Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell' Informazione

Corso di Laurea in

Ingegneria Informatica e dell'Automazione

Rilevanza delle componenti del nome di dominio in un sistema di malware detection basato sull'analisi del traffico DNS

Relevance of domain name components in a malware detection system based on DNS traffic analysis

Relatore:

Prof: Alessandro Cucchiarelli

Candidato:

Manuel Manelli

Correlatore:

Prof: Christian Morbidoni

Anno Accademico 2019-2020

Abstract

Da circa vent'anni a questa parte, l'umanità ha imparato a conoscere e ad utilizzare un nuovo e potente strumento che consente di veicolare le informazioni in maniera quasi istantanea.
Come tutti gli strumenti, internet può essere utilizzato per scopi fraudolenti e criminali. I cosidetti criminali informatici utilizzano degli algoritmi chiamati **DGA** o *Domain Generation Algorithm* per occultare i malware di cui si servono per prendere possesso di informazioni riservate da rivendere e/o sistemi remoti da sfruttare.
L'obbiettivo di questo lavoro di tesi sarà quello di riconoscere i nomi di dominio che questi malware utilizzano per connettersi con i criminali che li gestiscono tramite delle tecniche di *Machine Learning*.
Per fare questo, si è partiti da un dataset in cui è riportata una quantità notevole di nomi di dominio generati da DGA noti. Prima di poterli utilizzare, sono state messe in atto una serie di operazioni preliminari volte a sanificare l'insieme dei dati in modo da escludere eventuali errori. In seguito alla sanificazione, i domini sono stati caratterizzati tramite delle feature lessicali ottenendo per ogni nome di dominio un vettore di feature che lo caratterizza.

Grazie a questi dati è stato possibile addestrare un classificatore nell'ottica di organizzare una serie di esperimenti volti a valutare le performance di quest'ultimo nel riconoscere, oltre alla natura malevola di un nome di dominio, il tipo di DGA che lo ha generato.
Un altro aspetto valutato è quello della rilevanza delle componenti che formano il dominio analizzato. In quest'ottica è stato riscontrato che considerare il **TLD** o *Top Level Domain* come ulteriore feature dei dati utilizzati per eseguire l'addestramento, comporta un miglioramento complessivo della capacità di riconoscimento.

Indice

Capitolo 1

Introduzione

La crisi sanitaria del 2020 ha messo in evidenza, come mai prima nella storia, l'importanza di internet come strumento fondamentale delle nostre vite. Grazie alla rete abbiamo potuto continuare a lavorare da casa, seguire le lezioni a distanza, rimanere in contatto con le persone a noi più care e così via. Sostanzialmente internet ci ha permesso di continuare a vivere, anche se con qualche restrizione.
Con lo spostare il gran parte della nostra vita su internet, portiamo con noi inevitabilmente anche la maggior parte dei nostri dati riservati, che per definizione tali dovrebbero rimanere. Purtroppo questi dati hanno un valore per i criminali informatici, i quali impiegano da sempre una vasta gamma di risorse e di strumenti diversi per riuscire ad impossessarsene.

Attualmente i malware più diffusi sono quelli in grado di costruire una *botnet*, ovvero una rete di computer compromessi, controllati da un *bootmaster*. Le botnet vengono utilizzate per compiere una vasta gamma di operazioni illegali tra cui attacchi mirati a causare disservizi o per rubare dati personali.

Questo lavoro di tesi consiste nel trattare il problema delle botnet, descrivendone i diversi tipi e il loro funzionamento, in modo da definire dettagliatamente il problema e successivamente fornire una possibile tecnica per identificare la loro presenza attraverso l'analisi del traffico DNS.

L'analisi del traffico DNS consente di identificare la presenza del malware nella macchina osservata, in quanto questo prima o poi tenterà di connettersi con il proprio server di *Command & Control* (*C&C*) per ricevere gli aggiornamenti sulle operazioni da portare a termine. L'obbiettivo è quindi quello di interrompere questa procedura per rendere inoffensivo il malware e, per poterlo fare, è necessario essere in grado di riconoscere quando una richiesta di questo tipo avviene.

Per rendere il più difficile possibile il riconoscimento, i moderni malware includono degli algoritmi di DGA o *Domain Generation Algorithm* per generare dei nomi di dominio pseudocasuali in modo da occultare le richieste di connessione ai server di *C&C*.

La soluzione proposta in questo lavoro di tesi utilizza delle tecniche di *Machine Learning*

per determinare, con buona approssimazione, se una richiesta DNS contiene un nome di dominio associato ad un algoritmo DGA. Per fare questo verrà impiegato un ampio dataset, contenente sia vari nomi di dominio appartenenti a diverse famiglie di algoritmi DGA, sia nomi di dominio di utilizzo comune, cioè non associati ad algoritmi di generazione automatica. Il dataset verrà utilizzato come base per addestrare dei classificatori a riconoscere i nomi di dominio generati automaticamente da quelli ordinari ed eventualmente a riconoscere anche la famiglia DGA di appartenenza. Per arrivare a condurre questi esperimenti e a valutarne i risultati, andranno svolte una serie di operazioni sulla base di dati, in modo da caratterizzare i nomi di dominio in essa contenuti con una serie di feature lessicali adatte a costruire i modelli di classificazione da parte del classificatore.

Infine, parte degli esperimenti sarà dedicata a valutare le prestazioni del classificatore nel caso in cui, nella fase di addestramento, vengano incluse tutte le componenti che fanno parte del nome di dominio. Nello specifico, sarà valutata se la presenza del TLD produce un miglioramento significativo nella capacità di classificazione.

Capitolo 2

Contesto applicativo

Nel mondo in cui oggi viviamo, accedere ai servizi di internet è ormai all'ordine del giorno, quasi un'esigenza sia lavorativa che comunicativa. In accordo, il numero di dispositivi connessi negli ultimi anni ha visto un aumento esponenziale, in breve buona parte della nostra vita si è trasferita in rete. L'elevato numero di dispositivi connessi in aggiunta al numero di utenti poco esperti che li utilizzano ha consentito ai criminali informatici di trovare terreno fertile. Questi ultimi, utilizzano dei programmi malevoli per infettare normali dispositivi e compiere diverse attività illecite.

2.1 I malware

Si definisce malware un qualsiasi programma che disturba le operazioni svolte su un computer da un utente. I malware non necessariamente arrecano danni, sono intesi anche come dei programmi in grado di rubare informazioni di nascosto. Esistono diverse categorie di malware:

- **Virus**: Programma che infetta dei file per far copia di se stesso.
- **Warm**: Malware in grado di replicarsi.
- **Trojan**: Malware nascosto in un programma apparentemente innocuo.
- **Ransomware**: Limita l'accesso al dispositivo infettato chiedendo un riscatto da pagare (tipicamente in Bitcoin).
- **Spyware**: Raccoglie informazioni sull'attività online di un individuo senza il suo consenso.
- **Adware**: Software freeware che presenta al suo interno pubblicità esposte di proposito all'utente.

2.2 Botnet

Il termine *Botnet* indica un insieme di n dispositivi, appartenenti ad ignari utenti, infettati da un particolare malware, connessi in rete e controllati da un *Botmaster* attraverso un server di *Command & Control*. I dispositivi che compongono la Botnet vengono definiti dispositivi *zombie* o *bot*.
Il concetto di botnet può trovare sia applicazioni lecite, come implementare reti di calcolo distribuito per aiutare la ricerca, sia applicazioni altamente illecite quali realizzare attacchi *DDoS*, *Spam*, *Phishing*, diffondere *Spyware* oppure compiere furti di dati sensibili. La potenza di una botnet risiede nel numero di host che la compongono: ogni nuovo dispositivo infettato e arruolato rafforza l'efficacia dell'intera rete, ogni dispositivo perso la indebolisce. Per questo motivo, oltre che a cercare di controllare il maggior numero di dispositivi possibili, esse sono realizzate secondo architetture ben progettate che consentano la comunicazione sicura degli host con il server *C&C* rendendo il più difficile possibile rintracciare ed ostacolare i comandi impartiti da quest'ultimo.

2.2.1 Struttura e comunicazione

Come accennato nella sezione precedente, una botnet è composta da un insieme di bot che eseguono i comandi impartiti dal botmaster attraverso il server *C&C*. A seconda di una serie di caratteristiche da tenere in considerazione, il botmaster decide di implementare la propria botnet secondo vari tipi di rete.

Modello Client-Server

Implementando il modello client server, la botnet è organizzata secondo la **Topologia a Stella** come riportato in figura 2.1, in cui ogni bot dipende direttamente dal server *C&C*. Questa è la topologia più efficiente dal punto di vista prestazionale ma al contempo anche la più vulnerabile, in quanto *single point of failure*, ovvero basta oscurare il server *C&C* per compromettere l'intera rete.

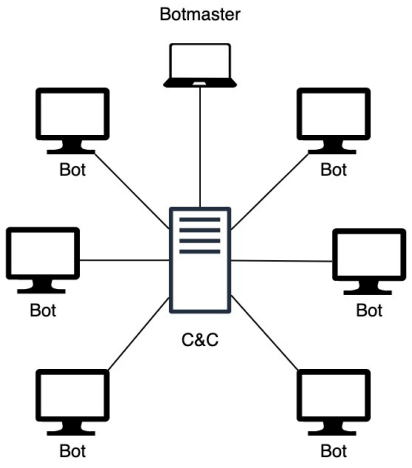

Figura 2.1: *Topologia a stella*

Per questo motivo, nel corso del tempo ne sono state sviluppate delle varianti con l'obbiettivo di aumentare la robustezza dell' infrastruttura.
Una di queste è la topologia a **Stella Estesa** (figura 2.2). Essa differisce rispetto alla stella standard per la presenza di server *C&C* ridondati all'interno della rete stessa con l'obbiettivo di aumentare la scalabilità di quest'ultima migliorandone la gestione del traffico che vi transita attraverso.

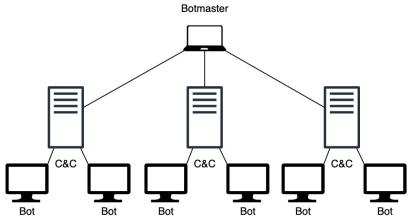

Figura 2.2: *Topologia a stella estesa*

Un'ulteriore evoluzione è rappresentata dalla **Topologia Gerarchica** (figura 2.3). Questa soluzione è caratterizzata dal fatto di impiegare alcuni bot della rete come *proxy*, che presentano caratteristiche analoghe a quelle del server *C&C*. Il principale vantaggio di questa soluzione è che nasconde la struttura della rete ai bot che la compongono rendedola estremamente sicura. Lo svantaggio principale è che i bot usati come proxy, dovendo assolvere il compito di fare da tramite tra server principale e altri bot, comportano un incremento delle latenze compromettendo di fatto il potere degli attacchi real-time.

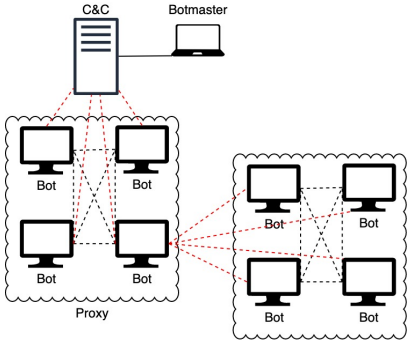

Figura 2.3: *Topologia gerarchica*

Ad ogni modo, il tallone d'Achille dell'architettura centralizzata rimane comunque la forte dipendenza da una singola entità: chi la controlla detiene il possesso dell'intera rete. Per questo motivo essa è stata archiviata in favore di un'architettura decentralizzata o **Peer to peer** (P2P).

Modello Peer to peer

Il modello **Peer to Peer**, a differenza del modello centralizzato, è implementato in maniera tale che ogni nodo sia a conoscenza esclusivamente dei nodi che gli sono accanto, ignorando completamente tutti gli altri. Esso prevede che i singoli nodi che compongono la rete siano al tempo stesso sia *client* che *server* consentendo ad un comando di entrare da qualsiasi punto della rete e propagarsi secondo rotte totalmente casuali. Inoltre se un nodo risulta compromesso, non intacca il funzionamento generale della rete che continuerà ad espandersi arruolando nuovi bot. L'arruolamento avviene interrogando sequenzialmente una serie di indirizzi *IP* generati in maniera pseudocasuale, nel caso in cui la richiesta vada a buon fine, la macchina interrogata risponde con la sua versione del software e una lista di altri bot conosciuti. Se uno dei due bot entrati in contatto possiede una versione inferiore allora esso viene aggiornato. In questo modo ogni bot accresce la propria lista di macchine conosciute incrementando le dimensioni della botnet stessa.
Il principale svantaggio di quest'architettura è dato dalla forte complessità che essa richiede, in quanto risulta necessario implementare protocolli di comunicazione customizzati.

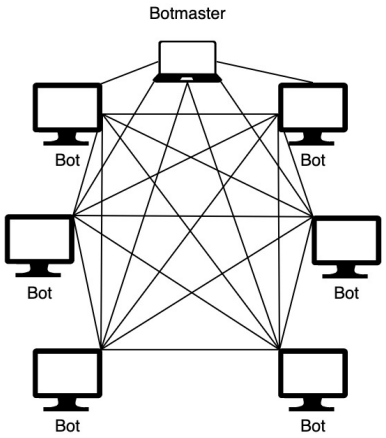

Figura 2.4: *Topologia peer to peer*

2.2.2 Fasi di una botnet

Le fasi che una botnet attraversa per propagarsi nella rete sono sostanzialmente le quattro riportate in seguito.

Exploitation

È la prima fase attraversata da una botnet. Essa è fondamentale per ottenere il controllo del dispositivo. In genere avviene attraverso una mail anonima contenente un allegato che ha il compito di infettare la vittima. In genere non è il malware vero e proprio ma è un piccolo tool che, una volta mandato in esecuzione, esegue una connessione, attraverso la risoluzione di un dominio DNS, ad un server remoto e da esso esegue il download del malware reale. Questa tecnica è la più utilizzata dalle botnet.

Rallying

La fase di **Rallying** è essenziale ma anche la più delicata, essa consente al bot di mettersi in contatto con il server *C&C* e deve essere implementata in maniera tale da inibire ad eventuali utenti la possibilità di risalire al server centrale. Vi sono diversi approcci per implementare questa fase: in seguito analizzeremo quelli principali osservandone pregi e difetti.
Una prima possibilità è quella di inserire direttamente l'indirizzo *IP* del server centrale all'interno del codice binario del malware. Questa tecnica è la più semplice dal punto di vista realizzativo, più efficiente da quello prestazionale, dato che la comunicazione è diretta, ma assolutamente inadeguata dal punto di vista della resilienza, dato che qualunque utente intenzionato ad attaccare la botnet, decompilando il binario del malware tramite un processo di *reverse-engineering*, riesce a risalire all'indirizzo *IP* del server centrale e, oscurando quello, mette fuori uso l'intera botnet.

