12] (5 560 campioni), con l'aggiunta di ulteriori elementi ottenuti da risorse liberamente accessibili online [16, 17, 18]. I campioni restanti vengono invece scaricati dai relativi *app store* mediante un *crawler* creato ad hoc; dato che il Google

[6] `http://www.jetbrains.com/idea/`
[7] `http://code.visualstudio.com`
[8] `http://play.google.com/store/apps`
[9] `http://www.aptoide.com/page/apps`
[10] `http://en.uptodown.com/android`

Play Store è il canale ufficiale per ottenere applicazioni per dispositivi Android, queste vengono considerate (solo inizialmente) non malevole[11]. A causa della natura eterogenea delle fonti, ad ogni app viene associato un codice univoco di 32 cifre esadecimali, chiamato *hash MD5*, per evitare di avere duplicati.

Prendere in considerazione una quantità elevata di dati da analizzare (come fatto per il *dataset* di questa tesi) è importante perché permette di ottenere risultati che valgono in generale e non solo in casi specifici, con il vantaggio di avere una maggiore disponibilità di elementi su cui testare gli algoritmi proposti. Tuttavia, a causa del numero relativamente basso di *malware* nel *dataset*, per condurre prove empiriche, vengono estratti in modo pseudo-casuale 3 insiemi distinti dalle applicazioni scaricate dal Google Play Store che abbiano la stessa dimensione della raccolta di *malware*, impiegando la funzione `random.sample`[12] presente in Python e fissando il parametro `random.seed` in modo da avere sempre risultati riproducibili. Pertanto, ognuno dei 3 set ricavati è formato per metà dalla collezione di applicazioni maligne e per l'altra metà da un insieme di app benigne provenienti dal Google Play Store; in questo modo ogni set creato contiene al suo interno un numero bilanciato di *malware* e di applicazioni non malevole.

4.2 *Common Vulnerabilities and Exposures*

Un primo tentativo di creare un punteggio di rischio è stato fatto utilizzando il *Common Vulnerabilities and Exposures (CVE)*, un database liberamente consultabile online all'indirizzo web `http://cve.mitre.org/` contenente informazioni di dominio pubblico su falle di sicurezza e vulnerabilità note relative ai sistemi informatici. In particolare, per gli scopi di questo elaborato, si era interessati esclusivamente alle *entry* collegate all'ecosistema Android. A ogni voce inserita nel CVE è assegnato un codice in formato `CVE-[anno]-[numero progressivo]` (ad esempio `CVE-2016-3897`) che identifica in modo univoco una vulnerabilità. Tramite questo codice è possibile reperire ulteriori informazioni dal *National Vulnerability Database (NVD)*, sito gestito dal governo degli Stati Uniti[13]. Nello specifico, si possono ottenere dettagli tecnici come la classe a cui appartiene la vulnerabilità, le configu-

[11] La collezione di applicazioni provenienti dal Google Play Store viene analizzata nella Sezione 5 per determinare se contiene o meno campioni appartenenti alla categoria *malware*

[12] `http://docs.python.org/3/library/random.html#random.sample`

[13] `http://nvd.nist.gov/`

razioni di software esposte al rischio, le competenze necessarie per sfruttare la falla e soprattutto un punteggio numerico fra 0.0 e 10.0 che ne indica la gravità, chiamato *Common Vulnerability Scoring System (CVSS)*[14].

Al momento della stesura della tesi, consultando il CVE si possono trovare circa 2 500 voci relative ad Android. Una prima strada esplorata per l'ottenimento di un indice di rischio per le applicazioni è quindi la seguente: per ogni app analizzata, si scorre l'elenco delle vulnerabilità del CVE collegate ad Android, controllando di volta in volta se la descrizione della falla di sicurezza è in qualche modo riconducibile all'applicazione in esame o a un suo componente. Se si riescono a trovare una o più vulnerabilità, queste vengono assegnate all'app e, grazie al loro identificativo univoco, è possibile estrarne il punteggio CVSS da cui ottenere infine l'indice di rischio desiderato. La fase cruciale e più difficile di questo metodo è riuscire a fare il *mapping* fra la descrizione in linguaggio naturale della vulnerabilità e l'applicazione stessa o un suo componente.

Analizzando più nel dettaglio l'elenco CVE inerente ad Android, si possono individuare più categorie di vulnerabilità:

a) Problemi relativi alla verifica dei certificati SSL X.509. Si tratta di vulnerabilità che affliggono versioni specifiche di determinate applicazioni e costituiscono circa il 56% della lista CVE considerata. Questo gruppo di problemi non può essere utilizzato per la creazione di un indice di rischio generale perché è riferito a versioni specifiche di app, che con buona probabilità avranno già corretto il baco nella *release* immediatamente successiva;

b) Vulnerabilità inerenti soltanto determinate applicazioni, quindi non utilizzabili per la costruzione di un punteggio di rischio generale. Ad esempio, sommando i problemi relativi ad Adobe Flash Player, Mozilla Firefox e Google Chrome, si ottiene circa l'11% del totale considerato;

c) Problemi con le librerie native sviluppate in C/C++. Quasi l'8% delle vulnerabilità nell'elenco CVE considerato contiene riferimenti a file `.c` e/o `.cpp`. Tuttavia, una volta che il codice nativo dell'applicazione viene compilato, non è più possibile risalire ai nomi dei sorgenti, quindi non si può realizzare un indice di rischio che tenga conto di queste informazioni;

[14] `http://www.first.org/cvss`

d) Vulnerabilità relative a componenti del sistema operativo Android sviluppati con Java. In questo caso, nella descrizione delle voci nell'elenco CVE considerato si fa riferimento a file `.java`, indicandone eventualmente il percorso completo (ad esempio `internal/telephony/SMSDispatcher.java`). Utilizzando uno strumento come Apktool [19], è possibile decompilare il *Dalvik bytecode* dell'applicazione in un linguaggio intermedio, chiamato Smali [20], che ha il vantaggio di mantenere intatti i nomi delle classi Java. Per esempio, decompilando un'app che richiede l'uso di `internal/telephony/SMSDispatcher.java`, internamente al codice Smali si ritroverebbe la stringa `Linternal/telephony/SMSDispatcher.java;`[15]. In questo caso è dunque possibile stabilire una corrispondenza fra le applicazioni nel *dataset* e le vulnerabilità presenti nel CVE.

Per la creazione di un indice di rischio a partire dalla lista CVE relativa ad Android si impiega dunque la categoria di vulnerabilità vista al punto d) dell'elenco precedente, utilizzando il procedimento illustrato nella Fig. 2. Nella prima fase si sceglie l'applicazione che si vuole analizzare e la si decompila con Apktool, generando in

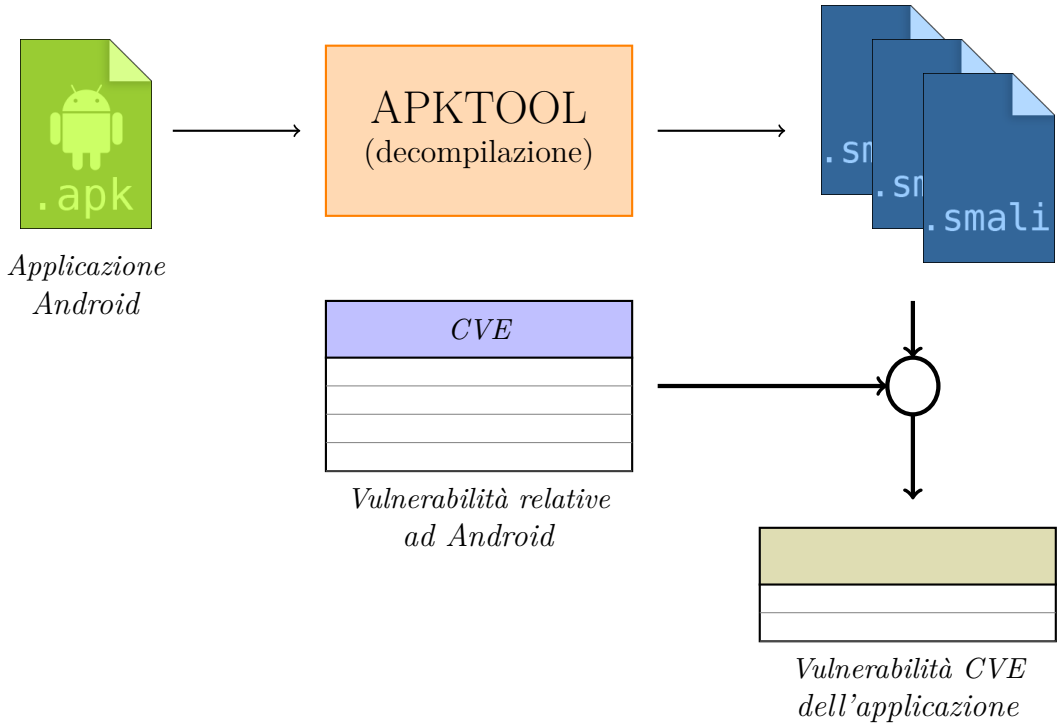


Figura 2: Identificazione delle vulnerabilità utilizzando Apktool

[15] `http://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields`

questo modo una cartella al cui interno si trovano anche i file `.smali`. Il secondo passo consiste nell'individuare all'interno del CVE soltanto quelle vulnerabilità nella cui descrizione è presente almeno un nome di file `.java`. Al momento della stesura della tesi, risultano 43 elementi che soddisfano questo criterio. A questo punto, per ogni nome di file `.java` individuato, si controlla se tale nome è presente (sotto forma di stringa) nel codice Smali generato da Apktool (impiegando, ad esempio, il *tool* da linea di comando `grep`). In caso di esito positivo, l'app in esame risulta soggetta alla vulnerabilità e si può procedere al calcolo di un indice di rischio adoperando il punteggio CVSS reperibile nel NVD, tramite l'identificativo univoco che caratterizza la vulnerabilità. Esiste tuttavia la possibilità, seppur non considerata in questa tesi, che il codice sorgente dell'applicazione risulti "offuscato" e in tal caso non è garantito il funzionamento del metodo proposto. Il termine "offuscato" si usa per indicare codice sorgente sintatticamente corretto che però è scritto in maniera volutamente contorta e apparentemente incomprensibile, in modo da rendere il più difficile possibile la comprensione anche da parte dell'utente l'utente più esperto. L'algoritmo 1 descrive più nel dettaglio la procedura utilizzata per il calcolo di un punteggio di rischio a partire dalla lista CVE con le voci relative ad Android.

Algorithm 1: Calcolo di un indice di rischio a partire dalla lista CVE

Input : Applicazione da analizzare, lista vulnerabilità CVE
Output: Indice di rischio per l'applicazione

```
smali = decompilaApk(applicazione);
foreach vulnerabilità in listaCVE do
    fileJava = ottieniNomiFileJava(vulnerabilità);
    foreach nomeFile in fileJava do
        if smali contains nomeFile then
            applicazione.listaVulnerabilità.aggiungi(vulnerabilità);
            continue

rischio = 0;
foreach vulnerabilità in applicazione.listaVulnerabilità do
    /* Viene scelto il massimo indice di rischio trovato */
    tmpScore = scoreCVSS(vulnerabilità);
    if tmpScore > rischio then
        rischio = tmpScore;

return rischio;
```

L'approccio appena proposto si rivela tuttavia poco efficace nell'uso pratico, a causa della scarsità di risultati ottenuti. Valutando infatti empiricamente questo metodo su un campione di circa 1 500 applicazioni del *dataset*, si riesce ad individuare un totale di solamente 3 vulnerabilità, quindi non si ritiene necessario procedere con l'analisi di ulteriori app. Il fatto di trovare così poche vulnerabilità può anche essere un risultato ragionevole se le applicazioni sono progettate bene e/o non utilizzano componenti di Android afflitti da problematiche di sicurezza; tuttavia l'esiguo numero di riscontri non è sufficiente per permettere di calcolare un punteggio di rischio riferibile in modo generale a tutte le applicazioni, in quanto seguendo questo approccio si avrebbe un indice di rischio nullo per la quasi totalità delle app prese in esame.

Data la scarsa efficacia di questo primo metodo proposto, si prendono in considerazione due ulteriori criteri per raggiungere lo stesso scopo di costruire un punteggio di rischio, questa volta però concentrandosi sull'**analisi dei permessi delle applicazioni**:

- **metodo probabilistico** (Sezione 4.4): si basa direttamente sulla probabilità di richiesta e/o utilizzo di determinati permessi da parte di *malware* e applicazioni benigne, similmente a quanto descritto in [4];
- **metodo basato sul *machine learning*** (Sezione 4.5): coinvolge l'utilizzo di tecniche di *machine learning* per estrarre un indice di rischio a partire dall'analisi dei permessi di *malware* e applicazioni benigne. Metodologie simili (come ad esempio [12]) sono già presenti in letteratura, però riguardano una classificazione binaria tra *malware* e non *malware*, senza fornire esplicitamente un punteggio di rischio per ogni applicazione esaminata. Questo nuovo metodo proposto nella tesi viene chiamato RiskInDroid.

4.3 Statistiche relative ai permessi

Prima di procedere illustrando le metodologie basate sull'analisi dei permessi, è opportuno soffermarsi ad esaminare alcune statistiche ad essi relative. In particolare, la motivazione per indagare ulteriormente su questo aspetto delle app nasce dall'osservazione delle differenze che ci sono fra permessi relativi ad applicazioni benigne e quelli collegati ai *malware*.