Un'altra possibilità è quella di utilizzare il protocollo *DNS* per risolvere un nome di dominio, o una lista di domini, contenuti all'interno del binario del malware. Ottenuto l'indirizzo del server *C&C* tramite la *query* DNS, è possibile puntare ad esso in maniera totalmente dinamica. Questo permette al botmaster di spostare il server centrale ovunque e in qualunque momento aggiornando il record DNS di volta in volta. Tuttavia, anche in questo metodo è sufficiente una blacklist DNS per impedire la risoluzione del nome di dominio in questione e neutralizzare l'intera rete.

Per ovviare a questo problema, quasi tutte le botnet includono algoritmi di **DGA** ovvero algoritmi di generazione automatica dei nomi di dominio in maniera pseudocasuale. Questo argomento è trattato in maniera approfondita nella sezione 2.2.3. La tecnica prevede che l'algoritmo venga eseguito sia sul bot che sul server di *C&C* partendo dallo stesso *seed* (elemento per inizializzare il generatore pseudocasuale) per entrambi, come *seed* può essere presa l'ora corrente, il primo trend-topic di Twitter in una determinata regione, la data corrente ecc.. Al termine della computazione, l'algoritmo ritorna una lista di nomi di dominio pseudocasuali, identica su entrambi i fronti (essendo stato preso lo stesso *seed*). Partendo da questa lista il server seleziona, in maniera casuale, uno o più nomi di dominio e li registra presso un servizio di DNS associandoli al proprio indirizzo *IP*. Il bot invece procede ad eseguire una *query* DNS per ogni nome di dominio presente nella lista fintanto che la *query* non va a buon fine. Quando questo avviene, il nome di dominio selezionato dal bot risulta lo stesso registrato dal server centrale, a questo punto il client può utilizzare l'indirizzo *IP* ottenuto per eseguire la procedura di *rendez-vous* con il server *C&C*.

Execution

La fase di **Execution** è la fase in cui i bot eseguono i comandi ricevuti dal botmaster per compiere attacchi ed attività dannose.

Update

Durante la fase di **Update**, i bot ricevono gli aggiornamenti dal bot master contenenti nuove feature, bug fix o nuove istruzioni su come contattare il server centrale per eludere le tecniche di rilevamento.

2.2.3 Domain Generation Algorithms

Le più sofisticate tecniche di *rallying* basano il loro funzionamento sul protocollo DNS.

Domain Name System DNS

Internet è basato sul protocollo **TCP/IP**, ciò comporta che ogni risorsa disponibile è individuata da un indirizzo *IP*, associato ad una macchina fisica e un numero di Porta che identifica univocamente il processo in esecuzione su quella macchina che eroga la risorsa in questione. L'unione di indirizzo *IP* e numero di Porta (**IP:Porta**) costituisce una *Socket TCP*. Ad esempio, se ho intenzione di richiedere la pagina web del sito di *Repubblica* dovrò eseguire una richiesta ad un processo in esecuzione sulla macchina fisica individuata dall'indirizzo *IP* `213.92.16.101` specificando il tipo di servizio che voglio richiedere, trattandosi di una pagina web dovrò utilizzare il protocollo **HTTP** che è associato in maniera standard dalla **Porta TCP 80**, dunque la socket a cui inoltrare la richiesta sarà `213.92.16.101:80`. Eseguita la richiesta, la machina remota risponderà con un documento *HTML* che poi verrà elaborato dal browser mostrandomi infine l'homepage di Repubblica.
Questo procedimento richiede necessariamente la conoscenza di tutti gli indirizzi *IP* dei servizi che utilizziamo quotidianamente. Il problema principale è che la mente umana non è molto abile a ricordare tante sequenze di numeri diversi tra loro senza un senso apparente, a differenza delle sequenze di caratteri a cui riesce ad associare qualcosa. Un esempio è la rubrica telefonica, dove associamo ad ogni numero di telefono il nome di una persona che conosciamo. Il protocollo DNS ha lo stesso compito di una rubrica telefonica, ovvero quello di associare ad una serie di indirizzi *IP* una sequenza di caratteri consentendoci di accedere a qualsiasi servizio senza ricordare necessariamente l'indirizzo *IP* della macchina che lo eroga. Ritornando all'esempio precedente, è possibile accedere al sito di repubblica digitando semplicemente *repubblica.it* chiamato **nome di dominio**. Un nome di dominio è composto da una serie di stringhe, di lunghezza massima pari a 63 caratteri, separate da punti e disposte in ordine gerarchico da destra a sinistra. Ad esempio *repubblica.it* è composto dalle stringhe *repubblica* e *it*. In totale il nome di dominio non può superare i 255 caratteri totali. La prima stringa a partire da destra, *it*, è denominata *TLD* (*Top Level Domain*) o dominio di primo livello, segue poi *repubblica* che rappresenta il dominio di secondo livello o *2LD* ed eventualmente anche domini di terzo, quarto livello e così via. La struttura gerarchica dei nomi di dominio rispecchia l'organizzazione gerarchica del servizio DNS. In sostanza quando si esegue una *query* DNS, il server DNS locale (tipicamente quello del proprio ISP) analizza il nome di dominio estraendone il TLD (*it*) ed esegue a sua volta una richiesta al root DNS server per risolvere il TLD ed ottenere l'indirizzo del DNS server di quel TLD a cui richiederà di risolvere la seconda stringa (*repubblica*) e così via. La procedura si ripete per ogni stringa che compone il dominio ottenendo di volta in volta l'indirizzo dell'authoritive DNS a cui effetuare la richiesta successiva fino a raggiungere l'authoritive per il nome desiderato. In genere non è necessario eseguire tutta la catena delle interrogazioni nella sua interezza in quanto i server mettono in atto un meccanismo di caching memorizzando le risposte ottenute per un certo lasso di tempo oppure fino a quando la cache non viene svuotata.

DGA

Gli esperti di sicurezza informatica che tentano di attaccare e mettere fuori uso le botnet, hanno come obbiettivo quello di impedire la connessione tra bot e server *C&C*. In seguito a ciò i botmaster hanno adottato l'utilizzo di DGA per generare un grande numero di domini casuali tramite l'ausilio di un *seed* comune tra bot e server.

In base alla natura del *seed* possiamo avere DGA **Tempo Dipendenti** oppure **Deterministici**. Nel primo caso, a seconda che il *seed* vari o meno nel tempo, possiamo avere rispettivamente algoritmi dipendenti o indipendenti. Nel secondo, parliamo di DGA **Deterministici** se siamo a conoscenza dei parametri richiesti per l'esecuzione, altrimenti parliamo di DGA **Non deterministici**.

In sintesi esistono quindi quattro diversi tipi di DGA in funzione del *seed*:

- **TID-DGA** *Time Independent Deterministic* utilizzano un seme statico per l'algoritmo di generazione, ottenendo quindi sempre la stessa lista di domini.
- **TDD-DGA** *Time Dependent Deterministic* stesso algoritmo del precedente ma utilizzano un seme di volta in volta diverso per generare la lista di domini.
- **TDN-DGA** *Time Dependent Not deterministic*
- **TIN-DGA** *Time Independent Not deterministic*

e quattro diversi schemi di generazione:

- **Hash-based DGA** algoritmo che genera domini partendo dalla rappresentazione esadecimale di un hash.
- **Permutation-based DGA** Produce i domini permutando un nome di dominio iniziale.
- **Wordlist-based DGA** unisce delle liste di parole codificate nel binario del malware per produrre nomi di dominio apparentemente meno casuali.
- **Arithmetic-based DGA:** Utilizza valori numerici calcolati secondo la codifica ASCII come nome di dominio.

2.3 Machine Learning

Il **Machine Learning** è una branca dell'intelligenza artificiale che si occupa di studiare lo sviluppo di algoritmi che perfezionano i risultati forniti in funzione dell'esperienza che acquisiscono. Questi algoritmi costruiscono un modello matematico basato su di un insieme di dati iniziale chiamato *training dataset* che funge da addestramento in modo da riuscire a prendere decisioni su un insieme di dati diverso e sconosciuto con un intervento umano ridotto al minimo. Negli ultimi anni il Machine Learning è diventato sempre più popolare grazie anche alla diffusione massiva di internet che consente di recuperare facilmente grandi quantità di dati utili per addestrare gli algoritmi, oltre che alla riduzioni dei costi delle elaborazioni e spazi di archiviazioni dei dati. Un altro aspetto cruciale per la diffusione del Machine Learning è dato anche dal *Cloud Computing*, esso permette a chiunque di accedere ad una potenza di calcolo incredibilmente elevata pagando solo per il tempo che la si utilizza. In questo modo chiunque è in grado di sviluppare questo tipo di algoritmi senza necessariamente dotarsi di hardware particolarmente performante e quindi sostenere i corrispettivi costi. Le tecniche di Machine Learning più utilizzate si suddividono in base al tipo di apprendimento che viene effettuato, possiamo avere apprendimento **supervisionato**, **non supervisionato**, **semi supervisionato** e **per rinforzo**.

L'**Apprendimento supervisionato** consiste nell'addestrare l'algoritmo con un insieme di dati, denominato *Training dataset* di cui è nota la natura (dati etichettati). La procedura di addestramento serve all'algoritmo per costruire il modello, associando man mano agli input i corrispettivi output e comparando i risultati.
Dell'apprendimento supervisionato fa anche parte la classificazione, essa prevede che l'algoritmo, in base a quello che ha imparato durante la fase di addestramento, riesca ad assegnare una categoria, tra quelle note a cui appartengono i dati di training, al dato fornito in ingresso. Affinché questo avvenga è importante che, durante la fase di addestramento, venga fornito all'algoritmo un dataset in cui i dati siano organizzati per categorie.

L'**Apprendimento non supervisionato** utilizza per l'addestramento un insieme di dati di natura ignota (non etichettati), dunque l'algoritmo deve imparare da sé a riconoscere le differenze nella struttura dei dati di input.

L'**Apprendimento semi supervisionato** è una tipologia di apprendimento in cui il dataset di training è composto sia da dati etichettati che da dati non etichettati. In genere si utilizza quando non è possibile classificare l'intero dataset.

L'**Apprendimento per rinforzo** prevede una suddivisione della realtà in tre parti: chi prende le decisioni denominato *agente*, quello con cui l'agente interagisce chiamato *ambiente* e infine le *azioni* ovvero quello che l'agente può fare. L'algoritmo prevede che l'agente interagisca con l'ambiente tramite le azioni che comportano una massimizzazione della ricompensa prevista per un lasso temporale: tramite una giusta sequenza di azioni l'agente raggiungerà l'obbiettivo in maniera più rapida. È particolarmente utilizzato nella robotica.

2.3.1 Sistemi basati su features

Molti algoritmi di Machine Learning richiedono che gli oggetti con i quali operano siano descritti tramite delle **features**. Esse sono la rappresentazione numerica di particolari aspetti del fenomeno da modellare, ritenuti rilevanti e formati appositamente per generare l'input dell'algoritmo. Ad esempio nel caso di un dominio applicativo caratterizzabile attraverso testi, le features possono essere la frequenza con cui occorrono i termini testuali. Le features per uno stesso dato vengono raccolte all'interno di un vettore chiamato *features vector*, esso rappresenta il dato da dare in input all'algoritmo. Lo spazio vettoriale associato ai features vector viene denominato spazio delle funzioni.

2.3.2 Deep-learning

Il **Deep-learning** o apprendimento profondo è una tecnica di apprendimento automatico che mira ad emulare l'apprendimento della mente umana, ovvero un insieme di reti neurali artificiali organizzate in diversi strati, come riportato in figura 2.5, dove ogni strato calcola dei valori per quello successivo affinché la struttura che lega i dati nel dominio dell'applicazione venga ricostruita nella maniera più completa possibile.

2.3.3 Decision tree

L'albero decisionale o **Decision Tree** è uno degli approcci di modellazione predittiva utilizzati in statistica, data mining e machine learning. Un albero decisionale è una struttura simile a un albero binario in cui ogni nodo interno rappresenta un "test" su un attributo (ad esempio, se un lancio di moneta esce testa o croce), ogni ramo rappresenta il risultato del test e ogni nodo foglia rappresenta un'etichetta della classe (decisione presa dopo aver calcolato tutti gli attributi). I percorsi dalla radice alla foglia rappresentano le regole di classificazione. I modelli ad albero in cui la variabile target può assumere un insieme discreto di valori sono chiamati alberi di classificazione. Gli alberi decisionali in cui la variabile target può assumere valori continui (in genere numeri reali) sono chiamati alberi di regressione. Gli alberi decisionali sono tra gli algoritmi di machine learning più diffusi data la loro intelligibilità e semplicità.

2.3.4 Support Vector Machine

L'algoritmo **Support Vector Machine** (SVM) è un popolare strumento di apprendimento automatico che offre soluzioni per problemi di classificazione e regressione. Esso presenta uno dei metodi di previsione più robusti, basato sul framework di apprendimento statistico. Dato un insieme di esempi di addestramento, ognuno segnato come appartenente all'una o all'altra di due categorie, un algoritmo di addestramento SVM costruisce un modello che assegna nuovi esempi a una categoria o all'altra, rendendolo un classificatore lineare binario non probabilistico. Un modello SVM è una rappresentazione degli esempi come punti nello spazio, mappati in modo tale che gli esempi delle categorie separate siano divisi da uno spazio vuoto il più ampio possibile. Nuovi esempi vengono quindi mappati nello stesso spazio e previsti per appartenere a una categoria in base al lato dello spazio su cui cadono.
Oltre a eseguire la classificazione lineare, le SVM possono eseguire in modo efficiente una

classificazione non lineare usando quello che viene chiamato *kernel trick*, mappando implicitamente i loro input in spazi di caratteristiche ad alta dimensione.

2.3.5 Multilayer perceptron

Un **Percettrone Multistrato** (MLP) è una classe di rete neurale artificiale feedforward. Un MLP è costituito da almeno tre livelli di nodi: un livello di input, un livello nascosto e un livello di output. Ad eccezione dei nodi di input, ogni nodo è un neurone che utilizza una funzione di attivazione non lineare. MLP utilizza una tecnica di apprendimento supervisionato chiamata backpropagation per la formazione. I suoi strati multipli e l'attivazione non lineare distinguono MLP da un percettrone lineare. Può distinguere dati che non sono separabili linearmente.

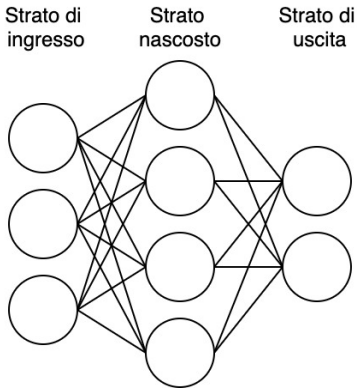

Figura 2.5: *Esempio di rete neurale*

Capitolo 3

Obbiettivi del progetto

Nella sua totalità il progetto si pone gli obbiettivi di identificare, catalogare ed eventualmente intervenire per neutralizzare le minacce illustrate nella sezione precedente.

La fase di identificazione prevede che i tool sviluppati siano in grado di apprendere le differenze tra nomi di dominio *legit* (ovvero legittimi) e nomi di dominio *illegit* (quelli generati da un DGA) e quindi riconoscere di conseguenza un nome di dominio mai visto in una delle due sopra citate categorie.
Per quanto riguarda la catalogazione invece prevede che i tool, oltre che a identificare un nome di dominio come legit o illegit, si occupino anche di riconoscere la famiglia DGA di appartenenza, in caso esso sia generato.
Infine la fase di intervento prevede l'identificazione del processo che ha eseguito quella richiesta DNS, la terminazione forzata dello stesso e il conseguente inserimento del nome di dominio nella blacklist DNS, oppure procedere alla sua risoluzione al posto del malware, ottenere quindi l'indirizzo del server di *C&C* per procedere ad oscurarlo.

L'obbiettivo principale del progetto è quello di eseguire quattro esperimenti, operando una classificazione *binaria* nei primi due e una classificazione *multiclasse* negli ultimi due. In entrambi i tipi di classificazione i due esperimenti differiscono solamente per la presenza nei dati in ingresso di un campo relativo al TLD, per verificare nei risultati se quest'ultimo contribuisce a migliorare o a peggiorare le performance del classificatore.

Le tecniche di malware detection possono essere analizzate sotto diverse prospettive.
Una prospettiva di analisi consiste nel monitorare il traffico di rete tramite tecniche di rilevamento che possono essere attive o passive. Monitorare il traffico in maniera attiva denota l'iniezione nella rete da monitorare, da parte del tool, di particolari pacchetti DNS con lo scopo di stimolare una risposta. Il principale svantaggio di questa tecnica è quello di generare traffico inutile sulla rete.
Il monitoraggio passivo consiste nel monitorare il traffico DNS sulla rete o su una singola macchina. La difficoltà, in questo caso, è che il traffico generato dalla botnet risulta essere misto a quello generato dalle applicazioni legittime, risultando quindi difficile da distinguere. Proprio per questo motivo, l'efficacia di questa tecnica dipende fortemente dal tipo di informazione monitorata e da quanto quest'informazione ci consente di distinguere le richieste

legittime da quelle illegittime.
I lavori focalizzati su questo, sono incentrati sul monitorare una serie di informazioni come il traffico generato da e verso i server DNS, nello specifico analizzando i nomi di dominio. Nel fare questo occorre considerare che i *C&C* server utilizzano delle tecniche di camuffamento come i DGA. Queste tecniche consentono al traffico generato dalle botnet di essere molto simile a quello generato dalle normali applicazioni rendendo il processo di rilevamento estremamente difficile.
In base a quanto detto, il progetto si pone l'obbiettivo di sviluppare tecniche di rilevamento basate sulla classificazione dei nomi di dominio in due categorie: benevoli (*legit*) e malevoli (*illegit).*

3.1 Definizione degli step

Introdotti gli strumenti matematici da utilizzare e i dataset dei nomi di dominio legit e illegit, procediamo con il suddividere l'intero progetto in due parti. La prima parte sarà dedicata alla sanificazione del dataset di partenza e alla composizione di un sottoinsieme di quest'ultimo, che verrà poi utilizzato nella seconda parte del progetto.
La seconda sarà dedicata alla messa in atto della cosiddetta **Catena di attivazione**, ovvero una serie di script che andranno attivati in sequenza per trasformare i dati grezzi nel formato richiesto dal classificatore.

3.2 Prima parte

Come accennato in precedenza, la prima parte del progetto sarà dedicata alla sanificazione del dataset di partenza e alla composizione di un sottoinsieme di quest'ultimo da utilizzare per la fase di training del classificatore. Essa si articolerà in tre fasi, descritte in seguito.

Fase 1: Sanificazione

La fase di **sanificazione** è resa necessaria dal fatto che il dataset di partenza potrebbe contenere nomi di dominio non validi dal punto di vista formale, oppure che presentano un TLD non classificato. Bisognerà quindi analizzare l'intero dataset per individuarli e rimuoverli.

Fase 2: Indicizzazione dei TLD

Una volta ricavato il dataset sanificato, bisognerà trovare un modo semplice per rappresentare i TLD di ogni dominio. Questo passo è di fondamentale importanza, in quanto gli esperimenti che andremo a condurre avranno lo scopo di mostrare se l'aggiunta del riferimento al TLD comporta un miglioramento o un peggioramento della precisione con cui il classificatore sarà in grado di riconoscere il tipo e la famiglia di appartenenza dei nomi di dominio.

Fase 3: Sanificazione di un sottoinsieme del dataset

Consideriamo D il dataset di partenza e $d \subset D$ un sottoinsieme di D. In quanto questo progetto non parte da zero ma discende da un lavoro precedente, sarebbe interessante poter

confrontare i risultati prodotti da entrambi partendo dallo stesso sottoinsieme del dataset d usato per addestrare il classificatore in ambo i progetti. Per ottemperare a ciò, bisognerà ripetere quanto fatto nella fase di sanificazione anche per il sottoinsieme d utilizzato nel lavoro precedente. Nel caso in cui emergano dei nomi di dominio da scartare, si procederà a rimpiazzarli con altri della stessa famiglia, prelevati dal dataset di partenza D in maniera casuale in modo da mantenere invariate le proporzioni.

3.3 Seconda parte

Come anticipato precedentemente, la catena di attivazione rappresenta la serie di script da lanciare per tramutare l'insieme dei nomi di dominio nei feature vectors che verranno poi forniti al classificatore. Il seguente grafico mostra una rappresentazione visuale della catena, in esso gli script sono rappresentati dai riquadri, mentre il flusso dei dati è rappresentato dalle frecce tratteggiate.
I dati in ingresso indicati nel grafico 3.1 sono suddivisi in due dataset denominati **setA** e **setB**. Ognuno di essi è composto da una serie di nomi di dominio suddivisi in file per famiglia di appartenenza (si veda il paragrafo 5.1 per maggiore chiarezza).
Questi due set, frutto di precedenti lavori di tesi e descritti nel dettaglio nel paragrafo 5.1, costituiscono i dati di partenza di questo lavoro.

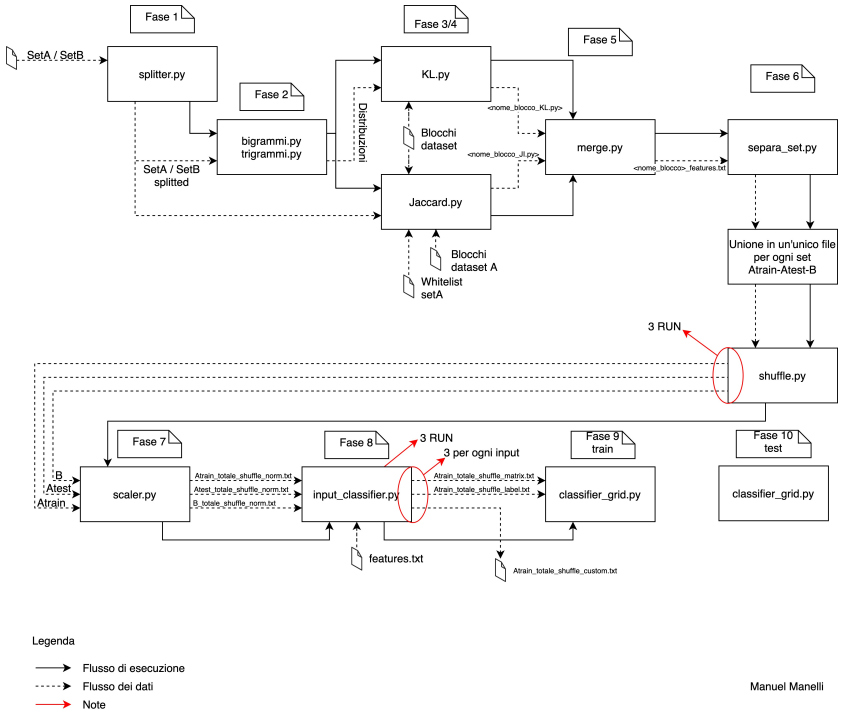

Figura 3.1: *Catena di attivazione*

Fase 1: Splitting

In questa fase, i nomi di dominio, appartenenti ai due dataset introdotti in precedenza, dovranno essere separati dal loro TLD e memorizzati in due file diversi, prestando molta attenzione all'ordine in cui questi verranno scritti nei file, in modo da preservare la relazione tra nomi e TLD. Questi ultimi dovranno essere memorizzati associandoli ad una chiave numerica.

Fase 2: Calcolo dei bigrammi e dei trigrammi

Per rappresentare i nomi di dominio utilizziamo i *bigrammi* e i *trigrammi*. Essi indicano tutte le possibili combinazioni di due/tre caratteri contigui presenti in una stringa data. Detto ciò, per ogni nome di dominio privato del suo TLD bisognerà calcolare le distribuzioni di tutti i possibili *bigrammi* e *trigrammi* partendo dal dizionario di caratteri descritto dalla **RegExr** `/([a-z0-9-_.])/`.

Fase 3/4: Calcolo KL e JI

L'obbiettivo di questa fase sarà quello di calcolare, partendo dal dataset, i valori della *distanza di Jaccard* e della *divergenza di Kullback-Leibler*, per quest'ultima partendo anche dalle distribuzioni calcolate nella fase precedente.

Fase 5: Merge

La fase di **merge** consisterà nell'unione, riga per riga, in un unico vettore per ogni nome di dominio, dei risultati ottenuti dal calcolo della *distanza di Jaccard* e della *divergenza di Kullback-Leibler* più ulteriori features quali *nome di dominio* (DN), *TLD code* e *black/white flag*.

Fase 6: Preparazione dei dati

Questa fase prevede una serie di operazioni preliminari sui file prodotti dalla precedente, ovvero consente di applicare un filtro alle features, unire i domini benevoli e malevoli in un unico file e mescolarne le righe in modo che il clasificatore riceva in ingresso una distribuzione uniforme dei dati. Il prodotto sarà quindi un unico file di feature vector per ogni set in cui sono contenuti elementi di ciascuna categoria, benevoli e malevoli.

Fase 7: Normalizzazione

In questa fase si opererà una normalizzazione tra 0 e 1 dei valori contenuti nei vettori relativi alla distanza di Jaccard e alla divergenza di Kullback-Leibler.

Fase 8: Preparazione del file di input

Partendo dal file dei feature vector, dovremo andare a realizzare 4 diversi file da dare in pasto al classificatore per i quattro diversi esperimenti che andremo a svolgere di seguito riportati.

- Esperimento A: classificazione binaria dove non è presa in considerazione la famiglia DGA di appartenenza del nome di dominio illegit
- Esperimento B: stesso caso precedente ma con l'inclusione del codice relativo al TLD di ogni dominio.
- Esperimento C: Classificazione multiclasse dove è presa in considerazione la famiglia DGA di appartenenza del nome di dominio.
- Esperimento D: stesso caso precedente ma con l'inclusione del codice relativo al TLD di ogni dominio.

Fase 9/10:

Ultima fase in cui verranno lanciati i classificatori. I classificatori operano in due fasi distinte, la prima, chiamata addestramento o *train*, è quella in cui viene sottoposto al classificatore un insieme di dati etichettati che serviranno per costruire il modello di classificazione. La seconda è chiamata *test*, in questa fase vengono utilizzati i modelli generati nella fase di train per classificare un insieme di dati non etichettati e diversi dai precedenti.

Capitolo 4

Strumenti utilizzati

Per conseguire gli obbiettivi illustrati nel precedente capitolo si è fatto uso di una vasta quantità di strumenti. In questo capitolo saranno introdotti e descritti i più rilevanti.

4.1 Linguaggio Python

Il linguaggio di programmazione che è stato impiegato nel progetto è il **Python**[2]. Esso è un linguaggio di programmazione di nuova concezione, orientato agli oggetti, adatto anche a sviluppare applicazioni distribuite, scripting e computazione numerica.
I suoi costrutti linguistici e l'approccio orientato agli oggetti mirano ad aiutare i programmatori a scrivere codice chiaro e logicamente corretto per progetti su piccola e grande scala. Python è tipizzato dinamicamente e sottoposto a garbage collection. Supporta più paradigmi di programmazione, inclusa la programmazione strutturata (in particolare procedurale), orientata agli oggetti e funzionale. Python è spesso descritto come un linguaggio a 360 gradi, a causa della sua completa standard library.
Python è un linguaggio interpretato (figura 4.1), ovvero il codice sorgente non viene compilato prima di essere eseguito, ma il sorgente stesso è il programma da eseguire, a patto di avere disponibile sulla propria macchina la versione corretta dell'interprete Python.

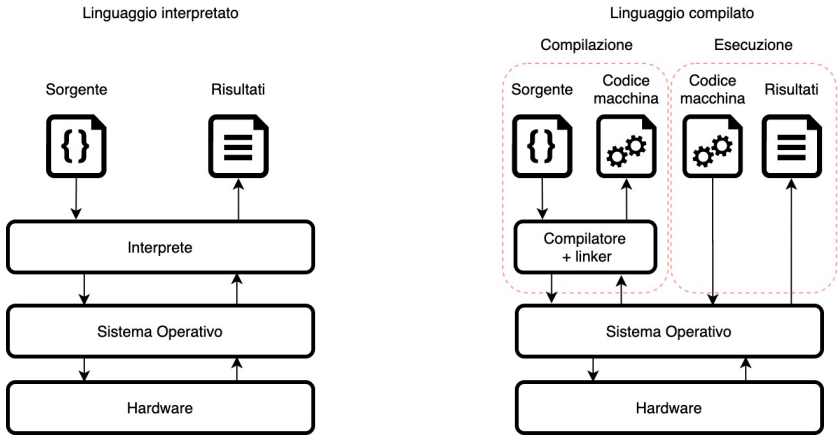

Figura 4.1: *Schema concettuale di un linguaggio interpretato e di uno compilato*

È possibile anche compilare un programma Python grazie all'ausilio del modulo `py_compile`. Esso mette a disposizione due funzioni, una permette di generare un file *byte code* da un file sorgente e l'altra viene utilizzata quando il sorgente del modulo viene richiamato come script. Python consente anche di aprire un terminale dove è possibile impartire comandi in tempo reale all'interprete. Un valore aggiunto del Python è dato dalla vasta gamma di librerie disponibili, del tutto open source messe a disposizione dalla community. Di seguito è riportata una descrizione dettagliata delle librerie utilizzate in questo progetto.