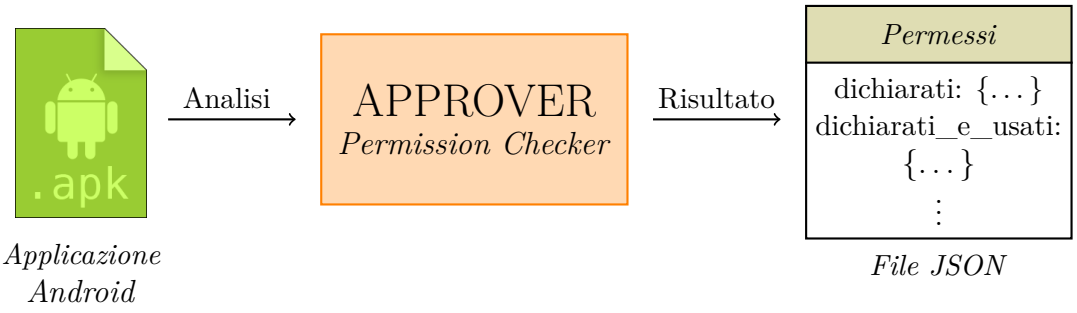


Figura 3: Modulo di Approver per l'estrazione dei permessi da un file `.apk`

Per ottenere i permessi delle applicazioni presenti nel *dataset* viene impiegato il modulo *Permission Checker* di Approver [6]. Come visibile nella Fig. 3, si tratta di un componente che prende in ingresso file di tipo `.apk` e restituisce in output una lista di permessi (in formato `JSON`) relativi all'app analizzata.

Nello specifico, il risultato ottenuto dal modulo raggruppa i permessi in 4 diverse categorie:

1) **Dichiarati**: permessi richiesti esplicitamente nel file `AndroidManifest.xml` presente in ogni applicazione per Android. Questa lista è relativamente facile da ricavare anche con altri *tool*, come ad esempio Androguard [21];

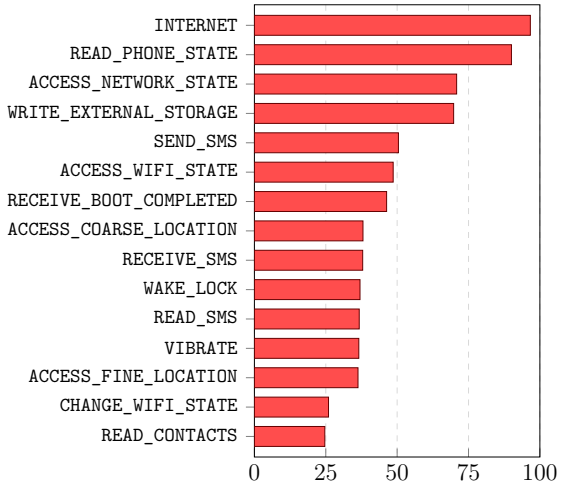
2) **Dichiarati e utilizzati**: permessi richiesti nel file `AndroidManifest.xml` che vengono realmente usati dall'applicazione. Per decidere se un permesso è utilizzato o meno, Approver analizza staticamente il *Dalvik bytecode* del file `classes.dex` contenuto nell'app e determina se vengono invocati metodi che hanno bisogno di permessi per essere eseguiti. Questa lista comprende dunque solo i permessi per cui c'è un riscontro all'interno del *bytecode*;

3) **Dichiarati ma non utilizzati**: permessi richiesti soltanto all'interno del file `AndroidManifest.xml` e per cui non c'è alcun riscontro dentro il *bytecode* dell'applicazione;

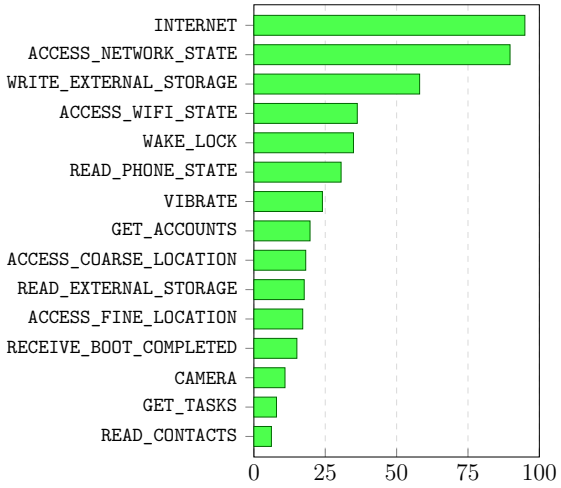
4) **Non dichiarati ma utilizzati**: permessi necessari per l'invocazione di metodi all'interno del *bytecode* ma che non sono presenti nel file `AndroidManifest.xml`.

Esaminando la Fig. 4, si può constatare la differente distribuzione dei permessi dichiarati fra le diverse categorie del *dataset* considerato. È interessante notare come, nel caso dei *malware*, tutti i 15 permessi della Fig. 4(a) siano richiesti da almeno

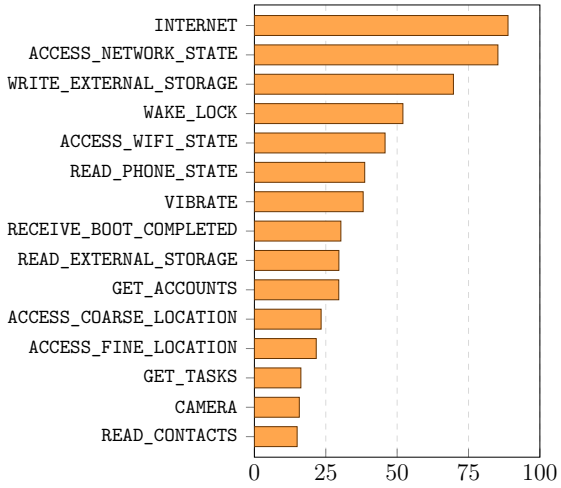
un quarto delle applicazioni malevole, mentre tale numero scende a 6 considerando solo i campioni del Google Play Store (Fig. 4(b)). Si può quindi dedurre che, statisticamente parlando, i *malware* necessitano mediamente di più permessi rispetto al resto delle app. Inoltre, sempre in Fig. 4, si può osservare come le applicazioni provenienti da *store* non ufficiali (Fig. 4(c) e 4(d)) abbiano statistiche che si trovano a metà strada tra *malware* e app benigne, e questo vale non solo per i permessi dichiarati, ma anche per le altre categorie di permessi, i cui grafici sono mostrati nelle Fig. da 5 a 7. Ciò potrebbe suggerire che le app di *store* alternativi siano in media



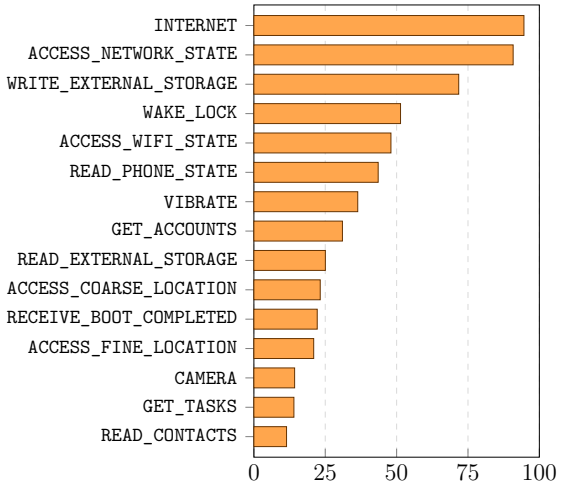
(a) Percentuale *malware*



(b) Percentuale app Google Play



(c) Percentuale app Aptoide



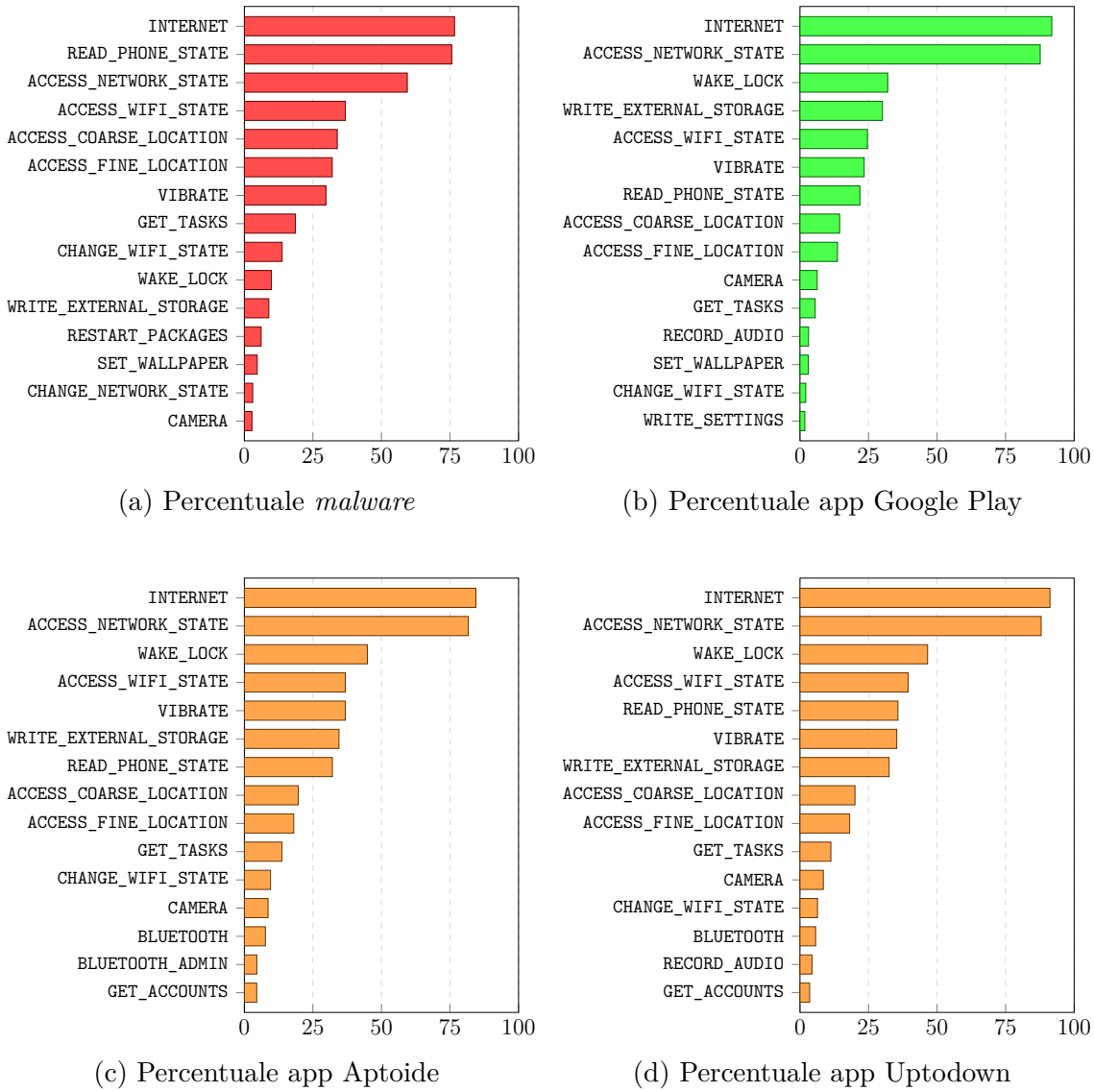
(d) Percentuale app Uptodown

Figura 4: I 15 permessi più **dichiarati** dalle applicazioni nel *dataset*

meno sicure di quelle provenienti dal Google Play Store, il ché è sensato se si pensa

ai maggiori controlli a cui dovrebbero essere sottoposte le applicazioni pubblicate nel Google Play Store.

Non tutti i permessi consentono però di fare una distinzione tra app benigne e maligne. `INTERNET`, ad esempio, è il permesso più dichiarato (Fig. 4) in tutte le categorie del *dataset* considerato ed è anche quello più utilizzato realmente dalle app (Fig. 5), quindi è impossibile valutare il rischio di un'applicazione considerando solamente questo parametro. Attualmente la maggioranza delle app necessita di una connessione ad Internet: applicazioni come quelle di messaggistica istantanea o rela-



(a) Percentuale *malware*

(b) Percentuale app Google Play

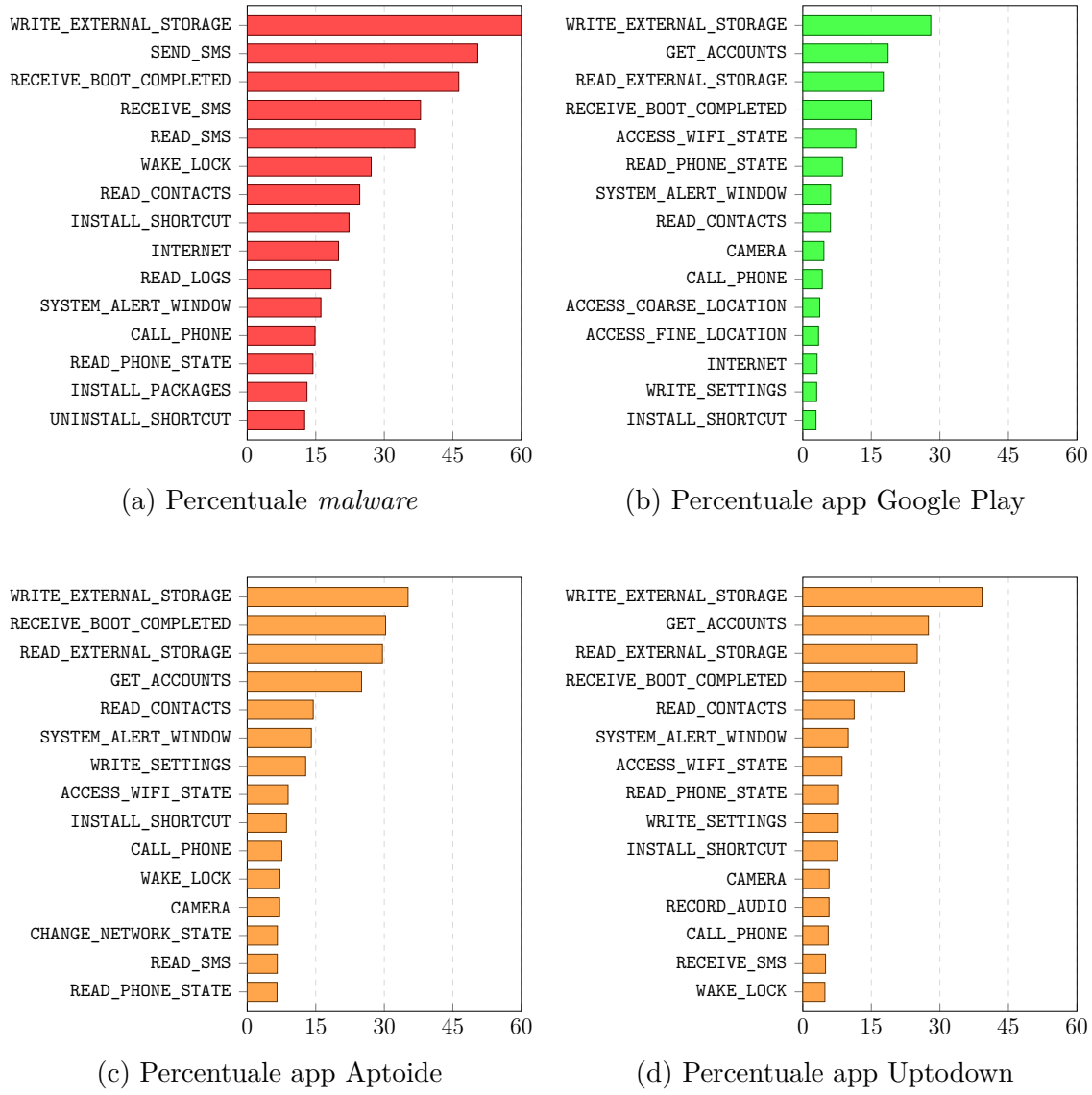
(c) Percentuale app Aptoide

(d) Percentuale app Uptodown

Figura 5: I 15 permessi più **dichiarati e anche utilizzati** dalle applicazioni nel *dataset*