4.2 Librerie utilizzate

In Python le librerie sono note come *moduli*. Per installarle ed utilizzarle si utilizza il tool a riga di comando `pip`, basta lanciarlo specificando il nome del modulo che si intenede utilizzare e `pip` provvederà ad installarlo risolvendo in automatico tutte le dipendenze che potrebbero occorrere. Lo scopo delle librerie software è consentire il riuso del codice. I principali vantaggi sono dati dalla separazione della logica di programmazione da funzioni specifiche come quelle di calcolo matematico, gestione delle strutture dati e così via. Un altro vantaggio è dato dal fatto che è possibile modificare le librerie in maniera separata dal programma, che continuerà ad utilizzarle senza problemi a patto che le interfacce delle funzioni che esso utilizza rimangano invariate. Nei seguenti sottoparagrafi sono riportate le librerie, che sono state impiegate per la realizzazione del progetto.

4.2.1 Sys

Il modulo Python `sys` fornisce funzioni e variabili che vengono utilizzate per manipolare diverse parti del Python Runtime Environment. Ci consente di accedere a parametri e funzioni specifici del sistema.

4.2.2 OS

Questo modulo mette a disposizione una serie di modi diversi per interagire con il sistema operativo, come ad esempio l' `I/O` di file, manipolazione dei path, la creazione di file o directory temporanee e cosi via. Esso consente quindi di astrarre il sistema operativo su cui il programma viene eseguito consentendo a quest'ultimo di interagire con sistemi diversi senza modificare le proprie istruzioni garantendo quindi l'interoperabilità del codice.

4.2.3 Time

Questo modulo fornisce varie funzioni relative al tempo. Il funzionamento è basato sul conteggio dei secondi trascorsi dall'epoca. L' epoca è il punto in cui inizia il tempo ed è dipendente dalla piattaforma. Per Unix l'epoca è il 1 gennaio 1970, 00:00:00 (UTC).

4.2.4 CSV

Il modulo `csv` implementa classi per leggere e scrivere dati tabulari in formato CSV. Consente di leggere e scrivere in tale formato senza conoscere i suoi dettagli precisi. Inoltre è possibile descrivere i formati CSV compresi da altre applicazioni o definire i propri formati CSV speciali.

4.2.5 Threading

Il modulo `_threading` fornisce primitive di basso livello per lavorare con più thread (chiamati anche processi o attività leggeri). Per la sincronizzazione vengono forniti semplici blocchi (quali mutex e semafori). Questo modulo mette a disposizione tali primitive ad un modulo di livello superiore che prende il nome di `Treading` module.

4.2.6 Collection

Questo modulo implementa delle strutture dati alternative ai tipi base, `dict`, `list`, `set` e `tuple` definiti dal linguaggio Python.

4.2.7 Pickle

Il modulo `pickle` implementa protocolli binari per serializzare e de-serializzare una struttura di oggetti Python. *Pickling* è il processo mediante il quale una gerarchia di oggetti Python viene convertita in un flusso di byte e *unpickling* è l'operazione inversa, per cui un flusso di byte (da un file binario o un oggetto simile a byte) viene riconvertito in una gerarchia di oggetti.

4.2.8 NumPy

`NumPy`[1] è il modulo fondamentale per il calcolo scientifico in Python. È una libreria che mette a disposizione un oggetto array multidimensionale, vari oggetti derivati (come array e matrici) e un vasto assortimento di routine per eseguire operazioni veloci sugli array, incluse operazioni di natura matematica, logica, manipolazione geometrica, ordinamento, selezione, `I/O`, trasformate discrete di Fourier, algebra lineare di base, operazioni statistiche di base, simulazione casuale e molto altro.
`NumPy` basa il suo funzionamento sull'oggetto `ndarray` in esso definito. Questa struttura dati incapsula degli array n-dimensionali di dati omogenei in tipo, implementando al tempo stesso i metodi con codice compilato per migliorare le prestazioni. Gli elementi dell'array possono essere indirizzati in maniera molto semplice tramite n numeri interi. Il tipo di ogni elemento appartenente all'array è specificato da un oggetto separato chiamato `data-type`. In esso sarà immagazzinato il tipo dei dati contenuti all'interno dell'`nd-array` come `integer`, `float`, ulteriori strutture dati e così via. Nel momento in cui viene eseguito l'accesso ad un elemento dell'`nd-array`, quest'ultimo ritornerà un oggetto Python di tipo `array-scalar` che conterrà a sua volta l'elemento desiderato. Uno schema è riportato in figura 4.2.

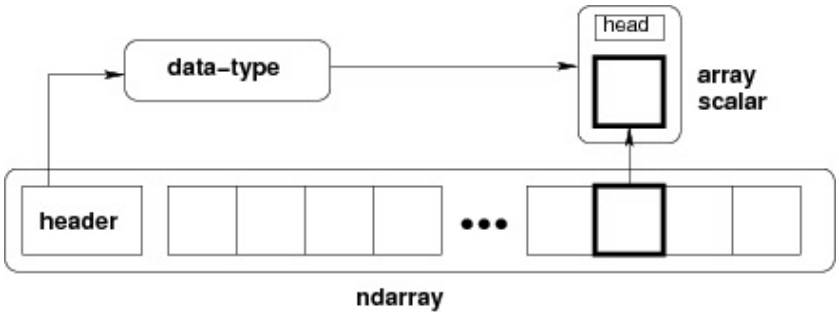

Figura 4.2: *Struttura nd-array*

Per quanto riguarda gli scalari, Python definisce solo un tipo di una particolare classe di dati (un solo tipo intero, un solo tipo a virgola mobile, e così via). Questo può essere utile nelle applicazioni ordinarie ma per il calcolo scientifico, tuttavia, è spesso necessario un maggiore controllo. In `NumPy`, ci sono 24 nuovi tipi fondamentali per descrivere diversi tipi di scalari (vedi figura 4.3).

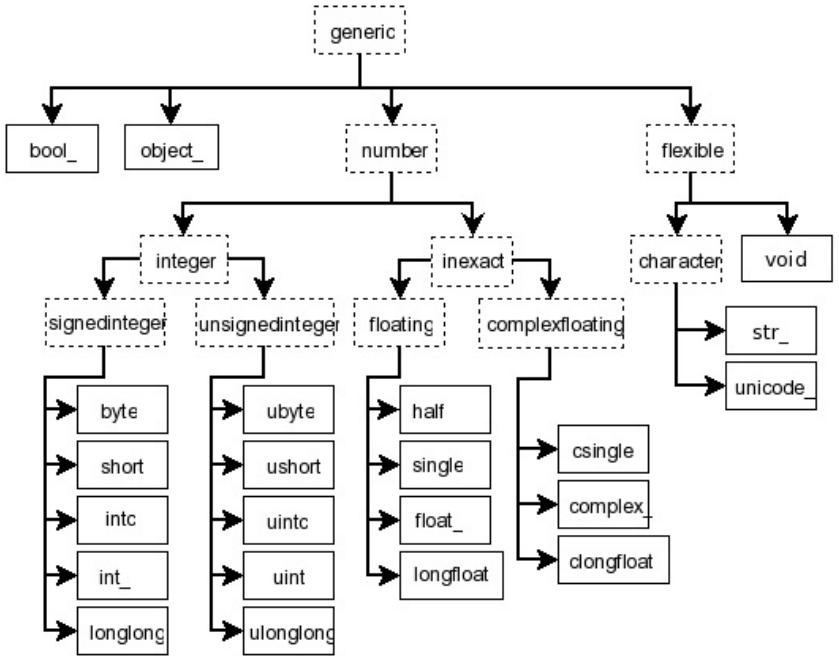

Figura 4.3: *Struttura gerarchica dei tipi scalari messi a disposizione da NumPy*

Questi descrittori di tipo si basano principalmente sui tipi disponibili nel linguaggio C in cui è scritto `CPython` , con diversi tipi aggiuntivi compatibili con i tipi di Python.
Per agevolare l'iterazione delle strutture dati, in `NumPy` è presente un oggetto iteratore `nditer`. Esso fornisce molti modi flessibili per accedere a tutti gli elementi di uno o più array in modo sistematico. `NumPy` grazie al **BLAS** (Basic Linear Algebra Subprograms) e al **LAPACK** (Linear Algebra PACKage) include inoltre una vasta gamma di funzioni di algebra lineare come il prodotto scalare, prodotto tra matrici, calcolo delle matrici inverse, calcolo di autovettori

e autovalori e così via. Queste procedure sono implementate a basso livello per permettere una maggiore efficenza.

4.2.9 SciPy

`SciPy`[4] è una raccolta di algoritmi matematici e funzioni utili basate sull'estensione `NumPy`. Aggiunge una potenza significativa alla sessione interattiva di Python fornendo all'utente comandi e classi di alto livello per la manipolazione e la visualizzazione dei dati. Con `SciPy` una sessione interattiva di Python diventa un ambiente di elaborazione dati e modellazione avanzata che compete con sistemi al livello di MATLAB, IDL, Octave, R-Lab e SciLab. `SciPy` è organizzato in sotto-pacchetti che coprono diversi domini di calcolo scientifico. In questo progetto è stato usato il pacchetto `stats` che mette a disposizione un'enorme quantità di strumenti statistici. Più precisamente per il calcolo dei valori della divergenza di Kullback-Leibler è stata utilizzata la funzione `scipy.stats.entropy()` per calcolare l'entropia di una distribuzione per determinati valori di probabilità.

4.2.10 Scikit-Learn

`Scikit-Learn`[3] è una libreria che contiene una serie di algoritmi che possono essere facilmente implementati e adattati per la classificazione e altre tecniche di machine learning. `Scikit-Learn` basa il suo funzionamento sugli strumenti messi a disposizione dalla libreria `SciPy`. Per quanto riguarda la classificazione, `Scikit-Learn` mette a disposizione differenti algoritmi di classificazione come:

- K-Nearest Neighbors
- SVM o Support Vector Machine
- Alberi decisionali/Random Forests
- Naive Bayes
- Linear Discriminant Analysis
- Logistics Regression

K-Nearest Neighbors

Per assegnare una classe a un numero k di campioni forniti in ingresso, l'algoritmo opera confrontando la distanza tra i campioni forniti e i campioni noti, suddivisi a loro volta per classi, con cui è stato addestrato. La classe che verrà assegnata ai campioni in ingresso sarà la classe del gruppo di campioni con cui risulta la distanza minore. Una rappresentazione grafica è riportata nella figura 4.4.

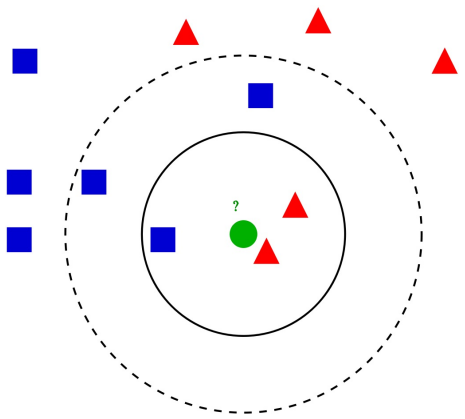

Figura 4.4: *Rappresentazione grafica dell'algoritmo K-Nearest Neighbors*

Alberi Decisionali

Gli **Alberi Decisionali**, citati nel paragrafo 2.3.3, sono un sistema con n variabili in input e m variabili in output. Le variabili in input, chiamate anche attributi, sono derivate dall'osservazione dell'ambiente. Le variabili in output, invece, identificano la decisione o azione da intraprendere. Esse presentano i seguenti vantaggi:

- I meccanismi che usano sono di semplice comprensione.
- Non richiedono un grande preprocessing dei dati.
- La relazione tra costo computazionale e numero di dati usati per l'addestramento assume un andamento logaritmico.
- In grado di gestire problemi in cui le decisioni possibili sono diverse.
- Possibilità di validare un modello tramite test di natura statistica.

`Scikit-Learn` mette a disposizione la classe `DecisionTreeClassifier` che fornisce tutti gli strumenti per eseguire la classificazione multiclasse usando gli alberi decisionali su un set di dati.

Naive Bayes

Un classificatore basato sul teorema di Bayes è chiamato **Classificatore Bayesiano**. Il principio che regola questa tipologia di classificatori si basa sul fatto che alcune istanze di un dominio applicativo appartengono ad un classe di interesse con una data probabilità sulla base di certe osservazioni. Tale probabilità è calcolata assumendo che le caratteristiche osservate possano essere tra loro dipendenti o indipendenti. In questo secondo caso il classificatore bayesiano è detto *naive* in quanto assume ingenuamente che la presenza o l'assenza di una particolare caratteristica in una data classe di interesse non è correlata alla presenza o assenza di altre caratteristiche semplificando notevolmente il calcolo. `Scikit-learn` prevede un modulo chiamato `naive_bayes` nel quale sono contenute varie implementazioni di questo algoritmo.

Analisi a discriminante lineare e a discriminante quadratico

Le analisi a **discriminante lineare** e a **discriminante quadratico** sono delle tecniche di riduzione della dimensionalità utilizzate come fase di pre-elaborazione nelle applicazioni di apprendimento automatico e classificazione. L'obiettivo principale delle tecniche di riduzione della dimensionalità è ridurre le dimensioni rimuovendo le informazioni ridondanti e dipendenti, trasformando le funzioni da uno spazio dimensionale superiore a uno spazio con dimensioni inferiori. Il grafico in figura 4.5 mostra i confini decisionali per l'analisi a discriminante lineare e l'analisi a discriminante quadratico. La riga inferiore dimostra che l'analisi a discriminante lineare può solo apprendere i confini lineari, mentre l'analisi a discriminante quadratica può apprendere i confini quadratici ed è quindi più flessibile.

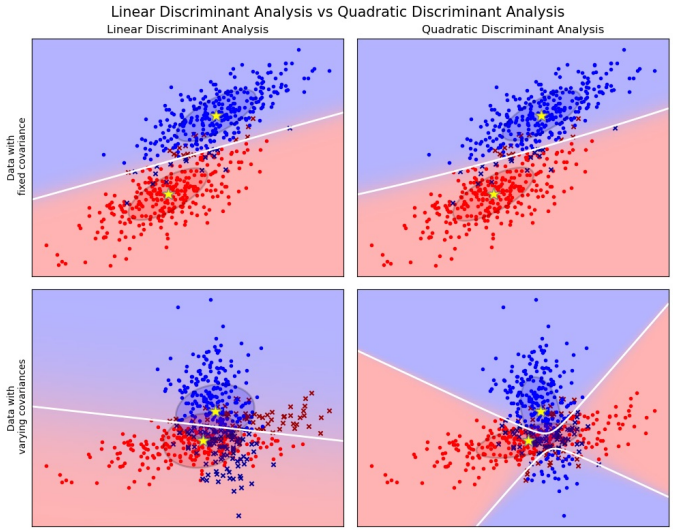

Figura 4.5: *Confini decisionali nei due casi di analisi*

4.3 Classificatore LSTM-MI

L'algoritmo di classificazione **LSTM-MI**[7] deriva dall'algoritmo LSTM di cui ne migliora alcuni aspetti. Esso impiega due tipi di classificatori, quello binario e quello multiclasse. Non fa parte della libreria `sk-learn`.
Preso un nome di dominio in ingresso, l'algoritmo prevede che venga prima stabilita la sua natura (black o white) tramite l'ausilio del classificatore binario e successivamente stabilita la classe DGA di appartenenza tramite il classificatore multiclasse.
Il motivo di cio è dato dal fatto che il classificatore multiclasse, avendo un numero maggiore di scelte da compiere, è più soggetto a commettere errori rispetto al classificatore binario, che di scelte ne ha solo due possibili. Questo consente quindi di chiamare in causa il classificatore multiclasse solo per determinare la famiglia di DGA, dopo aver stabilito che il dominio in questione risulta black tramite il classificatore binario. Ciò consente di ridurre il rapporto di sbilanciamento e di ottenere un alto tasso di rilevamento, andando a colmare l'attitudine dell'algoritmo LSTM ad etichettare i domini white come black.

Capitolo 5

Applicazione sviluppata

In questo capitolo sarà descritto in maniera dettagliata come gli obbiettivi definiti nel capitolo 3 sono stati raggiunti, illustrando sia gli algoritmi ideati che le strategie implementative attuate. Il lavoro si articolerà in due parti. La prima, più breve, affronterà la problematica del sanificare il dataset, in modo da avere una base pulita e priva di errori per fare in modo che i risultati finali prodotti dal classificatore non siano influenzati da questi ultimi. La seconda parte tratterà il preprocessing dei dati. Partendo dall'insieme grezzo dei nomi di dominio, verranno eseguite una serie di operazioni per arrivare ad ottenere dei vettori di feature che caratterizzano i nomi di dominio che saranno forniti al classificatiore per arrivare ad ottenere i risultati finali.

5.1 Il set di dati

Il lavoro parte da due set di dati[6] chiamati *A* e *B*. Il `setA` risulta a sua volta ulteriormente suddiviso in `setAtest` e in `setAtrain` di dimensioni rispettivamente 90% e 10% del numero totale dei nomi di dominio presenti nel set *A*. I due sottoinsiemi del set *A* verranno utilizzati per le fasi di training e test del classificatore ma, in questo caso, il test set conterrà famiglie di DGA già note al classificatore in quanto presenti nel training set. Quindi, il classificatore verrà addestrato sul `setAtrain` e testato sui `setAtest` e `setB`. Tutto questo è volto a voler valutare la bontà del classificatore sia su nomi di domino appartenenti a famiglie di DGA a lui già note, sia su nomi di dominio appartenenti a famiglie di DGA a lui ignote. I nomi di dominio in essi contenuti sono a loro volta suddivisi in *white* e *black*. Con il nome white ci si riferisce ai nomi di dominio leciti, con il termine black ci si riferisce all'insieme dei nomi di domini generati da varie famiglie di DGA. Come detto nel capitolo 2.2.3 gli algoritmi DGA si suddividono a loro volta in due grandi famiglie **TDD** o *Time Dipendent Deterministic* e **TID** o *Time Indipendent Deterministic*. Nel listato 5.1 è riportata la struttura del dataset.

```
originali
|    |-- set\ A
|    |    |-- blacklist
|    |    |    |-- A-test
|    |    |    |    |-- DGA-TDD
|    |    |    |    |    |-- conficker.txt
|    |    |    |    |    |-- corebot.txt
|    |    |    |    |    |-- cryptolocker.txt
|    |    |    |    |    |-- emotet.txt
|    |    |    |    |    |-- gozi.txt
|    |    |    |    |    |-- matsnu.txt
|    |    |    |    |    |-- murofet.txt
|    |    |    |    |    |-- necurs.txt
|    |    |    |    |    |-- nymaim.txt
|    |    |    |    |    |-- padcrypt.txt
|    |    |    |    |    |-- qadars.txt
|    |    |    |    |    |-- suppobox.txt
|    |    |    |    |    |__ symmi.txt
|    |    |    |    |__ DGA-TID
|    |    |    |         |-- dircrypt.txt
|    |    |    |         |-- fobber.txt
|    |    |    |         |-- kraken.txt
|    |    |    |         |-- pushdo.txt
|    |    |    |         |-- pykspa.txt
|    |    |    |         |-- ramdo.txt
|    |    |    |         |-- ramnit.txt
|    |    |    |         |-- ranbyus.txt
|    |    |    |         |-- rovnix.txt
|    |    |    |         |-- simda.txt
|    |    |    |         |-- tinba.txt
|    |    |    |         |__ vawtrak.txt
|    |    |    |__ A-train
|    |    |         |-- DGA-TDD
|    |    |         |    |-- conficker.txt
|    |    |         |    |-- corebot.txt
|    |    |         |    |-- cryptolocker.txt
|    |    |         |    |-- emotet.txt
|    |    |         |    |-- gozi.txt
|    |    |         |    |-- matsnu.txt
|    |    |         |    |-- murofet.txt
|    |    |         |    |-- necurs.txt
|    |    |         |    |-- nymaim.txt
|    |    |         |    |-- padcrypt.txt
|    |    |         |    |-- qadars.txt
|    |    |         |    |-- suppobox.txt
|    |    |         |    |__ symmi.txt
|    |    |         |__ DGA-TID
|    |    |              |-- dircrypt.txt
|    |    |              |-- fobber.txt
|    |    |              |-- kraken.txt
|    |    |              |-- pushdo.txt
|    |    |              |-- pykspa.txt
|    |    |              |-- ramdo.txt
|    |    |              |-- ramnit.txt
|    |    |              |-- ranbyus.txt
|    |    |              |-- rovnix.txt
|    |    |              |-- simda.txt
|    |    |              |-- tinba.txt
|    |    |              |__ vawtrak.txt
|    |    |__ whitelist
|    |         |-- A-test.txt
|    |         |__ A-train.txt
|    |__ set\ B
|         |-- DGA-TDD
|         |    |-- chinad.txt
|         |    |__ qakbot.txt
|         |-- DGA-TID
|         |    |__ banjori.txt
|         |__ whitelist.txt
```

Listato 5.1: Struttura del dataset

Per quanto riguarda le dimensioni, parliamo di circa 337.500 righe per i domini white (alexa) e 13.500 righe per ognuna delle 25 classi black, per un totale di 337.500 righe formando quindi un dataset perfettamente bilanciato. Si veda la tabella 5.1 per i dettagli.

NOME DGA	NUMERO DOMINI	NOME DGA	NUMERO DOMINI
whitelist(alexa)	337500	padcrypt	13500
banjori	13500	pushdo	13500
chinad	13500	pykspa	13500
conficker	13500	qadars	13500
corebot	13500	qakbot	13500
cryptolocker	13500	ramdo	13500
dircrypt	13500	ramnit	13500
emotet	13500	ranbyus	13500
fobber	13500	rovnix	13500
gozi	13500	simda	13500
kraken	13500	suppobox	13500
matsn	13500	symmi	13500
murofet	13500	tinba	13500
necurs	13500	vawtrak	13500
nymaim	13500		

Tabella 5.1: *DGA che costituiscono il dataset*

5.2 Divergenza di Kullback-Leibler e Distanza di Jaccard

Gli algoritmi più efficaci per eseguire la classificazione sono basati sulle tecniche di Machine Learning. Alcuni di essi necessitano dell'estrazione delle feature mentre altri non ne hanno bisogno. Le features più comunemente estratte dai nomi di dominio possono essere classificate come segue:

- Feature di tipo statistico
- Feature basate sulla teoria dell'informazione
- Feature lessicali

Tra queste ultime, alcune delle più efficaci sono la **Divergenza di Kullback-Leibler** e la **Distanza di Jaccard**[5].
La definizione di tali feature consente quindi di realizzare una misura della similitarietà di ciascun nome di dominio rispetto ai gruppi assunti come benigni o maligni suddivisi in diversi set, per un totale di 26 (1 set per i benigni e 25 per i maligni), ciascuno relazionato a ognuna delle 25 famiglie di DGA conosciute. Per calcolare questo è possibile considerare i 2-grammi e i 3-grammi estratti da ciascun nome di dominio e utilizzare la Divergenza di Kullback-Leibler o la Distanza di Jaccard per misurare la differenza tra gli n-grammi presenti nei due tipi di domini e le loro distribuzioni.

L'Indice di Jaccard è una metrica per misurare la similitarietà tra due set di campioni A e B, essa è definita come segue:

$$JI(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

La Divergenza di Kullback-Leibler invece è una metrica della distanza tra due distribuzioni di probabilità P e Q. Di seguito è riportata la variante per distribuzioni discrete (data la natura delle feature): date due distribuzioni di k elementi, $P = \{p_1, p_2, ..., p_k\}$ e $Q = \{q_1, q_2, ..., q_k\}$ è definita come:

$$KL(P, Q) = \sum_{i=1}^{k} p_i log \frac{p_i}{q_i}$$

con P distribuzione degli n-grammi del singolo dominio e Q dell'intero set.

5.3 Codici utilizzati

Gran parte dei codici utilizzati nella seconda parte del progetto derivano da precedenti progetti e lavori di tesi sviluppati da altri studenti negli anni passati.
I codici dei progetti in questione sono tutti disponibili e corredati di documentazione, inoltre funzionano tutti in maniera abbastanza simile per quanto riguarda la definizione degli input e degli output.
Per comprendere meglio il loro funzionamento e il modo in cui operano sui dati è stata prodotta una mappa concettuale che riporta gli script in ordine relativo alla sequenza in cui devono essere attivati e i dati che essi richiedono in ingresso e che producono in uscita, andando a formare quindi una vera e propria catena di attivazione riportata in figura 3.1.

5.4 Prima parte

Nella prima parte del progetto sono state svolte una serie di operazioni preliminari sul dataset descritte nelle seguenti fasi.

5.4.1 Sanificazione del dataset

Il primo problema che è stato affrontato è stato quello relativo alla sanificazione del dataset. Questa procedura si è resa necessaria in quanto all'interno del dataset sono stati riscontrati domini malformati o che presentavano TLD non validi. Lo script Python che si occupa di questo, in maniera automatica scorre tutta la struttura di directory che compone il dataset, aprendo di volta in volta i file che incontra per analizzarne il contenuto. L'analisi consiste nell'andare a verificare, riga per riga, se l'ultimo elemento ottenuto dallo split fatto sul carattere `'.'` di quest'ultima (ovvero il TLD) è presente all'interno della lista dei TLD ammessi. Se questo controllo fallisce il nome di dominio viene aggiunto alla lista dei nomi di dominio non corretti e conteggiato come tale, altrimenti viene aggiunto alla lista dei domini corretti.

```
    splitted_domains = [x.split('.') for x in ceck_domain]

    correct_domains = []
    incorrect_domains = []

    for x in splitted_domains:
        if x[-1] in pl_domain:
            correct_domains.append(".".join(x))
        else:
            incorrect_domains.append(".".join(x))
```

Listato 5.2: Porzione del codice che esegue il controllo

Arrivati alla fine del file, il contenuto delle due liste verrà depositato sottoforma di file all'interno di due directory, una per i domini corretti e una per quelli non corretti. Fatto questo la procedura si ripete con il file successivo. Si veda la figura 5.1 a chiarimento di quanto detto.

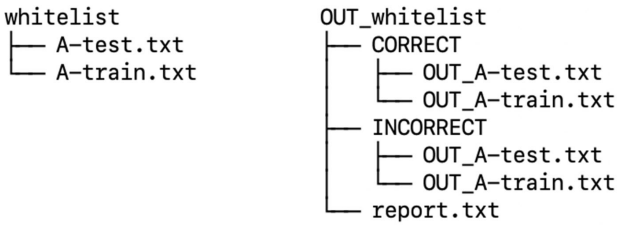

Figura 5.1: *Struttura di una sottodirectory del dataset prima e dopo*

Questa procedura è stata ripetuta per tutte e tre le parti del dataset, setAtrain, setAtest e setB, sia per i domini white che i black.