tive ai *social network* devono accedere alla rete per poter funzionare, ma è sufficiente

pensare a giochi per dispositivi mobili che richiedono Internet solo per stilare le classifiche dei punteggi migliori o per mostrare annunci pubblicitari all'utente. Come conseguenza, i permessi riguardanti l'accesso alla rete come `ACCESS_NETWORK_STATE` o `ACCESS_WIFI_STATE` sono anch'essi molto diffusi.



(a) Percentuale *malware*

(b) Percentuale app Google Play

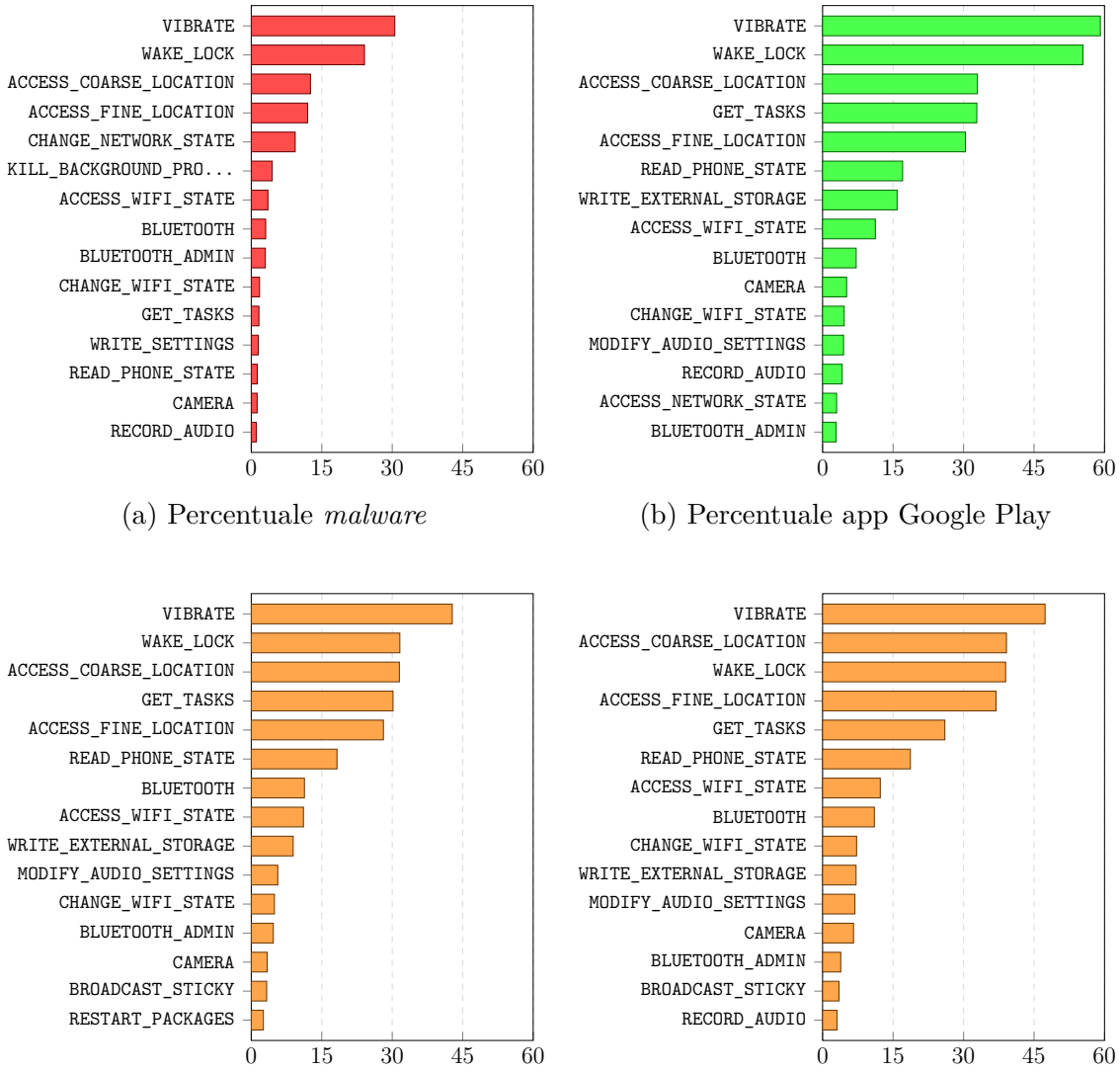
(c) Percentuale app Aptoide

(d) Percentuale app Uptodown

Figura 6: I 15 permessi più **dichiarati ma non utilizzati** dalle applicazioni nel *dataset*

Ci sono poi alcuni permessi come `READ_PHONE_STATE` e `RECEIVE_BOOT_COMPLETED` che sono dichiarati sia dai *malware* che dalle app benigne, ma con percentuali abbastanza differenti, tanto da poter intuire il rischio potenziale di un'applicazione in funzione di tali permessi. Ad esempio, secondo i grafici in Fig. 4(a) e 4(b), se un'app dichiara `READ_PHONE_STATE`, `RECEIVE_BOOT_COMPLETED` e `READ_CONTACTS` allora è

statisticamente più probabile che tale app appartenga alla categoria *malware*. La distinzione più significativa fra applicazioni benevole e malevole consiste tuttavia nei permessi relativi agli SMS: infatti, come dimostra la Fig. 4(a), 3 dei 15 permessi più dichiarati dai *malware* riguardano proprio gli SMS, mentre nelle altre categorie del *dataset* in Fig. 4 non c'è alcun riferimento al servizio di messaggistica. Sebbene



(a) Percentuale *malware*

(b) Percentuale app Google Play

(c) Percentuale app Aptoide

(d) Percentuale app Uptodown

Figura 7: I 15 permessi più **utilizzati ma non dichiarati** dalle applicazioni nel *dataset*

la maggior parte dei *malware* che richiedono permessi relativi agli SMS in realtà poi non li utilizzano (come testimonia la Fig. 6(a)), il fatto che un'app dichiari un permesso riferito agli SMS è comunque un indice di rischio statistico da tenere in considerazione. È infine interessante notare come la categoria dei *malware* sia quella

che dichiara più permessi senza poi utilizzarli (Fig. 6(a)), quindi è allo stesso tempo anche quella che utilizza meno permessi senza averli dichiarati (Fig. 7(a)).

4.4 Metodo probabilistico

Il metodo probabilistico implementato inizialmente è basato sul lavoro descritto in [4], in cui viene proposta la seguente formula:

$$R_A = \sum_i R\left(p_i\right) = \sum_i L\left(p_i\right) \times I\left(p_i\right) \tag{1}$$

dove R_A, il rischio associato all'applicazione A, è dato dalla somma dei rischi individuali $R\left(p_i\right)$ di ogni permesso, essendo $L\left(p_i\right)$ e $I\left(p_i\right)$ rispettivamente la probabilità e l'impatto del permesso p_i, con $i = 1, 2, \ldots, n$ ed n il numero totale di permessi relativi all'applicazione A. Si può definire $L\left(p_i\right)$ come la probabilità che l'app A sia maligna nel caso contenga il permesso p_i, utilizzando il teorema di Bayes:

$$L\left(p_i\right) = P\left(A \text{ è malware} \mid p_i\right) = \frac{P\left(p_i \mid A \text{ è malware}\right) \times P\left(A \text{ è malware}\right)}{P\left(p_i\right)} \tag{2}$$

dove, una volta fissato il *dataset*, $P\left(p_i \mid A \text{ è malware}\right)$ indica la probabilità che un *malware* abbia il permesso p_i, $P\left(A \text{ è malware}\right)$ è la probabilità che un'app del *dataset* sia maligna e $P\left(p_i\right)$ indica la probabilità che una qualsiasi applicazione del *dataset* contenga il permesso p_i. Ipotizzando, almeno per il momento, che ogni permesso abbia lo stesso impatto $I\left(p_i\right) = 1$, è possibile riscrivere l'Eq. (1) per il calcolo di un punteggio di rischio per un'app:

$$R_A = \sum_i \frac{P\left(p_i \mid A \text{ è malware}\right) \times P\left(A \text{ è malware}\right)}{P\left(p_i\right)} \tag{3}$$

A differenza di quanto fatto in [4], in questa tesi si vuole calcolare un punteggio di rischio percentuale compreso fra 0 e 100, quindi il risultato dell'Eq. (3) viene normalizzato dividendolo per il numero di permessi p_i relativi all'app A esaminata (moltiplicando infine il tutto per 100 in modo da ottenere una percentuale). Implementando in codice quanto appena scritto, si ottengono i dati presentati nella Tabella 2, che si riferiscono all'analisi dei soli permessi dichiarati dalle applicazioni nei 3 set descritti nella Sezione 4.1, usando la tecnica della *k–fold cross validation*

dopo aver impostato $k = 10$.

K–fold cross validation [22] è un metodo statistico che consiste nel suddividere in modo casuale il *dataset* iniziale in k parti, chiamate *fold*, che abbiano (per quanto possibile) lo stesso numero di elementi. A ogni iterazione su k, l'insieme k–esimo costituisce il *testing set* (chiamato anche *validation set*) su cui convalidare il modello costruito utilizzando i restanti $k - 1$ gruppi, il *training set*. Si calcola quindi un punteggio di accuratezza (percentuale di campioni del *testing set* per cui la classe predetta è uguale a quella reale) per ogni passo k–esimo e infine si trova la media di tali k valori. Questo metodo è vantaggioso perché tutti i campioni vengono impiegati sia per allenare sia per testare il modello, riducendo così il problema dell'*overfitting*, che si verifica quando il modello funziona molto bene con i dati usati per fare il *training*, ma i risultati ottenuti sono invece scadenti quando si devono classificare nuovi dati. Il parametro k viene comunemente impostato a 10, che empiricamente garantisce buoni risultati[16]. Nella Fig. 8 si può osservare graficamente un esempio di k–*fold cross validation* dove si fissa $k = 4$.

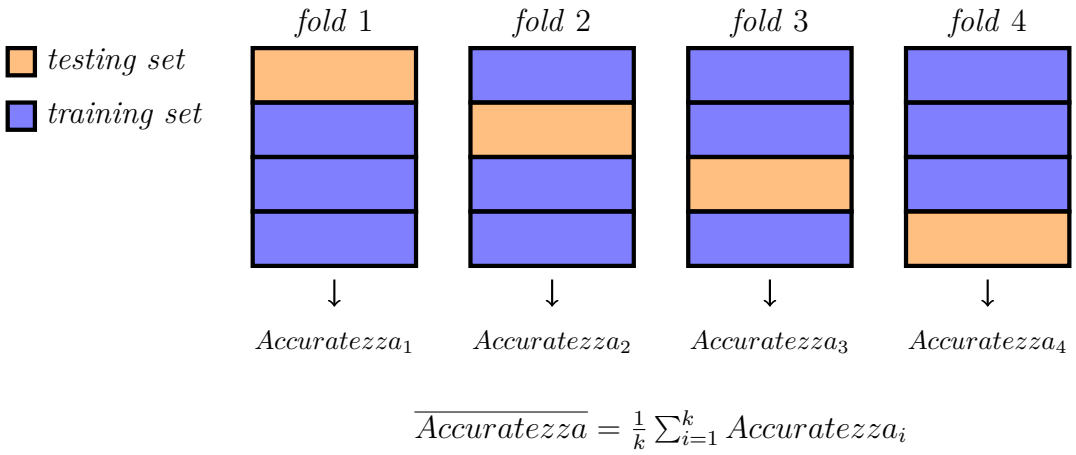


Figura 8: Esempio grafico di 4–*fold cross validation*

Osservando la Tabella 2 si può notare una distinzione abbastanza netta fra i punteggi percentuali generati per i *malware* e quelli per le app benigne; tuttavia è possibile aumentare ulteriormente questa differenza andando ad intervenire sull'impatto $I(p_i)$ di ogni permesso, finora considerato unitario.