5.4.2 Indicizzazione dei TLD

Definiamo con T l'insieme di tutti i TLD ammissibili. Indicizzare T significa andare a definire un'associazione $f : T \to \mathbb{N}$ di tipo *1 a 1* tra ogni elemento $t \in T$ e un numero $n \in \mathbb{N}$ tale che $0 \leq n \leq dim(T) - 1$. Per realizzare l'associazione, è stato scritto uno script che prende in ingresso il file contenente tutti i nomi di dominio possibili `"TLD_standard.txt"` e associa ad ognuno di essi un numero n corrispondente alla posizione che occupa all'interno del file. In Python questo è realizzabile in maniera molto veloce, si veda il listato 5.3.

```
1          tld = open("tld-list-basic.txt","r").read()
2          indice_tld = dict(zip(tld, range(len(tld)))) # indice_tld = {'aaa' : 0, 'aarp'
       : 1, 'abarth' : 2, ...}
3
4          with open(sys.argv[3] + f, "w") as target:
5              for k in file:
6                  target.write(str(indice_tld[k]) + "\n")
7          target.close()
```

Listato 5.3: Codice che indicizza i TLD

Il codice riportato importa i TLD letti dal file all'interno della lista `tld`. Successivamente viene generato un dizionario Python `indice_tld` a partire dal ritorno della funzione `zip()` che non fa altro che generare una lista i cui elementi sono a loro volta delle liste e ognuna di queste ultime avrà nella posizione `0` il TLD e nella posizione `1` un numero naturale n compreso tra 0 e `len(tld)`. Il commento alla riga `2` del listato fornisce un esempio della struttura dati appena descritta.
Fatto questo, non rimane altro che depositare il dizionario appena ottenuto all'interno di un file, operazione svolta a partire dalla riga `4`.
Il risultato finale sarà quindi un file CSV (listato 5.4) contenente, oltre al campo relativo al TLD, un campo numerico valorizzato con il valore desiderato.

```
1  tld-indexed.csv
2    |
3    |_ aaa;0
4       aarp;1
5       abarth;2
6       abb;3
7       abbott;4
8       ...
```

Listato 5.4: Estratto del file CSV prodotto dallo script

5.4.3 Adattamento di un sottoinsieme del dataset principale

Essendo questo progetto legato ad uno precedente, sarebbe interessante confrontare i risultati prodotti da entrambi. Per fare questo, sarebbe opportuno addestrare il classificatore con lo stesso sottoinsieme del dataset utilizzato per il progetto precedente.
Tale sottoinsieme è composto da tre file CSV, (`setAtrain`, `setAtest` e `setB`), ognuno contenente i nomi di dominio, corredati di altre informazioni (si veda listato 5.5) quali famiglia di appartenenza, l'etichetta `legit/dga` e lo stesso nome di dominio privato del TLD.

```
setAtest_full_domains.csv
 |
 |_ dga,kraken,fscmjqgmbtv,fscmjqgmb.tv
    legit,alexa,amaatacom,amaata.com
    dga,ramnit,vdiopxvufbcom,vdiopxvufb.com
    dga,matsnu,platformkissreferenceintendcom,platformkissreferenceintend.com
    dga,matsnu,gloveholechipbathatestablecom,gloveholechipbathatestable.com
    legit,alexa,guiadeconcursoscom,guiadeconcursos.com
    dga,kraken,pybmrwztgnet,pybmrwztg.net
    dga,kraken,ycchaazivyiorg,ycchaaziv.yi.org
    legit,alexa,hendycarstorecouk,hendycarstore.co.uk
    ...
```

Listato 5.5: Struttura CSV dei sottoinsiemi

Prima di procedere si è provveduto ad estrarre tutti i nomi di dominio dai 3 file CSV e a raggrupparli in singoli file secondo la classe di appartenenza, passando dalla situazione descritta dal listato 5.5 alla situazione descritta nel listato 5.6.

```
setAtest_full_domains
    |-- alexa.txt
    |-- conficker.txt
    |-- corebot.txt
    |-- cryptolocker.txt
    |-- dircrypt.txt
    |-- emotet.txt
    |-- fobber.txt
    |-- gozi.txt
    |-- kraken.txt
    |-- matsnu.txt
    |-- murofet.txt
    |-- necurs.txt
          ...
```

Listato 5.6: Struttura finale dei sottoinsiemi

Le stesse operazioni sono state ripetute anche per i sottoinsiemi `setAtest` e `setB`.

5.4.4 Sanificazione di un sottoinsieme estratto dal dataset originale

Essendo il sottoinsieme già estratto, necessita di essere sottoposto alla stessa procedura di sanificazione del dataset principale descritta in precedenza. Ultimata questa, onde evitare che nei sottoinsiemi fossero presenti nomi di dominio non appartenenti al dataset di partenza, è stata eseguita un'ulteriore controverifica. Lo script che se ne occupa (listato 5.7) esegue una differenza tra il sottoinsieme di partenza e l'intero dataset, verificando che il risultato sia nullo.
Lo scopo di questo è escludere eventuali nomi di dominio presenti nel sottoinsieme ma non nel dataset di partenza.

```
1       pool = []
2       for f in file_list:
3           read_domain = open(input_name + "/" + f, "r").read()
4           pool.extend(read_domain)
5       target_domain = open(target_file,"r").read()
6       not_present = []
7       present = []
8       for d in target_domain:
9           if d not in pool:
10              not_present.append(d)
11          else:
12              present.append(d)
```

Listato 5.7: Codice che implementa la differenza tra gli insiemi

Il codice proposto importa all'interno della lista Python `pool` il contenuto di tutti gli eventuali file che compongono il dataset di partenza in cui vanno verificate le occorrenze. Successivamente provvede ad importare nella lista `target_domain` gli elementi da verificare e, a partire dalla riga `8`, provvede a confrontare ogni elemento del set target all'interno del set specificato. Se l'elemento non è presente viene immagazzinato nella lista `not_present`, mentre se invece lo è viene memorizzato nella `present`. La differenza sarà quindi nulla se, al termine dell'esecuzione, gli elementi contenuti nel risultato di quest'ultima saranno pari a zero.
Questa operazione di verifica è stata svolta prendendo come target tutti e tre i sottoinsiemi (`setAtrain_full_domains`, `setAtest_full_domains`, `setB_full_domains`) e come pool il set totale.

TARGET	OPERAZIONE	POOL	RISULTATO
setAtrain_full_domains	–	setAfull	$\varnothing$
setAtest_full_domains	–	setAfull	$\varnothing$
setB_full_domains	–	setBfull	$\varnothing$
setAtrain_full_domains	–	setAtest_full_domains	$\neq \varnothing$
setAtest_full_domains	–	setB_full_domains	$\varnothing$
setB_full_domains	–	setAtrain_full_domains	$\varnothing$

Tabella 5.2: *Tabella delle operazione svolte*

Come si può evincere dalla tabella 5.2 la quarta operazione non ha dato come risultato un insieme vuoto. Ciò significa che all'interno del `setAtrain` sono presenti dei domini appartenenti ad `Atest`, (10 per la precisione). Per risolvere questo problema è stato deciso di sostituire i 10 domini con altrettanti prelevati dal set completo `Afull`.
Per realizzare questo è stato scritto uno script che, partendo dai domini da sostituire, ne preleva altrettanti dal set completo, purchè essi siano della stessa famiglia di quelli da sostituire (tutti e 10 sono white), siano scelti in maniera casuale e non siano già presenti né in `setAtrain` e né in `setAtest`.

```
1  f = 10
2  extracted = []
3  extracted_domain = []
4  max = len(all_correct_dom)-1
5  for j in range(f):
6      r = randint(0, max)
7      while r  in extracted or all_correct_dom[r] in extracted_domain:
8          r = randint(0, i)
9      extracted_domain.append(all_correct_dom[r])
10     extracted.append(r)
```

Listato 5.8: Codice che esegue la sostituzione dei nomi di dominio

Il listato 5.8 riporta la parte dello script relativa alla selezione dei nomi di dominio dal `setAFull`. L'idea che c'è alla base è quella di generare un numero casuale compreso tra 0 e `dim(all_correct_dom)-1` ovvero la dimensione del set da cui prelevare i domini sostitutivi meno uno, in modo da selezionare un nome di dominio in modo casuale in esso. Nello specifico `all_correct_dom` rappresenta la differenza tra `setAfull` e `setAtrain`, in questo modo siamo sicuri di escludere la possibilità di selezionare un elemento già presente nel `setAtrain`. Fatto questo bisogna controllare che il nome di dominio selezionato non sia stato già estratto, questo è implementato alla riga 7 con un ciclo while che ripete l'estrazione fintanto che non se ne estrae uno diverso dai precedenti. Alla fine si otterrà un dominio diverso da tutti quelli estratti in precedenza e verrà aggiunto al `setAtrain` precedentemente privato dei 10 domini da scartare.

5.5 Seconda parte

Ultimate le operazioni preliminari sui dati, la seconda parte del progetto consiste nell'attivare una serie di script in sequenza che consentiranno di trasformare i dati grezzi nel formato richiesto dal classificatore. Questa sequenza di script prende il nome di catena di attivazione. Ogni elemento della catena costituisce una fase che può comprendere uno o più script. Nella figura 3.1 è riportata una rappresentazione grafica delle fasi, con indicati sia il flusso delle attivazioni, sia il flusso dei dati.

5.5.1 Splitting

Per suddividere in maniera automatica tutti i nomi di dominio dei due set che conterranno, rispettivamente, nome e TLD, è stato realizzato uno splitter. Esso riceve in ingresso il path della directory su cui operare, il path della directory in cui andrà a depositare i nomi di dominio e il path di quella in cui andrà a depositare i TLD. Grazie alla potenza del linguaggio Python è possibile eseguire questo in sole tre righe di codice (listato 5.9).

```
1          elements = [x.split('.') for x in file] # elements = [['duckduckgo','com'],['
       github','com'],...]
2          tld = [x[-1] for x in elements]
3          domain = ['.'.join(x[:-1]) for x in elements]
```

Listato 5.9: Come eseguire uno splitting in Python

Nella riga `1`, ad ogni elemento della lista `elements` viene associato il risultato dello splitting dei nomi di dominio letti dal file. Dopo questo, ogni elemento i-esimo di `elements` conterrà un'ulteriore lista da due o più elementi contenente sia l'i-esimo nome di dominio in tutte le sue eventuali componenti (2LD, 3LD, ...) che il TLD ad esso associato, il tutto realizzato con una sola riga di codice. Successivamente vengono prima esportati i TLD da `elements`, salvati all'interno di `tld` e dopo, sempre con un'unica riga di codice, nella i-esima posizione della lista `domain` vengono raccordati tra loro tramite il carattere `'.'` tutti gli elementi rimanenti occupanti la posizione i-esima di `elements`. Questo viene ripetuto per tutti gli elementi contenuti in `elements`.
Anche questo, come gli script precedenti, naviga in maniera ricorsiva in tutta la struttura della directory di partenza alla ricerca dei file su cui andare a lavorare.
Al termine dell'esecuzione si avranno due nuove directory che riprodurranno la struttura di quella in ingresso, con all'interno i file contenenti i nomi di dominio nel primo caso e i TLD nel secondo. È importante specificare che l'associazione tra nome di dominio e il suo TLD non viene persa, ma è mantenuta grazie alla struttura speculare delle due directory prodotte. Il listato 5.10 riporta un esempio della struttura ottenuta per il `setB` al termine della procedura.

```
setB_full_domains_presenti
    |-- Domains
    |   |-- alexa.txt
    |           |-  0-km
    |           |-  001fans
    |                 ...
    |   |-- banjori.txt
    |   |-- chinad.txt
    |   |__ qakbot.txt
    |-- TLD
    |   |-- alexa.txt
    |           |-  com
    |           |-  com
    |                 ...
    |   |-- banjori.txt
    |   |-- chinad.txt
    |   |__qakbot.txt
    |-- alexa.txt
    |      |-  0-kms.com
    |      |-  001fans.com
    |            ...
    |-- banjori.txt
    |-- chinad.txt
    |-- qakbot.txt
```

Listato 5.10: Struttura finale della directory ottenuta dallo script

5.5.2 Calcolo dei bigrammi e dei trigrammi

In questa fase è stato eseguito il calcolo dei *bigrammi* e dei *trigrmmi* che compongono gli elementi del dataset. Con bi/tri-gramma si indicano tutte le possibili combinazioni di due/tre caratteri contigui che possono essere associati ad una stringa data. Si consideri una stringa di caratteri qualsiasi, ad esempio `"qwerty"`. I possibili bigrammi estraibili risultano essere `{'qw','we','er','rt','ty'}`, mentre i trigrammi risultano `{'qwe','wer','ert','rty'}`. Per la generazione dei dizionari di tutti i possibili bigrammi e trigrammi ottenibili è stato

realizzato uno script (si veda il listato 5.11) che, partendo dall'insieme dei caratteri ammissibili per i nomi di dominio (`/([a-z0-9-_.])/`), provvede a generarli in maniera automatica già formattati in sintassi Python, pronti per essere importati negli script che eseguono fisicamente il calcolo.

```
1  alpha = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d","e", "f"
       , "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r",
2           "s", "t", "u", "v", "w", "x", "y", "z", "-",".", "_"]
3
4  with open("bigrammi.txt", "w") as target:
5      target.write("alpha = {")
6      for i in range(len(alpha)):
7          for j in range(len(alpha)):
8              target.write("\"" + alpha[i] + alpha[j] + "\": 0, ")
9      target.write("}")
10     target.close()
11
12 with open("trigrammi.txt", "w") as target:
13     target.write("alpha = {")
14     for i in range(len(alpha)):
15         for j in range(len(alpha)):
16             for k in range(len(alpha)):
17                 target.write("\"" + alpha[i] + alpha[j] + alpha[k] + "\": 0, ")
18     target.write("}")
19     target.close()
```

Listato 5.11: Codice per la generazione dei dizionari

In quanto i codici utilizzati, descritti nel paragrafo 5.3, sono stati realizzati nel contesto di un progetto passato in cui non venivano considerate le componenti dei nomi di dominio di ordine superiore al secondo (3LD, 4LD, ...), essi non tengono conto del carattere `'.'` all'interno dei loro dizionari. Per risolvere questo problema i dizionari degli script sono stati sostituiti con i dizionari appena generati, consentendo quindi di riutilizzare questi codici che, una volta lanciati, hanno impiegato qualche giorno per completare il calcolo.

Essendo i trigrammi in numero maggiore rispetto ai bigrammi, per ottimizzare i tempi di calcolo, lo script utilizzato è stato scritto implementando i principi della programmazione concorrente. Nello specifico è stato utilizzato il modulo `Threading` del linguaggio Python, assegnando ad ogni thread il compito di calcolare la distribuzione dei trigrammi di una particolare classe di domini, per un totale di 26 Thread. Come riportato nel listato 5.12, per primo passo è stato wrappato l'override del metodo costruttore `__init__` della classe `Thread` tramite la classe utente `IlMioThread`. Questo stratagemma consente di creare un oggetto di tipo `Thread`, wrappato all'interno di un oggetto di tipo `IlMioThread`, così facendo è possibile inizializzare `IlMioThread` con i parametri desiderati. Questi parametri a loro volta verranno utilizzati per inizializzare all'interno dell'oggetto `IlMioThread` l'oggetto di tipo `Thread`. Questo viene fatto a partire dalla riga `8`, dove appunto vengono creati gli n oggetti di tipo `IlMioThread` e ad ognuno di essi è passata la famiglia di DGA su cui andrà ad operare. Dichiarati quindi i diversi thread, l'operazione successiva è quella di lanciarli tutti in maniera concorrente tramite il metodo `start()`.

```
class IlMioThread(Thread):
    def __init__(self,nome,args):
        Thread.__init__(self)
        self.nome = nome
        self.args = args
                    ...
# Creazione dei thread
thread1 = IlMioThread("Thread#1", args=('whitelist'))
thread2 = IlMioThread("Thread#2", args=('conficker'))
thread3 = IlMioThread("Thread#3", args=('corebot'))
                ...
thread1.start()
thread2.start()
thread3.start()
                ...
thread1.join()
thread2.join()
thread3.join()
                ...
```

Listato 5.12: Componenti per implementare l'esecuzione concorrente nel calcolo dei trigrammi

Infine, dopo aver lanciato tutti i thread per calcolare i trigrammi sulle diverse porzioni del dataset, il Thread principale richiama il metodo `join()` per ogni thread figlio; questo consente al programma principale di sospendersi nell'attesa che tutti i thread figli abbiano completato il loro lavoro.