[16] Valore di k consigliato in [23] e nella documentazione ufficiale della libreria *scikit-learn* consultabile all'indirizzo web `http://scikit-learn.org/dev/modules/cross_validation.html`

Applicazioni	Set 1		Set 2		Set 3	
	Punteggio medio %	Deviazione standard σ	Punteggio medio %	Deviazione standard σ	Punteggio medio %	Deviazione standard σ
Malware	70.16	8.66	70.04	8.64	69.99	8.65
Non *malware*	52.16	11.28	52.07	10.87	51.89	11.48

Tabella 2: Indice di rischio normalizzato, con impatto unitario $I(p_i) = 1$

Impatto dinamico dei permessi

In [4] si fissa $I(p_i) = 1$ per i permessi normali e $I(p_i) = 1.5$ per quelli *dangerous*. In questo elaborato si decide invece di stabilire l'impatto di un permesso p_i in maniera dinamica e dipendente dal permesso considerato. Si assegna pertanto a $I(p_i)$ il risultato della divisione fra la probabilità che un *malware* del *dataset* considerato abbia il permesso p_i e la probabilità che lo stesso permesso p_i sia invece contenuto da un'app benigna:

$$I(p_i) = \frac{P(p_i \mid A \text{ è malware})}{P(p_i \mid A \text{ non è malware})} \tag{4}$$

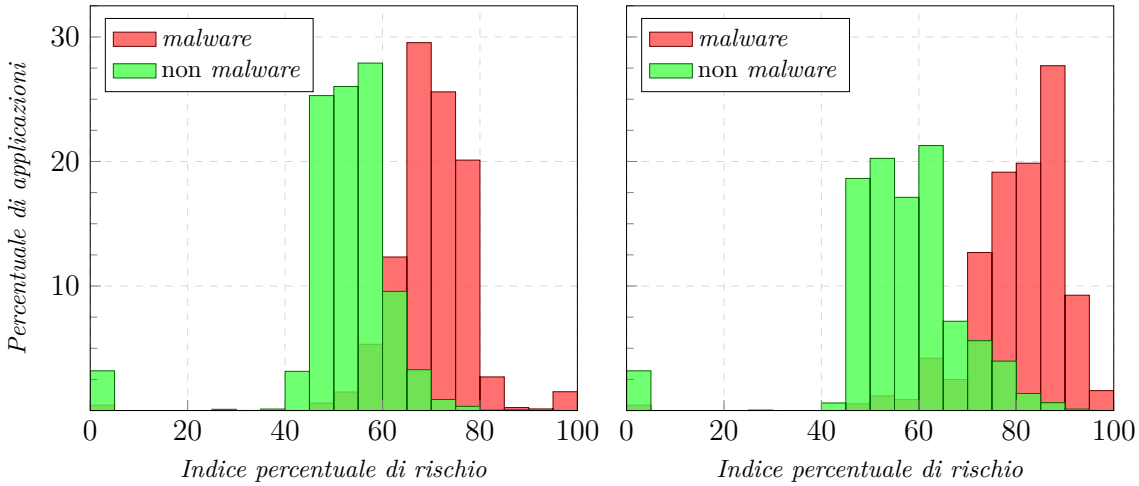
In questo modo, un permesso richiesto spesso dai *malware* ma che non compare quasi mai nelle applicazioni benigne ha un peso maggiore e influisce molto nel calcolo dell'indice di rischio. Viceversa, i permessi che caratterizzano prevalentemente le app non maligne condizionano poco il risultato finale. L'impatto dinamico attribuito ad ogni permesso dipende dunque dalla qualità del campione statistico utilizzato. Come visto nella Sezione 4.1, in questa tesi si dispone di un ampio *dataset* di applicazioni di cui si conoscono le statistiche relative ai permessi grazie all'analisi effettuata nella Sezione 4.3, pertanto è possibile assegnare all'impatto di ogni permesso un valore empiricamente valido. A questo punto, avendo tutti i parametri a disposizione, è possibile utilizzare la formula generale vista nell'Eq. (1) (per normalizzare il risultato, ora si divide invece per la somma degli impatti dei singoli permessi).

Implementando in codice l'Eq. (1) e tenendo conto dell'Eq. (4) si ottengono i risultati visibili nella Tabella 3 (lavorando sugli stessi dati della Tabella 2). È interessante notare che, indifferentemente dall'impatto $I(p_i)$ dei permessi, le statistiche mostrate nelle Tabelle 2 e 3 sono molto simili per i 3 set distinti di applicazioni. Que-

Applicazioni	Set 1		Set 2		Set 3	
	Punteggio medio %	Deviazione standard σ	Punteggio medio %	Deviazione standard σ	Punteggio medio %	Deviazione standard σ
Malware	80.54	10.35	80.42	10.37	80.34	10.39
Non *malware*	56.57	13.65	56.52	13.22	56.20	13.78

Tabella 3: Indice di rischio normalizzato, con impatto $I(p_i) = \frac{P(p_i|A \text{ è malware})}{P(p_i|A \text{ non è malware})}$

sto fatto conferma la validità del metodo utilizzato e permette inoltre di condurre ulteriori prove solamente su un singolo set di app, per poi generalizzare i risultati trovati. Per integrare i dati illustrati nelle Tabelle 2 e 3, la Fig. 9 presenta gli istogrammi con i punteggi di rischio nel caso di impatto unitario per ogni permesso e nel caso di utilizzo dell'Eq. (4). È possibile osservare come i *malware* abbiano mediamente un rischio maggiore rispetto alle applicazioni benigne, anche se è presente una zona dove gli istogrammi di *malware* e non *malware* si sovrappongono e quindi non si può distinguere la categoria a cui appartiene un'app che si trovi in questa regione. Si vede inoltre che esistono alcuni *malware* a rischio massimo ed è presente qualche applicazione a rischio nullo: quest'ultimo fatto si verifica per quelle app che non richiedono alcun permesso. È infine interessante constatare come anche le applicazioni benigne abbiano comunque un rischio quasi sempre maggiore del 40%.



(a) Impatto unitario $I(p_i) = 1$

(b) Impatto $I(p_i) = \frac{P(p_i|A \text{ è malware})}{P(p_i|A \text{ non è malware})}$

Figura 9: Istogramma dell'indice di rischio normalizzato

Per quanto i risultati ottenuti siano interessanti, il metodo di analisi del rischio presentato in questa sezione presenta tuttavia alcuni limiti:

- questa metodologia non riesce ad individuare i *malware* che dichiarano pochi o nessun permesso poiché, trattandosi appunto un metodo basato sul numero di permessi, se un'applicazione non richiede alcun permesso allora le verrà assegnato un indice di rischio nullo;
- il punteggio di rischio ottenuto impiegando questa tecnica è mediamente alto anche per le applicazioni benigne, per cui ci si aspetterebbe invece di avere indici di rischio situati nella prima metà dell'intervallo di percentuale;
- per facilitare il confronto fra diverse applicazioni, in questa tesi si decide di normalizzare il risultato in output dal metodo in modo che ogni app abbia un punteggio di rischio compreso fra due valori fissati (come ad esempio una percentuale da 0 a 100). Facendo così, è tuttavia possibile avere casi in cui applicazioni che dichiarano un solo permesso, ma a rischio molto alto (per esempio uno relativo agli SMS - Sezione 4.3), risultino più rischiose di altre app che richiedono più permessi ma meno pericolosi.

4.5 Metodo basato sul *machine learning*: RiskInDroid

Per tentare di ovviare ai limiti del metodo probabilistico presentato nella Sezione 4.4, in questa sezione viene introdotto RiskInDroid, un nuovo metodo per la generazione di un indice di rischio per le applicazioni Android basato su tecniche di *machine learning*. In particolare, viene spiegato come si ottengono i vettori di *feature* a partire dalle app e viene illustrato il processo che ha portato alla decisione di quali classificatori utilizzare per la generazione del punteggio di rischio.

4.5.1 Estrazione delle *feature*

Una volta ottenuti i permessi per tutte le applicazioni nel *dataset* (utilizzando il procedimento descritto nella Sezione 4.3), è opportuno riorganizzare i risultati in modo da creare vettori di *feature* confrontabili fra loro e facilmente interpretabili dai più comuni metodi di *machine learning*.

In questo elaborato di laurea, le *feature* vengono costruite in modo simile a quanto fatto in [9]: si crea un vettore monodimensionale di capacità P, dove a ogni indice

corrisponde un determinato permesso (ad esempio la posizione 0 coincide con il permesso `CAMERA`, alla posizione 1 corrisponde `INTERNET` ecc.). Per ogni applicazione analizzata viene quindi generato un vettore di P *feature* costituito da 1 e 0 a seconda che, rispettivamente, l'app contenga o meno il permesso relativo all'indice considerato del vettore. Ad esempio, per una determinata app, se all'indice 0 corrisponde il permesso `CAMERA` allora il vettore delle *feature* alla posizione 0 vale 1 se tale app contiene il permesso `CAMERA`, altrimenti vale 0.

Per la costruzione dei vettori delle *feature* vengono impiegati solamente i permessi disponibili nel sistema operativo Android, la cui lista ufficiale è disponibile online all'indirizzo web `http://developer.android.com/reference/android/Manifest.permission.html`. Al momento della stesura della tesi, il numero totale

		Permessi	Applicazione 1	Applicazione 2	...	Applicazione N
Tutti i permessi	Dichiarati	`CAMERA`	0	1		1
		`INTERNET`	1	1		0
		⋮	⋮	⋮	...	⋮
		`READ_SMS`	1	1		0
		`VIBRATE`	0	1		0
	Dichiarati e utilizzati	`CAMERA`	0	1		1
		`INTERNET`	1	1		0
		⋮	⋮	⋮	...	⋮
		`READ_SMS`	1	0		0
		`VIBRATE`	0	0		0
	Dichiarati ma non utilizzati	`CAMERA`	0	0		0
		`INTERNET`	0	0		0
		⋮	⋮	⋮	...	⋮
		`READ_SMS`	0	1		0
		`VIBRATE`	0	1		0
	Non dichiarati ma utilizzati	`CAMERA`	0	0		0
		`INTERNET`	0	0		0
		⋮	⋮	⋮	...	⋮
		`READ_SMS`	0	0		0
		`VIBRATE`	1	0		1

Tabella 4: Rappresentazione delle *feature* costruite con i permessi delle applicazioni

di permessi ammonta a 138, quindi si decide di impostare la lunghezza dei vettori a $P = 138$. RiskInDroid è progettato per essere un metodo generale e applicabile a qualsiasi applicazione Android, pertanto si sceglie di non aggiungere nei vettori delle *feature* alcun permesso specifico definito dalle app. Includendo infatti come *feature* ogni nuovo permesso definito dalle app analizzate, si otterrebbero vettori di *feature* di lunghezza molto maggiore (sperimentalmente, in questa tesi sono stati individuati in totale 397 permessi), con un elevato numero di permessi riferiti ad una quantità di app trascurabile rispetto alla grandezza del *dataset* considerato (nelle prove empiriche effettuate sono stati trovati circa 130 permessi personalizzati dichiarati da meno dello 0.1% dei campioni analizzati).

Per ognuna delle applicazioni esaminate si costruiscono un totale di 5 vettori di *feature*, di cui il quinto è ridondante poiché costituito dalla sequenza dei primi 4 (quindi di lunghezza totale $4P$), ma viene creato per comodità e verrà utilizzato nel seguito del lavoro. Gli altri 4 vettori corrispondono invece alle 4 categorie di permessi fornite dall'output di Approver: c'è pertanto un vettore di *feature* per i permessi dichiarati, uno per i permessi dichiarati e anche utilizzati, uno per quelli dichiarati ma non utilizzati e infine uno per quelli utilizzati senza essere stati dichiarati. La Tabella 4 chiarisce meglio graficamente la struttura dei vettori delle *feature*.

4.5.2 Classificazione

Il problema della classificazione [22] consiste nell'assegnare le diverse osservazioni sperimentali a una certa categoria fra quelle disponibili e già note. Per gli scopi di questa tesi, le osservazioni sperimentali si riferiscono alle applicazioni Android (in particolare ai loro permessi), mentre le possibili classi sono `malware` e `non_malware`. Si tratta di un metodo supervisionato: è necessario avere una collezione di dati di cui si conosce già la relativa classe di appartenenza. Con tali dati si costruisce un modello per consentire la classificazione accurata di nuovi elementi la cui categoria è inizialmente ignota. L'insieme dei dati di partenza viene chiamato *training set*, e generalmente l'accuratezza del modello viene verificata mediante un *testing set*. La Fig. 10 mostra lo schema generale di un classificatore: a partire da un *training set* noto, viene creato un modello che associa a ogni elemento una categoria (in funzione dei suoi attributi o *feature*). Applicando successivamente un *testing set* al classificatore, si può valutare la precisione dello stesso andando a vedere la percen-

tuale di volte in cui la classe predetta dal modello corrisponde a quella realmente presente nel *testing set*. Se i dati di *training* e di *testing* sono stati scelti in maniera opportuna, e l'accuratezza del modello viene ritenuta soddisfacente, si può dare in ingresso al classificatore un elemento la cui categoria è ignota assumendo che la classe predetta sia ragionevolmente uguale a quella reale. In questa tesi si considerano soltanto i classificatori che, oltre ad assegnare una classe, generano in output anche un punteggio di probabilità per quella determinata classe.

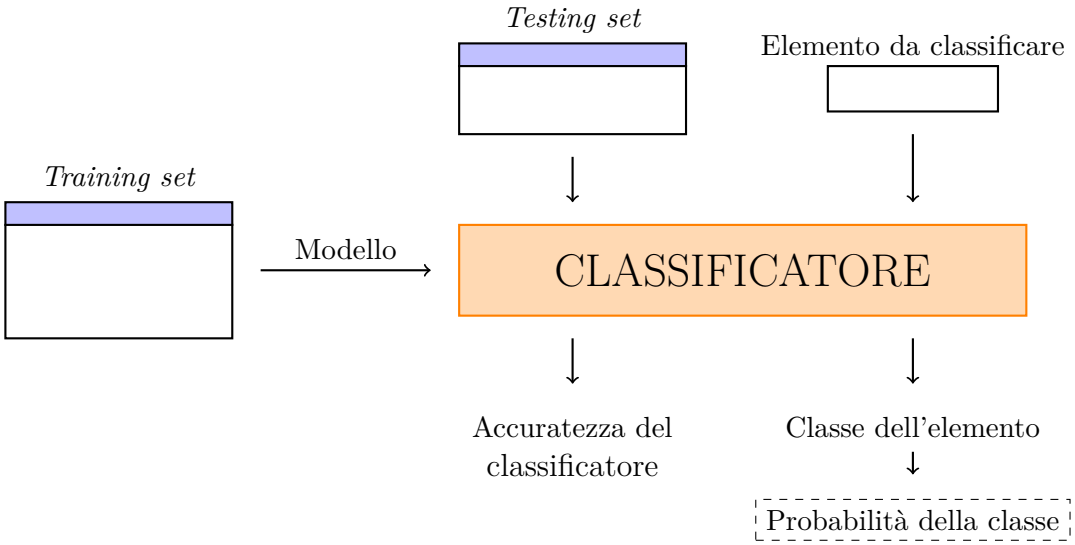


Figura 10: Rappresentazione schematica di un classificatore

Per la creazione di un indice di rischio per le applicazioni viene impiegata la libreria *scikit-learn* per la sua diffusione fra la comunità scientifica, per la relativa semplicità di utilizzo e soprattutto perché contiene molti metodi di classificazione che in output indicano anche la probabilità con cui una determinata classe viene predetta (e non solo il nome della classe stessa). Tale probabilità è una quantità numerica che può essere adoperata per il calcolo di un punteggio di rischio.

La Tabella 5 elenca una lista dei classificatori individuati nella libreria *scikit-learn* che contengono una funzione chiamata `predict_proba`, la quale permette di avere in output una probabilità per ognuna delle possibili classi del modello. Pertanto, un primo indice di rischio (approfondito in seguito) può essere ottenuto considerando la percentuale riferita alla classe `malware` quando il classificatore analizza una certa applicazione: più il punteggio è alto, maggiore è la probabilità che tale applicazione sia maligna, quindi il rischio aumenta. Viceversa, un punteggio basso indica che l'applicazione non fa parte della classe `malware`, pertanto il rischio è minore.

Classificatore	Accuratezza %			
	Set 1	Set 2	Set 3	Media
Support Vector Machines	92.99	93.00	92.77	92.92
Gaussian Naive Bayes	74.88	74.83	79.61	76.44
Multinomial Naive Bayes	90.24	90.01	90.07	90.11
Bernoulli Naive Bayes	86.99	87.20	87.03	87.07
Decision Tree	94.19	94.46	94.39	94.35
Random Forest	95.09	95.25	95.19	95.18
AdaBoost	92.63	93.16	92.84	92.88
Gradient Boosting	93.08	93.19	93.30	93.19
Stochastic Gradient Descent	92.43	92.31	92.36	92.36
Logistic Regression	93.10	93.35	93.29	93.25
Logistic Regression CV	93.27	93.35	93.31	93.31
Nearest Neighbors	78.60	78.79	77.99	78.46
Linear Discriminant Analysis	92.46	92.46	92.33	92.42
Quadratic Discriminant Analysis	61.89	62.52	67.82	64.08
Multilayer Perceptron Neural Network	95.13	95.01	95.16	95.10

Tabella 5: Accuratezza dei classificatori della libreria *scikit-learn* nel classificare le applicazioni appartenenti ai 3 set descritti nella Sezione 4.1

Per misurare l'affidabilità dei metodi della libreria *scikit-learn* nel classificare fra applicazioni benigne e maligne, vengono effettuate alcune prove per poter valutare empiricamente la percentuale di accuratezza di tali metodi, il cui esito è visibile nella Tabella 5. I risultati presentati sono ottenuti analizzando soltanto i permessi dichiarati dalle applicazioni nei 3 set descritti nella Sezione 4.1 ed usando la tecnica della *k–fold cross validation*, impostando $k = 10$ e mantenendo invariati i parametri di default dei classificatori. Si decide di fissare $k = 10$ perché sperimentalmente garantisce buoni risultati [23]; si utilizzano invece i parametri di default dei classificatori poiché si vuole ottenere una comparazione obiettiva dei metodi di classificazione, non influenzata da specifiche configurazioni che potrebbero essere particolarmente adatte alle *feature* impiegate in questo elaborato di laurea.

Come si può notare nella Tabella 5, ci sono alcuni metodi che hanno prestazioni inferiori rispetto agli altri, pertanto vengono scartati a favore di quelli con un'al-

ta percentuale di accuratezza: in questa tesi si imposta empiricamente una soglia minima di precisione a 90% e si continua a lavorare solo con i classificatori che raggiungono questo valore. In alcuni casi, i risultati ottenuti non sono ottimi perché i vettori delle *feature* non sono adatti al tipo di modello scelto: ad esempio, un requisito fondamentale del metodo *Gaussian Naive Bayes* è che i dati in ingresso abbiano una distribuzione di probabilità gaussiana, condizione non verificata dalle *feature* utilizzate in questo elaborato. Altre volte invece la percentuale di accuratezza può essere leggermente aumentata andando ad intervenire sui parametri di costruzione del classificatore, senza avere tuttavia incrementi prestazionali significativi: ad esempio, utilizzando *Nearest Neighbors*, si può modificare il valore di `n_neighbors` in ingresso al modello, ottenendo i risultati presentati nella Tabella 6.

Valore di `n_neighbors`	Accuratezza %			
	Set 1	Set 2	Set 3	Media
2	79.85	77.94	81.44	79.75
default (5)	78.60	78.79	77.99	78.46
10	82.26	82.74	82.10	82.37
20	81.84	82.46	82.06	82.11
50	82.20	82.92	82.56	82.56

Tabella 6: Accuratezza di *Nearest Neighbors* in funzione di `n_neighbors`

Nella Tabella 7 si possono vedere ulteriori informazioni riguardo ai classificatori rimasti dopo l'esclusione di quelli meno efficaci. I risultati sono stati ottenuti sugli stessi 3 set di applicazioni utilizzati nella Tabella 5 ma, per ragioni di spazio, vengono riportati i dati solo dopo aver eseguito la media sui 3 insiemi. La decisione di mostrare solamente i valori medi è giustificata anche dal fatto che la deviazione standard, che indica la variabilità dei dati rispetto alla media, è sempre minore dell'1% per ognuno dei risultati presentati in Tabella 7.

Mentre nella Tabella 5 per calcolare la percentuale di accuratezza viene usata una funzione già inclusa nella libreria *scikit-learn* (`cross_val_score`), nella Tabella 7 le statistiche vengono calcolate manualmente. Una volta allenato il modello del classificatore, per ogni elemento del *testing set* analizzato si controlla la probabilità generata per la classe `malware`: se tale probabilità è maggiore o uguale a 0.5 (50%) e il campione sotto esame è realmente un *malware*, oppure la probabilità è minore

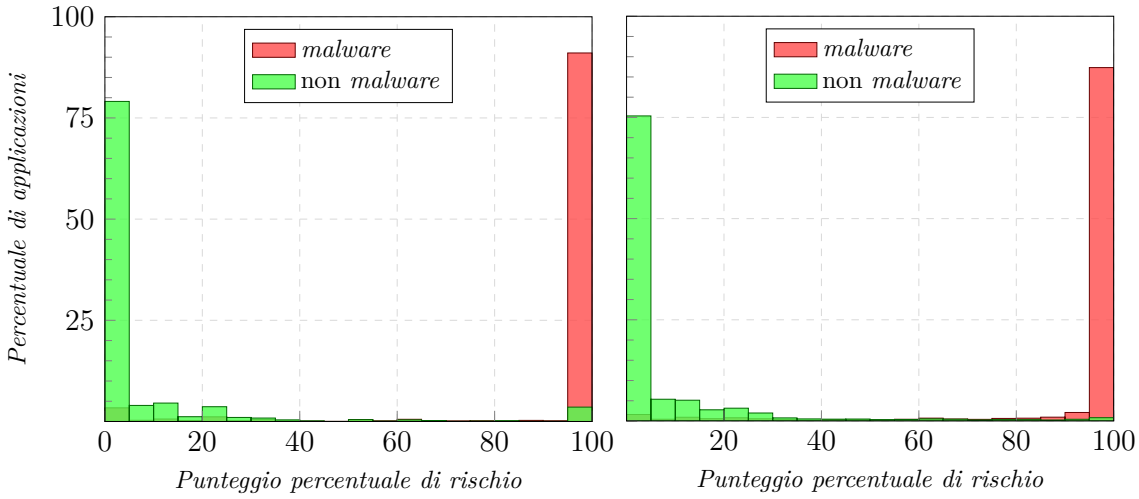
Classificatore	Accuratezza media %	*Malware*		Non *malware*	
		Punteggio medio %	Deviazione standard σ	Punteggio medio %	Deviazione standard σ
Support Vector Machines	93.05	94.76	8.52	7.71	9.77
Multinomial Naive Bayes	88.75	93.52	11.93	12.74	9.79
Decision Tree	94.23	99.17	5.10	3.39	6.99
Random Forest	95.06	97.23	8.38	5.95	9.62
AdaBoost	92.88	52.46	1.66	48.29	1.10
Gradient Boosting	93.19	93.77	8.80	9.77	11.26
Stochastic Gradient Descent	92.46	95.05	9.29	6.78	10.42
Logistic Regression	93.25	94.93	9.40	7.99	10.79
Logistic Regression CV	93.31	95.43	9.05	7.48	10.54
Linear Discriminant Analysis	92.42	96.55	8.55	3.32	6.96
Multilayer Perceptron Neural Network	95.12	98.08	6.64	4.45	7.71

Tabella 7: Punteggi generati dai classificatori

di 0.5 e l'applicazione non è maligna, allora la predizione del classificatore è esatta. Dividendo quindi il numero di predizioni corrette per il numero totale di elementi nel *testing set*, si ottiene la percentuale di accuratezza. Il motivo per cui quest'ultimo risultato è leggermente diverso fra le Tabelle 5 e 7 è che per calcolare l'output del classificatore la libreria *scikit-learn* ricorre a due metodi differenti, a seconda della necessità o meno di sapere anche la probabilità della classe predetta oltre al nome della classe stessa. Nel primo caso (Tabella 5) viene utilizzata la funzione `predict`, che in output indica solo il nome della classe, mentre nella Tabella 7 viene impiegata `predict_proba`, la quale restituisce anche un valore numerico per la classe predetta. Come si può notare osservando le Tabelle 5 e 7, le differenze fra i due metodi sono tuttavia esigue nella maggior parte dei casi. Le altre informazioni presenti in Tabella 7 riguardano il punteggio medio generato dai classificatori e la relativa

deviazione standard σ: questi valori sono calcolati facendo riferimento soltanto alle classi predette correttamente, in modo da non introdurre errori dovuti al metodo di classificazione.

Per la creazione di un indice di rischio per le applicazioni si è interessati ai punteggi generati in output dai classificatori (il cui valor medio è riportato nelle colonne 3 e 5 della Tabella 7). Sebbene tutti i metodi di classificazione presentati nella Tabella 7 abbiano una buona accuratezza, non tutti sono adatti per ricavarne un punteggio di rischio affidabile. È possibile infatti eliminare subito *AdaBoost* dalla lista poiché le probabilità che genera in output sono tutte concentrate intorno al 50%, impedendo dunque di intuire la pericolosità di un'app se si considera solamente questo dato. In generale, come si può osservare sempre in Tabella 7, gli altri classificatori restituiscono punteggi prossimi agli estremi dell'intervallo di percentuale, quindi i *malware* tendono ad avere il massimo livello di rischio mentre le applicazioni benigne propendono per un rischio più vicino a 0, senza avere dunque valori intermedi. Tuttavia,



(a) *Decision Tree* (b) *Multilayer Perceptron Neural Network*

Figura 11: Istogramma del punteggio percentuale di rischio

metodi di classificazione come *Decision Tree* e *Multilayer Perceptron Neural Network* generano in output probabilità troppo sbilanciate verso gli estremi dell'intervallo di percentuale (come visibile in Fig. 11), tanto da essere comparabili direttamente a classificatori binari, in quanto, oltre ad indicare se un'app analizzata risulta o meno maligna, non forniscono alcuna informazione aggiuntiva. Di conseguenza è possibile rimuovere tali metodi dall'elenco in Tabella 7.

Per quanto riguarda invece *Logistic Regression CV*, si tratta di un classificatore molto simile a *Logistic Regression*, che però implementa un meccanismo interno di *cross validation* con lo scopo di trovare il valore ottimo di un parametro C[17], che nel caso della *Logistic Regression* semplice deve essere fornito manualmente (nel caso tale parametro venga omesso, di default si ha $C = 1.0$). Dai dati visibili in Tabella 7 si può osservare che c'è una differenza minima fra i risultati di questi due metodi, pertanto, data la similarità che c'è tra i due classificatori, in questa tesi si decide di proseguire solamente con l'utilizzo di *Logistic Regression*, scartando *Logistic Regression CV*. Un'altra motivazione per questa scelta deriva dal confronto dei tempi di esecuzione: nelle prove empiriche effettuate si è notato che, per fare il *training* del modello, *Logistic Regression* impiega meno di un secondo, mentre per *Logistic Regression CV* sono necessari circa 15 secondi, senza tuttavia migliorare in modo significativo il risultato finale.