Il risultato finale di questa fase è quindi composto da 52 file, 25 black + 1 white per i bigrammi e 25 black + 1 white per i trigrammi, pronti ad essere impiegati per la fase successiva.

5.5.3 Calcolo della Divergenza di Kullback-Leibler e della Distanza di Jaccard

Ultimati i preparativi, si è passato al calcolo vero e proprio delle feature. Gli script che eseguono il calcolo della **Divergenza di Kullback-Leibler** (**KL**) e della **Distanza di Jaccard** (**JI**) fanno parte della raccolta degli script del progetto, quindi non sono stati oggetto di modifica. Per cercare di diminuire i tempi di calcolo, si è scelto di suddividere il file relativo ai 337.500 domini appartenenti ad alexa (white) in 25 parti composte ciascuna da 13.500 elementi, al pari di ognuna delle 25 famiglie black dei DGA ottenendo quindi un dataset (si veda il listato 5.13) composto in totale da 50 file o blocchi (25 alexa + 25 black). Ottenuti i blocchi, ognuno di essi è stato impiegato due volte, la prima per il calcolo dei valori della KL e la seconda per quelli relativi alla JI.

```
dataset/
|
|--  white/
|      |
|      |--  alexa00.txt
|      |--  alexa01.txt
|      |--  alexa02.txt
|             ...
|--  black/
|      |
|      |--  conficker.txt
|      |--  corebot.txt
|      |--  cryptolocker.txt
|             ...
```

Listato 5.13: Struttura del dataset dopo la suddivisione di alexa in 25 parti

Lo script relativo alla KL prende in ingresso il path relativo contenente il blocco oggetto del calcolo e tutte le distribuzioni calcolate nella fase precedente e ritorna in uscita un file di output dove ogni singolo dominio caratterizzato appare come un array di 53 elementi (52 feature KL più il nome di dominio). Per lo script relativo alla JI vale un discorso del tutto analogo al precedente, con la differenza che, in questo caso, non si utilizzano le distribuzioni. Anche in questo caso, in uscita si ha un file contenente ogni singolo dominio del blocco caratterizzato come un array di 53 elementi (52 feature JI più il nome di dominio). In entrambi i casi, l'ordine di calcolo delle feature per ogni dominio è lo stesso.

Questa, dal punto di vista computazionale, è stata la fase più lunga in assoluto.
Nonostante la suddivisione del dataset in 50 blocchi, il calcolo dei 100 blocchi complessivi tra KL e JI ha richiesto una settimana di tempo, nonostante venissero calcolati 40 blocchi alla volta grazie alle 40 CPU messe a disposizione dell'ambiente di esecuzione SMT.
Terminato il calcolo delle feature su tutti i blocchi, la situazione finale è riassunta dalla seguente tabella.

SCRIPT	TIPOLOGIA	NUMERO FILE OTTENUTI
`KL.py`	WHITE	25
`KL.py`	BLACK	25
`JI.py`	WHITE	25
`JI.py`	BLACK	25

Tabella 5.3: *Tabella riassuntiva dei file ottenuti*

5.5.4 Composizione dei Feature Vector

Terminato il calcolo delle feature operato nella fase precedente, come si può evincere dalla tabella 5.3 sono stati ottenuti in totale 100 file diversi. Risulta quindi necessario condensare i risultati relativi alle feature KL e JI di ogni dominio passando quindi da due file che caratterizzano uno stesso blocco (uno per le feature KL e uno per quelle JI) ad un unico file per ogni blocco che riunisce al suo interno un unico vettore per ogni nome di dominio, comprendente: le 52 feature KL, le 52 feature JI e il nome di dominio a cui le feature risultano associate. La struttura dati appena descritta prende quindi il nome di **Feature Vector**. Oltre a quanto

appena descritto, al Feature Vector di ogni domino vengono aggiunti ulteriori tre campi che contengono: famiglia di appartenenza, il codice relativo alla famiglia e la label binaria che indica se il dominio è white o black. In totale si avrà la situazione descritta in tabella 5.4.

NOME FEATURE	NUMERO DI OCCORRENZE	DESCRIZIONE
`KL`	52	feature KL
`JI`	52	feature JI
`dns`	1	nome di dominio
`family`	1	famiglia di appartenenza
`family_code`	1	codice famiglia
`label`	1	etichetta binaria black/white
	108	**TOTALE**

Tabella 5.4: *Tabella riassuntiva delle feature*

Lo script che esegue questo fa parte di quelli già scritti del progetto e prende il nome di `merge.py`. Esso necessita di conoscere il path di due directory, contenenti i dataset composti da 50 blocchi caratterizzati rispettivamente dalle feature KL e dalle feature JI e il path della directory dove andrà a depositare i file prodotti. Lo script definisce una funzione chiamata `unisci()` il cui prototipo è riportato nel listato 5.14, la quale riceve in ingresso il nome del blocco caratterizzato con la KL, il nome di quello caratterizzato con la JI, il nome della famiglia caratterizzata e la label binaria. Il listato 5.15 mostra invece come è stata invocata la funzione.

```
1 def unisci(file_kl,file_ji,family,label):
```

Listato 5.14: Prototipo della funzione unisci()

```
1  #black
2  unisci('conficker_KL','conficker_JI','conficker',1)
3  unisci('corebot_KL','corebot_JI','corebot',1)
4  unisci('cryptolocker_KL','cryptolocker_JI','cryptolocker',1)
5                           ...
6  #white
7  unisci('WLsetA_part0_KL','WLsetA_part0_JI','white',0)
8  unisci('WLsetA_part1_KL','WLsetA_part1_JI','white',0)
9  unisci('WLsetA_part2_KL','WLsetA_part2_JI','white',0)
10                          ...
```

Listato 5.15: Invocazione della funzione unisci() all'interno dello script merge.py

Il funzionamento della `unisci()` è molto semplice. Il contenuto dei due blocchi ricevuti per parametro viene letto e importato all'interno delle liste `rows_kl` ed `rows_ji`, la prima per contenere i feature vector dei blocchi caratterizzati da KL, la seconda per quelli caratterizzati da JI. Dopo aver ricavato un'unica riga di intestazione per il nuovo file, ogni feature vector contenuto nelle due liste viene valutato tramite la funzione Python `eval()`, diventando a tutti gli effetti una struttura dati semplificando notevolmente le operazioni successive. A questo punto si esegue un controllo per verificare che le feature caratterizzate tramite KL e quelle caratterizzate tramite JI siano riferite allo stesso nome di dominio (riga 12 del listato 5.16), in caso positivo viene eseguito il merge tra i due feature vector, con l'aggiunta dei

campi `family` , `label` e infine scritto nel file di out, in caso contrario lo script interrompe la propria esecuzione fornendo un messaggio di errore.

```
    for e1,e2 in zip(rows_kl,rows_ji):
        if flag == True:
            e1 = eval(e1)
            e2 = eval(e2)
            del e1[56]
            merge = e1 + e2 + ['family'] + ['label']
            file_out.write(str(merge)+'\n')
            flag = False
        else:
            e1 = eval(e1)
            e2 = eval(e2)
            if e1[56] == e2[54]:
                del e1[56]
                merge = e1 + e2 + [family] + [label]
                file_out.write(str(merge)+'\n')
            else:
                exit("Nessun matching tra nomi dns")
```

Listato 5.16: Implementazione della funzione unisci()

Aggiunta del campo relativo al TLD

Terminato la fase precedente, è stato necessario includere nei blocchi ottenuti il campo `tld_code` che raccogliesse il codice associato al TLD di ogni dominio contenuto in ogni blocco precedentemente separato nella Fase di splitting. Essendo stato mantenuto l'ordine posizionale nel corso delle fasi precedenti questa operazione è risultata abbastanza semplice in quanto è bastato ripetere in maniera molto simile quanto fatto per la fase di merge precedente.
Al termine di questa fase avremo quindi il dataset composto da 50 blocchi (25 white + 25 black) ed ognuno di essi conterrà 13.500 feature vector strutturati secondo lo schema riportato nella tabella 5.5.

NOME FEATURE	NUMERO DI OCCORRENZE	DESCRIZIONE
KL	52	feature KL
JI	52	feature JI
dns	1	nome di dominio
family	1	famiglia di appartenenza
family_code	1	codice famiglia
label	1	etichetta binaria black/white
tld_code	1	codice relativo al TLD
	109	**TOTALE**

Tabella 5.5: *Tabella riassuntiva delle features con TLD*

5.5.5 Unione dei blocchi in un unico file

Prima di procedere, l'operazione da eseguire è quella di riunire sotto un unico file per ogni set (`setAtrain`, `setAtest` e `setB`) tutti i feature vector, dato che al momento si ha un blocco

per ognuna delle 25 famiglie di DGA e 25 per i domini alexa. Questa operazione risulta molto veloce e banale da svolgere tramite `bash` (listato 5.17).

```
$ cat setAtrain/* > setAtrain.txt
$ cat setAtest/* > setAtest.txt
$ cat setB/* > setB.txt
```

Listato 5.17: Comando bash per riunire più file in uno solo

Il comando `cat` consente di mostrare a video tutto il contenuto del file passatogli come parametro. Nel primo caso è stato passato il quantificatore universale `*` riferito alla directory `setAtrain/`, in sostanza si sta passando a `cat` tutto il contenuto della directory `setAtrain/`. Il risultato di questo comando sarebbe quello di stampare a video il contenuto di questi file, ma tramite l'operatore di redirezione `>` ridirigiamo l'output di `cat` all'interno di un nuovo file di testo `setAtrain.txt`, ottenendo quindi come risultato l'unione in un unico file di tutto il setAtrain. Analogamente viene ripetuta la stessa operazione anche per i set `setAtest` e `setB`.

5.5.6 Mescolamento dei dati

Per evitare che il classificatore venga addestrato con le feature ordinate per blocchi si rende necessario esegurire un rimescolamento delle righe del `setAtrain`, in modo da rendere la distribuzione dei dati uniforme. Lo script che si occupa di questo è lo `shuffle.py`. Questo script fa uso della funzione `random.shuffle()` messa a disposizione dalla libreria `rand` del Python che non fa altro che mischiare gli elementi della lista che le viene passata come parametro. Il compito di `shuffle.py` è quindi solo quello di leggere il contenuto dei set sotto forma di lista, passare questa alla `random.shuffle()` e infine scrivere in un nuovo file la lista ritornata da quest'ultima (listato 5.19).

```
import random

def script(nome):
    f = open(pathr + nome +'.txt', "r")
    list_domains = [line for line in f]
    f.close()
    random.shuffle(list_domains)
    out_file = open(pathw + nome +'_shuffle'+'.txt','a')
    for i in range(0,len(list_domains)):
        out_file.write(list_domains[i])
    out_file.close()

script('Atrain_totale')
script('Atest_totale')
script('B_totale')
```

Listato 5.18: Script per mescolare le righe dei dataset

5.5.7 Normalizzazione

Il prossimo passo consiste nel normalizzare tra 0 e 1 i valori numerici contenuti nei feature vector. La normalizzazione consiste essenzialmente nel limitare l'escursione di un insieme di

valori entro un certo intervallo predefinito, nel nostro caso l'intervallo $[0, 1]$. Viene svolta per tutti e 3 i set `setAtrain`, `setAtest`, `setB` e produce un file di testo con lo stesso formato di quello dato in input. Lo script che realizza questo è lo `Scaler.py` , presente insieme agli altri script del progetto.
`Scaler.py` ha bisogno in ingresso dei tre path delle directory che contengono i tre set da normalizzare e i tre path delle directory dove andrà a memorizzare i file di out. Al suo interno è definito il metodo `normalize()`, esso importa dai set i feature vector valutati come liste all'interno di un'ulteriore lista andando a formare di fatto una matrice m dove ogni riga contiene le feature di un nome di dominio.
Successivamente la lista di liste, o matrice m, è convertita in una struttura dati specifica della libreria NumPy per ottimizzare l'esecuzione dei calcoli. Una volta convertita la matrice in questa struttura dati, viene utilizzato il metodo `fit()` relativo all'oggetto `scaler` appartenente alla classe `MinMaxSclaer` della libreria `sklearn.preprocessing` per stimare, tramite i dati di addestramento, i valori minimi e massimi osservabili. Fatto questo, il metodo `scaler.transform()` applica la scala stimata ai dati ottenendo infine i valori normalizzati. Convertiti infine i risultati ottenuti in una normale lista Python, lo script si conclude andando a scrivere quest'ultima all'interno del file di out.

```
import random

    def normalize(file, file_out, limit):
        count = 0
        m = []
        s = []
        heared_row = None
        with open(file, "r") as f:
            #skip header row
            heared_row = f.readline()
            for row in f:
                if (limit!=None):
                    count += 1
                    if count>limit:
                        break
                r = eval(row)
                m.append(r)

        m = np.array(m)
        sm = m[:,:-3].astype(float)
        ss = m[:,-3:-1]
        ssl = m[:,-1:].astype(int)
        s = ss.tolist()
        sl = ssl.tolist()
        m = sm.tolist()
        #print(m)
        #print("\n")
        scaler = MinMaxScaler()
        #scaler = StandardScaler()
        scaler.fit(m)
        norm = scaler.transform(m)
        norm = norm.tolist()
                    ...
# Chiamate a normalize() dal programma principale
normalize(file_Atr, file_out_Atr, limit)
normalize(file_Ate, file_out_Ate, limit)
normalize(file_B, file_out_B, limit)
```

Listato 5.19: Estratto della funzione normalize dello script Scaler.py

5.5.8 Preparazione dei dati da fornire al classificatore

Terminata la fase di preprocessing, è giunto il momento di impacchettare i dati nel formato richiesto dal classificatore. Lo script che svolge questa operazione fa sempre parte degli script del progetto ed è `input_classifier.py`. Come prima cosa, questo script necessita di un file di testo che contiene tutte le feature che desideriamo fornire in ingresso al classificatore. Nel nostro caso il file conteneva le feature riportate nel listato 5.20.

```
['alexa_2g_KL', 'conficker_2g_KL', 'corebot_2g_KL', 'cryptolocker_2g_KL', '
    dircrypt_2g_KL', 'emotet_2g_KL', 'fobber_2g_KL', 'gozi_2g_KL', 'kraken_2g_KL', '
    matsnu_2g_KL', 'murofet_2g_KL', 'necurs_2g_KL', 'nymaim_2g_KL', 'padcrypt_2g_KL',
    'pushdo_2g_KL', 'pykspa_2g_KL', 'qadars_2g_KL', 'ramdo_2g_KL', 'ramnit_2g_KL', '
    ranbyus_2g_KL', 'rovnix_2g_KL', 'simda_2g_KL', 'suppobox_2g_KL', 'symmi_2g_KL', '
    tinba_2g_KL', 'vawtrak_2g_KL', 'alexa_3g_KL', 'conficker_3g_KL', 'corebot_3g_KL',
    'cryptolocker_3g_KL', 'dircrypt_3g_KL','emotet_3g_KL', 'fobber_3g_KL', 'gozi_3g_KL
    ', 'kraken_3g_KL', 'matsnu_3g_KL', 'murofet_3g_KL', 'necurs_3g_KL', 'nymaim_3g_KL'
    , 'padcrypt_3g_KL', 'pushdo_3g_KL', 'pykspa_3g_KL', 'qadars_3g_KL', 'ramdo_3g_KL',
     'ramnit_3g_KL', 'ranbyus_3g_KL', 'rovnix_3g_KL', 'simda_3g_KL', 'suppobox_3g_KL',
     'symmi_3g_KL', 'tinba_3g_KL', 'vawtrak_3g_KL', 'conficker_2g_JI', '
    conficker_3g_JI', 'corebot_2g_JI', 'corebot_3g_JI', 'cryptolocker_2g_JI', '
    cryptolocker_3g_JI', 'dircrypt_2g_JI', 'dircrypt_3g_JI', 'emotet_2g_JI', '
    emotet_3g_JI', 'fobber_2g_JI', 'fobber_3g_JI', 'gozi_2g_JI', 'gozi_3g_JI', '
    kraken_2g_JI', 'kraken_3g_JI', 'matsnu_2g_JI', 'matsnu_3g_JI', 'murofet_2g_JI', '
    murofet_3g_JI', 'necurs_2g_JI', 'necurs_3g_JI', 'nymaim_2g_JI', 'nymaim_3g_JI', '
    padcrypt_2g_JI', 'padcrypt_3g_JI', 'pushdo_2g_JI', 'pushdo_3g_JI', 'pykspa_2g_JI',
     'pykspa_3g_JI', 'qadars_2g_JI', 'qadars_3g_JI', 'ramdo_2g_JI', 'ramdo_3g_JI', '
    ramnit_2g_JI', 'ramnit_3g_JI', 'ranbyus_2g_JI', 'ranbyus_3g_JI', 'rovnix_2g_JI', '
    rovnix_3g_JI', 'simda_2g_JI', 'simda_3g_JI', 'suppobox_2g_JI', 'suppobox_3g_JI', '
    symmi_2g_JI', 'symmi_3g_JI', 'tinba_2g_JI', 'tinba_3g_JI', 'vawtrak_2g_JI', '
    vawtrak_3g_JI', 'alexa_2g_JI', 'alexa_3g_JI']
```

Listato 5.20: Feature selezionate per il classificatore

Selezionate le features da estrarre, bisogna indicare allo script i path delle directory di input e di output. All'interno della prima vanno inseriti i 3 set ottenuti grazie alle precedenti elaborazioni e, per ognuno di essi, bisogna eseguire un run dello script. Quest'ultimo, ogni volta che viene lanciato, produce 3 diversi file, chiamati rispettivamente `custom.txt`, `matrix.txt`, e `label.txt`. Il primo non va fornito al classificatore, serve solo per ispezionare le feature estratte, il secondo è costituito da un'unica riga, ovvero un array di array, ed ogni array più interno contiene le feature di ogni riga esclusa la feature `label`. Il terzo ed ultimo file invece ha una struttura analoga al precedente ma al posto delle feature, gli array più interni conterranno solamente la `label` che era stata esclusa nel precedente. Questi ultimi due file sono quelli che il classificatore prende in input durante la fase di train.
Dal momento che gli esperimenti eseguiti sono quattro, questo processo è stato ripetuto per ognuno di essi.

NOME	CLASSIFICATORE	METAFEATURES
Esperimento A	k-fold_atrainfull_classifier_binary.py	Label
Esperimento B	k-fold_atrainfull_classifier_binary.py	Label + tld_code
Esperimento C	k-fold_atrainfull_classifier_multiclass.py	Family_dga_code
Esperimento D	k-fold_atrainfull_classifier_multiclass.py	Family_dga_code + tld_code

Tabella 5.6: *Tabella degli esperimenti*

5.5.9 Classificatore

Preparati i dati, come accennato in precedenza, sono stati effettuati i quattro esperimenti. Per i primi due è stato utilizzato il classificatore binario su dieci fold con algoritmo *mlp*, per gli altri due è stato utilizzato un classificatore multiclasse, sempre con dieci fold e sempre con classificatore *mlp*.

La divisione del dataset di partenza in fold viene eseguita dall'algoritmo stesso per testare durante l'addestramento la bontà dei modelli ottenuti e salvare quindi il modello con la best-accuracy tra le varie fold.
I codici che implementano i classificatori erano già pronti in quanto realizzati in progetti precedenti. Essi in fase di invocazione necessitano del path della cartella relativa all'esperimento in cui andranno a operare. Nel momento in cui viene avviato il run è possibile inoltre scegliere se eseguire o meno il train, produrre o meno un report di sintesi e scegliere quale tra i tre classificatori supportati utilizzare.
Per i quattro esperimenti sono stati lanciati i classificatori con i parametri riportati nel listato 5.21.

```
1 [cyber@smt ~] $ python k-fold_atrainfull_classifier_binary.py "mlp" "../EsperimentoA" "
      True" "Total_report"
2 [cyber@smt ~] $ python k-fold_atrainfull_classifier_binary.py "mlp" "../EsperimentoB" "
      True" "Total_report"
3 [cyber@smt ~] $ python k-fold_atrainfull_classifier_multiclass.py "mlp" "../
      EsperimentoC" "True" "Total_report"
4 [cyber@smt ~] $ python k-fold_atrainfull_classifier_multiclass.py "mlp" "../
      EsperimentoD" "True" "Total_report"
```

Listato 5.21: Comandi per lanciare i classificatori

I classificatori supportati dagli script sono:

- `mlp` → Multi-layer Perceptron classifier
- `svm` → Support Vector Machine
- `rf` → Random Forest

Al termine dell'esecuzione di ogni esperimento, il classificatore ha prodotto (listato 5.22), oltre ai modelli addestrati delle dieci fold anche un report abbastanza dettagliato, per ogni fold, sulla precisione del riconoscimento relativo ad ogni classe di domini, sia nel caso binario che in quello multiclasse.

```
1
2  esperimento_A(binario_nopld)
3     |__ results
4         |-- Full_report
5         |__ mlp
6             |-- report
7             |   |__ report_k-fold_setAtrain_features_scaler_custom{
       max_iter_200_learning_rate_init_0.001_early_stopping_False_hidden_layer_sizes_128
       }.txt
8             |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_1.pkl
9             |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_10.pkl
10            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_2.pkl
11            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_3.pkl
12            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_4.pkl
13            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_5.pkl
14            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_6.pkl
15            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_7.pkl
16            |-- saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_8.pkl
17            |__ saved_model_mlp_{max_iter_200_learning_rate_init_0.001
       _early_stopping_False_hidden_layer_sizes_128}_fold_9.pkl
```

Listato 5.22: Struttura della directory dell'esperimento A

Per quanto riguarda i tempi di elaborazione mediamente i quattro esperimenti hanno impiegato tutti all'incirca tre giorni di computazione continuativa. A differenza dei calcoli delle distribuzioni, in questo caso non è stato possibile far partire tutti e quattro gli esperimenti contemporaneamente in quanto i codici dei classificatori sono già ottimizzati per l'elaborazione parallela e ognuno di essi occupava tutti e 40 i processori dell'ambiente SMT per la fase di addestramento.

5.6 Considerazioni sui risultati ottenuti

Terminati tutti e quattro gli esperimenti, è stato possibile trarre le considerazioni finali.

5.6.1 Classificazione binaria

Partendo dai risultati dell'esperimento A (Tabella 5.7), possiamo constatare che la precisione del riconoscimento, mediata sulle 10 fold, si aggira intorno al 94,3% per quanto riguarda i domini di tipo white, mentre cresce fino al 95,6% per quanto riguarda i domini di tipo black. Questo vuol dire che siamo in grado di determinare con maggiore precisione un dominio malevolo piuttosto che uno benevolo.

TIPO	PRECISION	RECALL	F1-SCORE
White	0.943297	0.95720	0.95011
Black	0.956823	0.942374	0.949446

Tabella 5.7: *Tabella dei risultati mediati nelle 10 fold dell'Esperimento A*

Osservando invece i risultati dell'esperimento B, possiamo notare immediatamente dalla tabella 5.8 un miglioramento della precisione, abbastanza marcato per i domini di tipo black, mentre un po' meno accentuato per quelli white, arrivando per i primi ad una percentuale del 96.5% mediata sulle 10 fold per i domini malevoli. Questo significa che l'aggiunta del campo relativo al TLD nella composizione dei feature vector per l'esperimento B ha prodotto un miglioramento della precisione del riconoscimento.

TIPO	PRECISION	RECALL	F1-SCORE
White	0.946411	0.965486	0.955741
Black	0.965057	0.945186	0.954895

Tabella 5.8: *Tabella dei risultati mediati nelle 10 fold dell'Esperimento B*

5.6.2 Classificazione multiclasse

Per quanto riguarda invece i risultati relativi agli esperimenti eseguiti con il classificatore multiclasse, in tabella 5.9 sono riportati i valori del parametro accuracy per ogni famiglia, riferiti sia all'esperimento C che all'esperimento D. In questo caso abbiamo che, mentre per alcune famiglie l'introduzione della feature relativa al TLD produce un miglioramento abbastanza sostanziale nell'accuratezza della classificazione (ad esempio per *dircrypt*, *murofet*, *pykspa* e così via), per altre questo produce un peggioramento dell'accuratezza, anche notevole in alcuni casi, come ad esempio per la famiglia *conficker* che passa da un valore pari al 73.4% nel caso privo di TLD ad una accuracy pari al 52,7%.

	ACCURACY	
FALIGLIA	**ESPERIMENTO C**	**ESPERIMENTO D**
alexa	0.967	0.969
conficker	0.734	0.527
corebot	0.956	0.842
cryptolocker	0.736	0.56
dircrypt	0.448	0.621
emotet	0.999	0.999
fobber	0.942	0.992
gozi	0.864	0.926
kraken	1.0	0.994
matsnu	1.0	1.0
murofet	0.779	0.921
necurs	0.815	0.951
nymaim	1.0	1.0
padcrypt	0.995	0.996
pushdo	0.958	0.961
pykspa	0.767	0.967
qadars	1.0	1.0
ramdo	0.924	0.809
ramnit	0.986	0.999
ranbyus	0.881	0.851
rovnix	0.744	0.828
simda	0.878	0.999
suppobox	0.881	0.936
symmi	0.92	0.893
tinba	0.757	0.744
vawtrak	0.95	0.959

Tabella 5.9: *Tabella dei risultati mediati nelle 10 fold della classificazione multiclasse*

Un'ulteriore osservazione da fare è quella che per le famiglie *emotet*, *kraken*, *matsnue* e *nymaim*, indipendentemente dalla presenza o meno del TLD, riusciamo a determinare con esattezza i domini che gli appartengono, in quanto l'accuratezza sale al 100%.

Capitolo 6

Conclusioni e sviluppi futuri

In questo lavoro è stato utilizzato un approccio per la classificazione dei nomi di dominio come dannosi o benigni basato solo su caratteristiche lessicali, vale a dire l'uso congiunto di due misure di similarità basate su n-grammi, JI e KL, applicate sia a bigrammi che a trigrammi associati ai nomi di dominio.
Dai risultati ottenuti si può concludere che il loro utilizzo può essere un buon punto di partenza per quanto riguarda questo tipo di caratterizzazione.
Questi ultimi hanno anche evidenziato che l'utilizzo dei metodi di apprendimento automatico rappresenta il futuro nel contrasto ad una delle tecniche più pericolose di attacco presenti nel panorama cybercrime. Essi consentono di rilevare con buona probabilità ed eventualmente bloccare questo tipo di attacchi sul nascere, potenzialmente in maniera del tutto automatica.

Per quanto riguarda eventuali sviluppi futuri, si potrebbe pensare di ottimizzare gli algoritmi implementati in questo lavoro nell'ottica di rendere il processo di preparazione dei dati per il classificatore del tutto automatico. Questo consentirebbe di concentrarsi esclusivamente sullo sviluppo degli esperimenti da eseguire e sull'analisi approfondita dei risultati, senza dover ogni volta eseguire tutti i passaggi intermedi illustrati in questo lavoro di tesi.

Un altro possibile sviluppo potrebbe essere quello di andare ad aumentare le dimensioni del dataset in ingresso, con lo scopo di verificare se un addestramento basato su una quantità maggiore di dati produce un miglioramento della capacità di riconoscimento del classificatore.

Un ulteriore sviluppo potrebbe consistere nel ripetere gli esperimenti proposti provando a variare l'algoritmo utilizzato dal classificatore tra quelli supportati e confrontare i risultati ottenuti in tutti i casi.

Come considerazione finale è opportuno fare una riflessione sul fatto che in questo lavoro si sono utilizzate delle tecniche di apprendimento automatico per eseguire degli esperimenti e confrontare i risultati, ma in un futuro non molto prossimo, le stesse tecniche potrebbero essere implementate all'interno di sistemi più complessi.
Essi saranno quindi in grado di rilevare autonomamente di essere sotto attacco ed, eventualmente, agire di conseguenza per impedire che questo avvenga, il tutto senza l'intervento dell'uomo.

Bibliografia

[1] Numpy.org. `https://www.numpy.org`.

[2] Official documentations of the python language. `https://docs.python.org/3/`.

[3] scikit-learn.org. `https://www.scikit-learn.org`.

[4] Scipy.org. `https://www.scipy.org`.

[5] L. S. Alessandro Cucchiarelli, Christian Morbidoni and M. Baldi. Malicious domain names detection based on n-grams features. *In Stampa*, 2019.

[6] F. Mambella. Uso di feature lessicali nell'analisi di nomi di dominio per il rilevamento di malware, 2017-2018.

[7] S. D. Saverio. Rilevazione di malware tramite analisi di richieste dns basata su deep learnig, 2018-2019.