Similmente a quanto fatto per la Tabella 5, in questo elaborato di laurea si stabilisce empiricamente un intervallo limite compreso tra 7% e 95%, con lo scopo di escludere i classificatori che restituiscono probabilità troppo sbilanciate verso gli estremi dell'intervallo di percentuale. Applicando questo filtro alla Tabella 7, rimangono da considerare solamente i seguenti 4 metodi di classificazione: *Support Vector Machines*, *Multinomial Naive Bayes*, *Gradient Boosting* e *Logistic Regression*. Pertanto, prima di procedere ulteriormente nella trattazione della tesi, è utile soffermarsi a esaminare più nel dettaglio questi 4 classificatori.

Support Vector Machines (SVM)

Le macchine a vettori di supporto (SVM) [24] sono un insieme di metodi di apprendimento supervisionato per la risoluzione di problemi di classificazione o di regressione. Si tratta di classificatori binari non probabilistici il cui scopo è la ricerca dell'iperpiano di separazione ottimale fra le due possibili classi all'interno dello spazio delle *feature*, ma è possibile estenderne il funzionamento anche a problemi in cui il numero di classi da predire è maggiore di 2. Nel caso in cui i dati in input non siano direttamente linearmente separabili, si può ricorrere a funzioni chiamate *kernel* che mappano i dati iniziali in uno spazio di dimensione superiore dove invece è possibile trovare un iperpiano di separazione, pertanto le SVM si adattano bene anche a pro-

[17] `http://scikit-learn.org/dev/modules/linear_model.html#logistic-regression`

blemi di classificazione non lineare. Le *Support Vector Machines* sono dunque dei classificatori molto potenti che permettono di ricondursi alla classificazione lineare anche quando si devono risolvere problemi molto complessi. Per gli scopi di questo lavoro non è tuttavia necessaria una rigorosa trattazione matematica delle macchine a vettori di supporto, quindi ci si limita a fare un esempio grafico che chiarisca meglio il funzionamento delle SVM in un caso di classificazione binaria lineare: come si può vedere in Fig. 12, in questo semplice esempio è la retta h a dividere le due classi con il margine di separazione massimo.

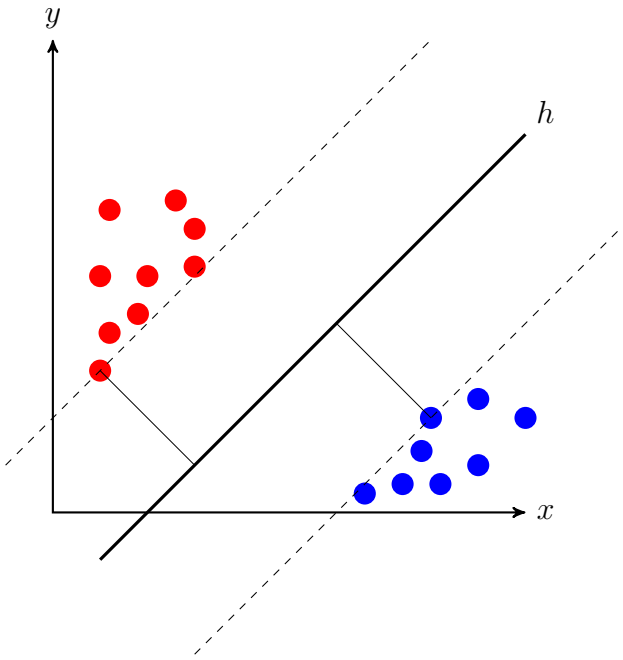


Figura 12: Esempio grafico di una macchina a vettori di supporto

Multinomial Naive Bayes

I classificatori bayesiani [25] sono classificatori lineari efficienti e relativamente semplici da realizzare. Si tiene conto del fatto che spesso la relazione tra i valori delle *feature* e quello della classe non è di tipo deterministico; pertanto, per descrivere meglio tale relazione, si ricorre ad un modello probabilistico basato sul teorema di Bayes. In particolare, durante lo svolgimento di questa tesi si utilizzano i classificatori bayesiani naive, ovvero si ritiene che gli attributi degli elementi considerati siano indipendenti fra loro, ipotesi poco realistica che però generalmente si rivela efficace nei problemi pratici. Ci sono più tipi di classificatori bayesiani e si distinguono in base al tipo di distribuzione di probabilità che utilizzano; un'ipotesi molto

comune è che le *feature* abbiano una distribuzione di probabilità gaussiana, pertanto il classificatore *Gaussian Naive Bayes* è molto diffuso. Tuttavia, come si può vedere in Tabella 5, in questo elaborato di laurea i risultati migliori si ottengono invece assumendo che le *feature* considerate abbiano una distribuzione probabilistica multinomiale.

Logistic Regression

La regressione logistica [26] è un metodo di classificazione che consiste nella creazione di un modello non lineare per studiare la relazione causale esistente tra una variabile dipendente y binaria e una o più variabili indipendenti x, che possono assumere valori sia quantitativi sia qualitativi. Si tratta di un problema simile alla regressione lineare multipla, ma in questo caso la variabile y è vincolata ai valori 0 e 1, quindi è necessario ricorrere a una funzione particolare detta sigmoide (Fig. 13) per rappresentare il legame fra x e y (impiegando la regressione lineare, la variabile y non sarebbe invece limitata ma potrebbe variare fra $(-\infty, +\infty)$).

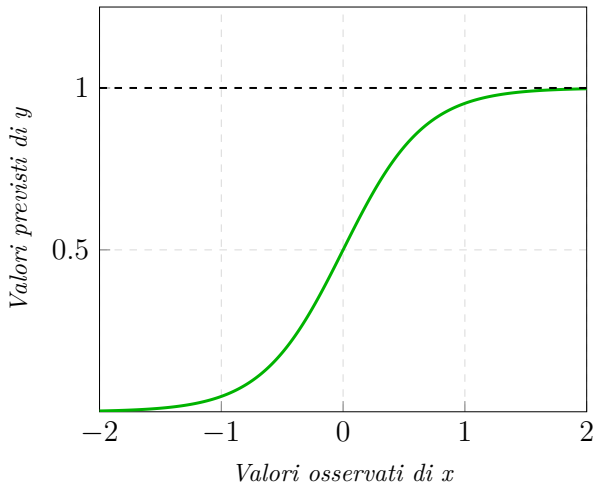


Figura 13: Esempio di funzione sigmoide utilizzata nella regressione logistica

Gradient Boosting

Il *Gradient Boosting* [27] è un metodo *ensemble* di classificazione e di regressione ottenuto dall'accostamento sequenziale di più classificatori. Viene chiamato *ensemble* perché è creato dalla combinazione di più metodi che singolarmente non sono sempre efficaci, ma che insieme garantiscono buoni risultati (in generale come metodo

di base si utilizzano gli alberi decisionali). Si tratta di un procedimento iterativo dove ad ogni passo n viene aggiunto un nuovo classificatore di base allo scopo di migliorare il modello complessivo. Considerando come $valore_{predetto} = F(x)$ la funzione del modello di classificazione, tale funzione viene aggiornata ad ogni iterazione allo scopo di migliorare la predizione globale ponendo $F_{n+1}(x) = F_n(x) + h_n(x)$, dove $h_n(x)$ è un classificatore di base creato in modo da minimizzare l'errore di predizione di $F_n(x)$. Se si riuscisse a creare un modello perfetto, ad un certo passo n si avrebbe $F_n(x) + h_n(x) = valore_{atteso}$; riscrivendo $h_n(x)$ come $h_n(x) = valore_{atteso} - F_n(x)$ si può intuire che lo scopo di ogni classificatore di base aggiunto è modellare $valore_{atteso} - F_n(x)$ così da ottenere $F_{n+1}(x) = valore_{atteso}$. Definendo la funzione di errore $L(valore_{atteso}, F(x)) = \frac{1}{2}(valore_{atteso} - F(x))^2$ e calcolandone il gradiente rispetto ad $F(x)$ si ottiene $F(x) - valore_{atteso}$ che è anche uguale a $-h(x)$, pertanto si ottiene che $F_{n+1}(x) = F_n(x) - \nabla L$, quindi ci si è ricondotti a creare un modello utilizzando la tecnica di discesa del gradiente, da cui deriva il nome *Gradient Boosting*. In questo caso si è considerata la funzione di errore L ai minimi quadrati, ma è possibile utilizzare anche altre funzioni di errore (con l'unico vincolo di essere differenziabili).

A questo punto della tesi si è stabilito empiricamente quali classificatori della libreria *scikit-learn* utilizzare per la creazione di un indice di rischio per le applicazioni Android. Nella Tabella 7 vengono riportati i risultati aggregati ottenuti analizzando i 3 set di app descritti nella Sezione 4.1; tuttavia, per poter calcolare un punteggio di rischio anche per nuove applicazioni, è necessario prima di tutto allenare il modello di classificazione, e per fare ciò bisogna scegliere quale dei 3 set impiegare per eseguire il *training*. La Tabella 8 mostra più nel dettaglio i risultati per ognuno dei 3 set descritti nella Sezione 4.1 (impiegando la 10–*fold cross validation*), in modo da facilitare la scelta del *training set* da utilizzare nel seguito dell'elaborato di laurea. Come si può infatti notare in Tabella 8, le differenze fra i diversi set di ogni classificatore sono pressoché inesistenti, pertanto si decide di fare il *training* dei modelli di classificazione utilizzando solamente il set 1, in quanto si è visto empiricamente come i risultati ottenuti siano più generali.

Il passo conclusivo per la creazione di un indice di rischio consiste nel combinare i punteggi calcolati dai 4 classificatori scelti in un unico valore. Per fare ciò si decide di utilizzare la media aritmetica fra le probabilità (riferite alla classe *malware*) generate

Classificatore		*Malware*		Non *malware*	
		Punteggio medio %	Deviazione standard σ	Punteggio medio %	Deviazione standard σ
Support Vector Machines	Set 1	94.99	8.07	7.77	9.84
	Set 2	94.71	8.66	7.72	9.68
	Set 3	94.57	8.79	7.66	9.80
	Media	94.76	8.52	7.71	9.77
Multinomial Naive Bayes	Set 1	93.33	12.16	12.89	9.93
	Set 2	93.51	11.99	12.52	9.70
	Set 3	93.73	11.65	12.80	9.74
	Media	93.52	11.93	12.74	9.79
Gradient Boosting	Set 1	93.90	8.59	9.84	11.23
	Set 2	93.63	8.96	9.66	11.20
	Set 3	93.77	8.86	9.80	11.34
	Media	93.77	8.80	9.77	11.26
Logistic Regression	Set 1	95.06	9.16	7.96	10.61
	Set 2	94.86	9.46	8.01	10.89
	Set 3	94.88	9.57	8.00	10.86
	Media	94.93	9.40	7.99	10.79

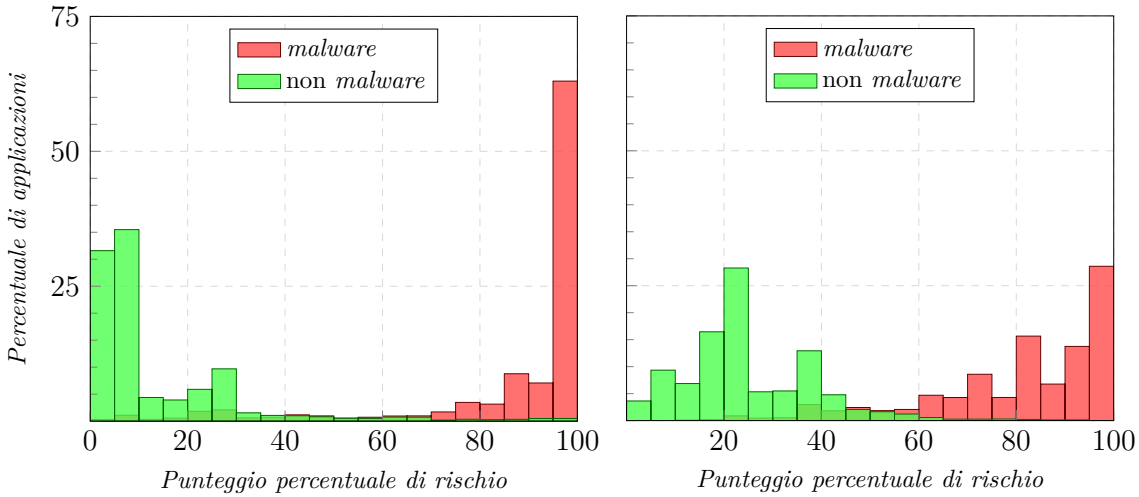
Tabella 8: Punteggi dettagliati generati dai classificatori

in output dai 4 metodi di classificazione considerati. Denotando dunque con R_A il punteggio percentuale di rischio associato ad un'applicazione e con P_i l'output dell'i-esimo classificatore (la probabilità relativa alla classe *malware*), si può scrivere l'Eq. (5) per il calcolo del rischio:

$$R_A = \frac{1}{N} \sum_{i}^{N} P_i \qquad \text{con} \quad N = 4 \tag{5}$$

Implementando in codice l'Eq. (5) e applicandola all'insieme di app presenti nel set 1, si ottiene l'istogramma presentato in Fig. 14(a). Nonostante sia meno marcato

rispetto a quanto visto in Fig. 11, si verifica ancora una volta lo sbilanciamento dei punteggi verso gli estremi dell'intervallo di percentuale. Si decide pertanto di ridimensionare l'indice di rischio calcolato precedentemente in modo da ottenere una distribuzione più uniforme, e per fare ciò si impiega una funzione descritta in Fig. 15 ed ispirata alla funzione `logit` [26], che in questa tesi viene chiamata "funzione di ridimensionamento". La funzione `logit` è l'inversa della funzione sigmoide vista nel caso della regressione logistica (Fig. 13), è definita in $[0, 1]$ (perché la variabile dipendente è una probabilità) e può assumere valori in tutto l'intervallo dei numeri reali.



(a) Prima del ridimensionamento (b) Dopo il ridimensionamento

Figura 14: Istogramma dell'indice di rischio finale

Rispetto alla funzione `logit`, nella funzione di ridimensionamento utilizzata in questo elaborato i coefficienti sono impostati in modo che la funzione intersechi i punti $(0, 0)$, $(50, 50)$ e $(100, 100)$; in questo caso specifico il dominio della funzione viene considerato $[0, 100]$ in quanto si tratta di termini percentuali, di conseguenza anche la funzione assumerà valori in $[0, 100]$. L'Eq. (6) descrive la funzione utilizzata e il grafico della stessa è visibile in Fig. 15.

$$f(x) = \frac{50}{\log(101)} \log\left(\frac{1+x}{101-x}\right) + 50 \tag{6}$$

Lo scopo della funzione di ridimensionamento in Fig. 15 è quello di rendere più uniforme la distribuzione della percentuale di rischio presentata in Fig. 14(a). In particolare, siccome i punteggi generati dai classificatori tendono a raggrupparsi in

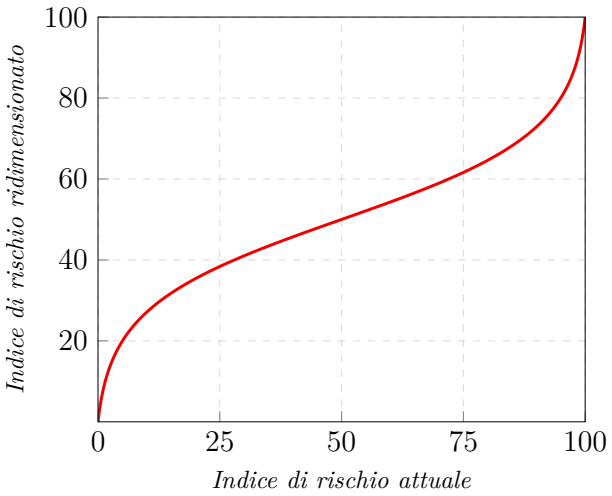


Figura 15: Funzione di ridimensionamento dell'indice di rischio

prossimità degli estremi dell'intervallo di percentuale, la funzione di ridimensionamento "dilata" questa regione e "restringe" la zona centrale dell'intervallo, dove invece la concentrazione di punteggi è più bassa. In questo modo diventa più facile confrontare le applicazioni ad alto/basso indice di rischio: ad esempio, se per una coppia di app i classificatori generano rispettivamente 99.95% e 99.75%, applicando l'Eq. (6) si ottengono invece i punteggi 99.47% e 97.56%; pertanto, aumenta la differenza dei valori di rischio e risulta più immediato intuire quale delle due applicazioni sia più rischiosa. Adoperando la funzione descritta dall'Eq. (6) sui dati visualizzabili in Fig. 14(a), è possibile ottenere una distribuzione più uniforme dei punteggi di rischio, come dimostra il nuovo istogramma in Fig. 14(b).

5 Risultati

Nella Sezione 4.5 viene presentato RiskInDroid concentrandosi maggiormente sugli aspetti teorici, spiegando le motivazioni per cui sono stati scelti determinati classificatori al posto di altri. In questa sezione si vuole invece testare in maniera più approfondita il metodo RiskInDroid, valutando tutte le categorie di permessi (e non solamente quelli dichiarati come fatto finora) ed estendendo le prove a tutto il *dataset* a disposizione (Tabella 1). L'ambiente di test impiegato per eseguire le prove empiriche è descritto in Tabella 9.

CPU	Intel i7-3635QM
RAM	16 GB
Sistema Operativo	Windows 10
Linguaggio di programmazione	Python 3.5.2

Tabella 9: Ambiente di test utilizzato per le prove empiriche

Dal momento che per RiskInDroid si utilizzano le probabilità generate dai classificatori, è necessario impostare una soglia per poter distinguere i *malware* dal resto delle app. In questa tesi, tale soglia viene fissata a 50% (lo stesso valore usato nella Tabella 7 per determinare l'accuratezza dei classificatori). Il fatto di aver stabilito una soglia di probabilità non va inteso come un metodo generale per la distinzione fra *malware* e applicazioni benigne, che esula dagli scopi di questo elaborato, ma è necessario per poter capire durante i test se un'app, di cui si conosce già la natura, viene classificata in modo corretto. Infatti, nonostante alcune eccezioni, si assume che un *malware* abbia un punteggio di rischio sopra il 50%, mentre le applicazioni benigne dovrebbero avere un indice di rischio inferiore a tale soglia. Se dalle prove sperimentali risulta inoltre che ad un alto punteggio di rischio corrisponde, nella maggior parte dei casi, un vero *malware*, allora si ha un'ulteriore conferma della correttezza e dell'affidabilità della metodologia proposta.

Prima di proseguire nell'applicare il metodo RiskInDroid sull'intero *dataset* a disposizione, è interessante valutare la possibilità di ottenere risultati più accurati impiegando tutte le categorie di permessi viste nella Sezione 4.3. Finora infatti sono stati utilizzati soltanto i permessi dichiarati per analizzare le prestazioni dei classificatori. Nella Tabella 10 vengono quindi mostrate le statistiche di accuratezza relative a tutte le tipologie di permessi, utilizzando le *feature* descritte nella

Sezione 4.5.1. Per ottenere questi risultati si impiega la 10–*fold cross validation* sull'insieme di applicazioni contenute nel set 1 (nella Sezione 4.5 si è visto empiricamente che è sufficiente fare le prove su soltanto uno dei 3 set introdotti nella Sezione 4.1).

Categoria di permessi	Accuratezza %
Dichiarati	92.86
Dichiarati e utilizzati	88.39
Dichiarati ma non utilizzati	90.91
Non dichiarati ma utilizzati	76.60
Tutti i permessi	94.94

Tabella 10: Accuratezza in base alle categorie di permessi

Osservando la Tabella 10 si può vedere che l'accuratezza di RiskInDroid dipende soprattutto dai permessi dichiarati; tuttavia, combinando insieme tutte le categorie di permessi, si riesce ad ottenere un valore di accuratezza maggiore rispetto al considerare ogni gruppo singolarmente. Sebbene l'estrazione di altre categorie di permessi sia un'operazione più complessa in confronto al ricavare solamente i permessi dichiarati, nelle prove empiriche effettuate nella Sezione 5.2 si è notato che ciò non penalizza in modo significativo i tempi di esecuzione di RiskInDroid, se non per quanto riguarda la fase di *training* dei modelli di classificazione (che tuttavia è un'operazione che va eseguita una volta sola). In questa tesi si decide pertanto di utilizzare tutte le categorie di permessi disponibili per la creazione di un indice di rischio, in quanto si è visto empiricamente che in questo modo si ottiene un'accuratezza maggiore.

Una volta definite le categorie di permessi da utilizzare, è necessario stabilire quali e quanti campioni del *dataset* riservare per il *training* dei classificatori. Seguendo la strada intrapresa finora, si decide di impiegare il set 1 di applicazioni per allenare i modelli di classificazione, ricordando che si tratta di un insieme formato per metà dalla collezione di *malware* presenti nel *dataset* e per l'altra metà da un gruppo di app estratte in modo pseudo-casuale fra le applicazioni scaricate dal Google Play Store, come già visto nella Sezione 4.1. Per valutare sperimentalmente la qualità del *training set* scelto si ricorre alla 10–*fold cross validation*, i cui risultati sono presen-

tati sotto forma di istogramma nella Fig. 16; in questa circostanza i dati ottenuti vengono mostrati anche all'interno di una matrice di confusione (Tabella 11).

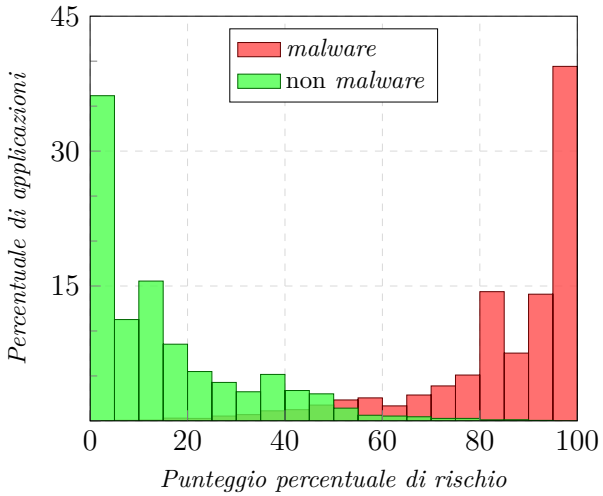


Figura 16: Istogramma dell'indice di rischio generato da RiskInDroid utilizzando tutte le categorie di permessi disponibili

App totali: 13 414		Classe predetta	
		Malware	Non *malware*
Classe reale	*Malware*	6 295	412
	Non *malware*	263	6 444

Tabella 11: Matrice di confusione del *training set* utilizzato

La **matrice di confusione** [22] è una tabella che descrive in maniera sintetica e intuitiva le prestazioni di un modello di classificazione. In questo caso specifico, sulla diagonale principale è possibile visualizzare il numero di applicazioni classificate in modo corretto: i veri positivi (*malware* correttamente classificati come tali) ed i veri negativi (applicazioni benigne riconosciute correttamente). Nelle altre due celle si trovano invece i falsi positivi (app benigne scambiate per *malware*) ed i falsi negativi (*malware* classificati incorrettamente come applicazioni benigne).

Dalla Tabella 11 è possibile calcolare alcuni indici statistici per ottenere una valutazione empirica dell'affidabilità di RiskInDroid e del *training set* selezionato:

- **Accuratezza** = 0.949, denota il rapporto fra il numero di applicazioni correttamente classificate (sia maligne sia benigne) e il numero totale di app analizzate;
- **Precisione** = 0.959, indica la proporzione fra i *malware* classificati correttamente e il totale di applicazioni classificate come *malware*;
- ***Recall*** = 0.938, esprime il rapporto fra i *malware* classificati correttamente e il totale di *malware* esaminati.

Visti i buoni risultati ottenuti empiricamente per il *training set* considerato, è possibile procedere con il calcolo del punteggio di rischio per tutte le applicazioni presenti nel *dataset* a disposizione (il *training set* è stato fissato, pertanto il resto delle app costituiscono il *testing set*). Nella Fig. 17 si può quindi vedere l'istogramma con gli indici di rischio per la collezione completa di app provenienti dal Google Play Store (ad esclusione del gruppo di applicazioni già incluse nel *training set*).

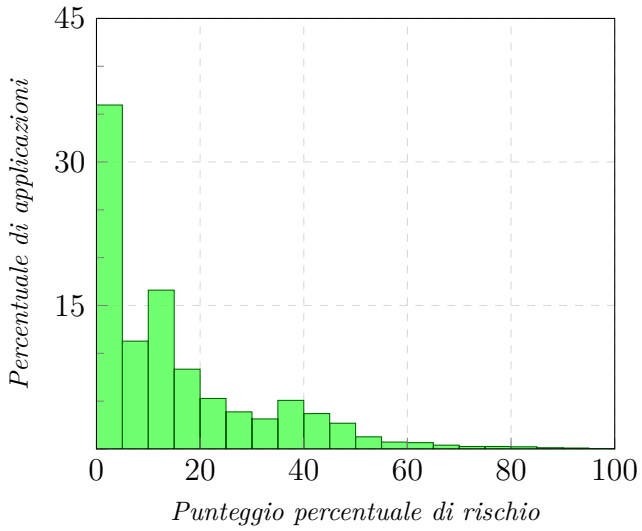
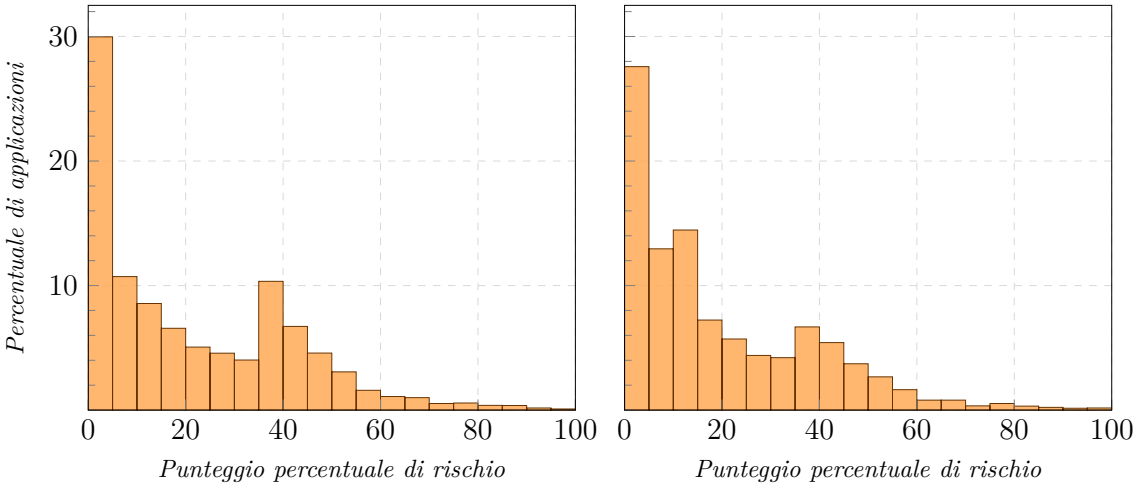


Figura 17: Istogramma dell'indice di rischio generato da RiskInDroid per le app provenienti dal Google Play Store

Nella Fig. 18 è possibile invece osservare l'istogramma con i punteggi di rischio per i campioni del *dataset* provenienti da canali non ufficiali. A prescindere dalla fonte da cui sono state scaricate le applicazioni, esaminando gli istogrammi in Fig. 17 e 18 si può intuire che si tratta perlopiù di app non malevole.



(a) Aptoide (b) Uptodown

Figura 18: Istogramma dell'indice di rischio generato da RiskInDroid per le app provenienti da fonti non ufficiali

5.1 Confronto con antivirus

Per concludere l'analisi sperimentale di RiskInDroid, si vuole determinare la reale pericolosità delle applicazioni per le quali è stato individuato un punteggio di rischio alto da parte del metodo proposto in questa tesi. Per fare ciò, si raccolgono tutte le app del *dataset* per le quali RiskInDroid ha calcolato un valore di rischio superiore al 75%, e si analizzano successivamente con una suite di più di 50 antivirus online resa disponibile gratuitamente da VirusTotal[18]. Lo scopo di questo esperimento è di valutare il livello di affidabilità del metodo di analisi di RiskInDroid rispetto ai tradizionali approcci *signature-based* generalmente utilizzati dagli antivirus. I risultati ottenuti da questo confronto sono mostrati nella Tabella 12, dove il numero di *flag* indica quanti degli antivirus interrogati da VirusTotal riconoscono una determinata

Fonte applicazioni	# app con rischio $\geq 75\%$	# *flag* ≥ 1	# *flag* ≥ 3	# *flag* ≥ 15
Google Play Store	792	173	101	30
Aptoide	110	22	11	5
Uptodown	103	29	14	4

Tabella 12: Statistiche ottenute esaminando le applicazioni più rischiose con VirusTotal

[18] `http://www.virustotal.com`

applicazione come *malware*. I numeri riferiti alle app provenienti dal Google Play Store sono più alti perché le applicazioni esaminate provengono in massima parte da tale *store*; se però, al posto di considerare i valori assoluti, si calcolano invece le percentuali, si ottengono risultati simili per tutte le categorie di applicazioni presenti nella Tabella 12. In particolare, si può notare che in ogni categoria circa il 4% delle app più rischiose sia riconosciuto come *malware* da almeno 15 antivirus. In conclusione, tale analisi comparativa suggerisce che l'approccio adottato da RiskInDroid sia promettente e che l'indice di rischio generato sia affidabile, in quanto fra le applicazioni più rischiose sono stati individuati alcuni *malware* da un buon numero di antivirus.

Dal momento che RiskInDroid risulta empiricamente attendibile per la definizione del rischio delle applicazioni Android, un suo possibile impiego consiste nel valutare potenziali *zero-day malware*. Si tratta di *malware* che sfruttano vulnerabilità di sistema non ancora note pubblicamente o che agiscono con modalità mai viste in precedenza; in questo caso gli approcci tradizionali *signature-based* sono poco efficaci in quanto il loro funzionamento si basa sulla conoscenza pregressa delle "firme" di *malware* già noti, e pertanto non riescono ad individuare i *malware* che presentano comportamenti nuovi rispetto a quelli già conosciuti.

5.2 Considerazioni sui tempi di esecuzione

Finora è stata valutata solamente la qualità e l'affidabilità dei risultati generati da RiskInDroid. Per avere un quadro più completo, in questa sezione si analizzano invece le prestazioni di RiskInDroid per quanto riguarda i tempi di esecuzione, utilizzando la configurazione hardware/software descritta nella Tabella 9.

Il funzionamento di RiskInDroid può essere suddiviso in 3 passi principali: estrazione dei vettori di *feature* dalle applicazioni, *training* dei modelli di classificazione (eseguito una volta sola) e calcolo effettivo del punteggio di rischio delle app che si vogliono analizzare. Come già visto più in dettaglio nella Sezione 4.3, per l'estrazione delle *feature* viene impiegato il modulo *Permission Checker* di Approver. Nella Tabella 13 è possibile osservare i tempi medi e la deviazione standard (σ) in millisecondi necessari per ottenere le *feature* delle applicazioni nel *dataset*, sia nel caso in cui si decida di estrarre tutte le categorie di permessi, effettuando dunque un'analisi statica completa, sia quando si vogliono invece ottenere solamente i per-

Fonte applicazioni	Analisi completa		Analisi del *Manifest*	
	t_{medio} [ms]	σ [ms]	t_{medio} [ms]	σ [ms]
Raccolta *malware*	47	40	30	16
Google Play Store	99	62	34	17
Aptoide	127	91	50	43
Uptodown	127	88	47	40

Tabella 13: Prestazioni del modulo *Permission Checker* di Approver

messi dichiarati nel *Manifest* (ovvero nel file `AndroidManifest.xml` contenuto in ogni app). Le differenze nelle statistiche delle categorie di applicazioni sono imputabili alla diversa grandezza media delle app (le app più grandi richiedono infatti più tempo per essere esaminate), tuttavia si tratta di valori trascurabili se comparati al tempo medio necessario per fare il *training* dei modelli di classificazione utilizzati da RiskInDroid, visualizzati nella Tabella 14 (tali valori corrispondono alla media dei risultati ottenuti dopo 10 iterazioni). In questo caso si può notare come, usando tutte le categorie di permessi, ci sia un divario abbastanza marcato rispetto all'utilizzare soltanto i permessi dichiarati. Questa differenza è dovuta soprattutto alle *Support Vector Machines* e al *Gradient Boosting*, in quanto gli altri due classificatori (*Multinomial Naive Bayes* e *Logistic Regression*) hanno prestazioni quasi uguali a prescindere dal numero di permessi impiegati.

Classificatore	Permessi dichiarati		Tutti i permessi	
	t_{medio} [ms]	σ [ms]	t_{medio} [ms]	σ [ms]
Support Vector Machines	43 460	60	97 170	870
Multinomial Naive Bayes	32	4	53	11
Gradient Boosting	5 620	52	21 806	403
Logistic Regression	81	9	188	11
Totale	49 193	67	119 220	890

Tabella 14: Tempo medio necessario per il *training* dei classificatori di RiskInDroid

Una volta eseguito il *training* dei classificatori presenti in RiskInDroid ed assumendo di aver già estratto le *feature* dalle applicazioni, calcolare i punteggi di rischio delle

app risulta un'operazione estremamente veloce, nell'ordine di pochi millisecondi, pertanto non si ritiene necessario riportare tali valori in una tabella. È importante sottolineare il fatto che il *training* dei modelli di classificazione è un'operazione che va eseguita una volta sola prima di analizzare le applicazioni, pertanto è accettabile avere dei tempi di esecuzione in questa fase se poi si ottengono benefici in termini di accuratezza ed affidabilità.

6 Conclusioni e Sviluppi Futuri

In questa tesi è stato proposto RiskInDroid, un metodo per la generazione di un punteggio di rischio per le applicazioni Android basato su tecniche di *machine learning*. Prima però di definire RiskInDroid sono state valutate altre due tecniche con lo scopo di calcolare un indice di rischio per le app. In un primo momento, si è tentato di sviluppare un metodo che sfruttasse le informazioni sulle vulnerabilità presenti nell'elenco CVE, che però si è rivelato inefficace a causa di un esiguo numero di riscontri ottenuti. Come alternativa, si è provato a migliorare una metodologia già esistente nella letteratura scientifica [4], ottenendo risultati interessanti ma che presentavano ancora alcune caratteristiche indesiderate (come ad esempio un punteggio di rischio mediamente alto anche per le applicazioni benigne). Dal momento che nessuno dei due metodi proposti inizialmente si è rivelato soddisfacente, è stato ideato un nuovo approccio per il calcolo del rischio, a cui è stato attribuito il nome RiskInDroid. Una estensiva analisi sperimentale su 116 541 app benigne e 6 707 *malware* ha dimostrato che RiskInDroid permette di calcolare indici di rischio affidabili con buone prestazioni computazionali.

L'obiettivo di creare un metodo per l'analisi del rischio delle applicazioni Android è stato quindi raggiunto: si è riusciti ad implementare un punteggio di rischio che risulta corretto (la percentuale di accuratezza ha un valore alto, 94.9%) ed affidabile (sono state eseguite prove empiriche su un campione statistico significativo formato da un totale di 123 248 elementi). Uno dei fattori che hanno contribuito all'ottenimento di questi risultati è l'aver considerato tutte le categorie di permessi riferiti alle app e non solo quelli dichiarati, come invece è stato fatto nella maggior parte degli altri lavori presenti nella letteratura scientifica. Sebbene lo scopo di questo lavoro non sia la classificazione binaria di *malware*, le app benigne esaminate per cui risultava un alto valore di rischio sono state analizzate anche tramite il portale VirusTotal, per determinare se esiste una qualche relazione tra un punteggio di rischio elevato e la reale pericolosità di un'applicazione. Si è scoperto che l'indice calcolato da RiskInDroid è ragionevole, in quanto fra le app (in teoria benigne) a rischio maggiore erano presenti alcuni *malware*. Tuttavia, RiskInDroid non è un classificatore binario di *malware* e non deve essere impiegato in tale contesto. Infine, è importante osservare che un'app non deve necessariamente appartenere alla categoria *malware* per essere rischiosa. Si consideri, ad esempio, un'applicazione

legittima per la gestione dei contatti e degli SMS, che necessita inoltre di una connessione ad Internet per fornire alcune funzionalità aggiuntive; pur non trattandosi di un *malware*, tale applicazione è comunque una potenziale minaccia per la privacy dell'utente, pertanto è lecito aspettarsi che abbia un elevato punteggio di rischio.

Come già accennato nell'introduzione, il problema della sicurezza del sistema Android è molto attuale, pertanto è bene disporre di quanti più strumenti possibile che aiutino a salvaguardare l'utente dalle applicazioni pericolose. In questo contesto si inserisce anche Approver, un prodotto software professionale per l'analisi completa delle applicazioni mobili, sviluppato da Talos S.r.l.s (`http://www.talos-security.com/`) e che verrà rilasciato ad ottobre 2016. RiskInDroid si presenta dunque come un *tool* implementato e funzionante per il calcolo di un punteggio di rischio per le applicazioni Android, che verrà integrato come componente di Approver, di cui utilizza già il modulo *Permission Checker* per ottenere i vettori di *feature* delle applicazioni esaminate.

Sviluppi Futuri

Questo elaborato di laurea lascia spazio ad ulteriori approfondimenti e sviluppi futuri. Si è visto che i vettori di *feature* utilizzati per allenare i modelli di classificazione permettono di ottenere empiricamente buoni risultati; si potrebbe dunque provare a migliorare ulteriormente tali risultati aumentando il numero e il tipo di *feature* impiegate, aggiungendo altre informazioni (oltre ai permessi) ricavate dall'analisi statica delle app, come ad esempio le chiamate ad API sospette oppure indirizzi di rete e URL individuati nel codice sorgente decompilato delle applicazioni (analogamente a quanto fatto in [12]). Potrebbe infine essere interessante esplorare anche tecniche di analisi dinamica (come ad esempio Crowdroid [7]), non prese in considerazione durante lo svolgimento di questa tesi, al fine di ottenere un metodo ancora più preciso ed affidabile di calcolo del rischio per le applicazioni Android.

Riferimenti bibliografici

[1] Gartner. Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016. [Online]. Available: http://www.gartner.com/newsroom/id/3415117

[2] C. S. Gates, N. Li, H. Peng, B. Sarma, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Generating Summary Risk Scores for Mobile Applications," *IEEE Transactions on dependable and secure computing*, vol. 11, no. 3, pp. 238–251, 2014.

[3] H. Hao, Z. Li, and H. Yu, "An Effective Approach to Measuring and Assessing the Risk of Android Application," in *Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on.* IEEE, 2015, pp. 31–38.

[4] Y. Wang, J. Zheng, C. Sun, and S. Mukkamala, "Quantitative Security Risk Assessment of Android Permissions and Applications," in *IFIP Annual Conference on Data and Applications Security and Privacy.* Springer, 2013, pp. 226–241.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] Talos S.r.l.s. (2016) Approver. [Online]. Available: http://www.talos-security.com/

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices.* ACM, 2011, pp. 15–26.

[8] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM conference on Computer and communications security.* ACM, 2009, pp. 235–245.

[9] Z. Aung and W. Zaw, "Permission-Based Android Malware Detection," *International Journal of Scientific and Technology Research*, vol. 2, pp. 228–234, 2013.

[10] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "PUMA: Permission Usage to detect Malware in Android," in *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*. Springer, 2013, pp. 289–298.

[11] X. Liu and J. Liu, "A Two-layered Permission-based Android Malware Detection Scheme," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*. IEEE, 2014, pp. 142–148.

[12] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *NDSS*, 2014.

[13] S. Li, T. Tryfonas, G. Russell, and P. Andriotis, "Risk Assessment for Mobile Systems Through a Multilayered Hierarchical Bayesian Network," 2016.

[14] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[15] A. Niculescu-Mizil and R. Caruana, "Predicting Good Probabilities With Supervised Learning," in *Proceedings of the 22nd international conference on Machine learning*. ACM, 2005, pp. 625–632.

[16] Contagio mobile malware mini dump. [Online]. Available: http://contagiominidump.blogspot.com/

[17] N. Husted. (2011) Android malware dataset. [Online]. Available: http://cgi.cs.indiana.edu/~nhusted/dokuwiki/doku.php?id=datasets

[18] A. Bhatia. Collection of android malware samples. [Online]. Available: http://github.com/ashishb/android-malware

[19] R. Wiśniewski and C. Tumbleson. Apktool - A tool for reverse engineering Android apk files. [Online]. Available: http://ibotpeaches.github.io/Apktool/

[20] B. Gruver. Smali - Assembler/Disassembler for the dex format. [Online]. Available: http://github.com/JesusFreke/smali/

[21] A. Desnos. Androguard - Reverse engineering, Malware and goodware analysis of Android applications. [Online]. Available: http://github.com/androguard/androguard/

[22] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated, 2014.

[23] R. Kohavi *et al.*, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, vol. 14, 1995, pp. 1137–1145.

[24] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer New York, 2009.

[25] S. Raschka, "Naive Bayes and Text Classification I - Introduction and Theory," 2014.

[26] C. M. Bishop, *Pattern Recognition and Machine Learning.* Springer-Verlag New York, Inc., 2006.

[27] A. Mayr, H. Binder, O. Gefeller, M. Schmid *et al.*, "The evolution of boosting algorithms," *Methods of information in medicine*, vol. 53, no. 6, pp. 419–427, 2014.