

Università Politecnica delle Marche

Dipartimento di Ingegneria
dell'Informazione

Facoltà di Ingegneria Informatica e dell'Automazione



Studio e Classificazione di Domain Names malevoli tramite
classi di DGA con metodi di Deep Learning

Andrian Melnic

Edoardo Conti

Lorenzo Federici

Prof. Luca Spalazzi

ANNO ACCADEMICO 2022/2023

Ancona

Indice

1	Introduzione	4
1.1	Ambiente di sviluppo	5
2	Pre-processing	6
2.1	Recupero logs da server GARR	6
2.2	Decompressione ed Estrazione DNS	8
2.3	DNs N-gram Tokenizer	8
2.4	Addestramento modelli FastText	10
3	Rete di Classificazione	12
3.1	Dataset	12
3.2	Architettura Stacked Generalization	14
3.3	Architettura REFS	15
3.4	Architettura EFRP	16
4	Risultati	17
4.1	Stacked Generalization	18
4.1.1	Stacking UMUDGA 10%	18
4.1.2	Stacking UMUDGA 25%	22
4.2	REFS	26
4.3	EFRP	30
5	Conclusioni	34

Elenco delle figure

1.1	Algoritmi di generazione del dominio	5
1.2	Principali tecnologie utlizzate: Python e Google Colab	5
2.1	Script python in esecuzione per il recupero dei dati di log	7
2.2	Decompressione file di log ed estrazione dei DNS	8
2.3	Unigrammi (1-gram), Digrammi (2-grams) e Trigrammi (3-grams)	9
2.4	Libreria per l'apprendimento di incorporamenti di parole	10
3.1	Estratto dataset UMUDGA 1K	13
3.2	Schema architettura Stacking	15
3.3	Schema architettura REFS	16
3.4	Schema architettura EFRP	16
4.1	Confusion matrix modello Stacking UMUDGA 1K (10%)	21
4.2	Confusion matrix modello Stacking UMUDGA 1K (25%)	25
4.3	Confusion matrix modello REFS UMUDGA 1K (10%)	29
4.4	Confusion matrix modello EFRP UMUDGA 1K (10%)	33

Elenco delle tabelle

4.1	Percentuali Dateset UMUDGA utilizzate	17
4.2	Tempi di addestramento	17
4.3	Metriche Stacking UMUDGA 1k (10%)	18
4.4	Risultati Stacking UMUDGA 1k (10%)	18
4.5	Metriche stacking UMUDGA 1K (25%)	22
4.6	Risultati Stacking UMUDGA 1k (25%)	22
4.7	Metriche REFS UMUDGA 1k (10%)	26
4.8	Risultati REFS UMUDGA 1k (10%)	26
4.9	Metriche EFRP UMUDGA 1k (10%)	30
4.10	Risultati EFRP UMUDGA 1k (10%)	30
5.1	Confronto dei risultati ottenuti dai vari modelli	34

Capitolo 1

Introduzione

Questo lavoro si colloca nel dominio dell'analisi di malware responsabili della creazione di "Botnet", ovvero reti di computer infettati che vengono gestite da un hacker attraverso un server di comando e controllo. Di norma, per contrastare questo tipo di infezione, si cerca di individuare e oscurare il server centrale. Tuttavia, tale operazione è tutt'altro che semplice, poiché vengono adottate diverse strategie per ostacolarne l'individuazione. Tra queste, rientra la tecnica di modifica dei Domain Names tramite un algoritmo di generazione dei nomi (DGA, Domain Generation Algorithm).

L'algoritmo di generazione dei nomi di dominio (DGA) è una tecnica impiegata nei malware al fine di creare periodicamente una vasta gamma di nomi di dominio casuali e inesistenti, utilizzati per il server di comando e controllo. Il malware cerca, quindi, di risolvere tali nomi di dominio generati inviando richieste DNS fino a quando non riesce a individuare un dominio che risolva all'indirizzo IP di un server di comando e controllo. I nomi di dominio generati tramite DGA fungono da collegamenti tra il malware e il server di comando e controllo. L'utilizzo del DGA impedisce la disattivazione del server centrale e rende più complessa l'individuazione dei malware. Quindi, sfruttando questo processo, i DGA dimostrano la capacità di generare nomi di dominio che possono essere facilmente scambiati per autentici e simili a quelli reali. Che risulta essere il vero punto di forza di questa metodologia, rendendo estremamente onerosa l'analisi del problema.

La soluzione testata consiste nell'implementazione di un'architettura Deep Learning che consenta di classificare i nomi di dominio in categorie malevoli e non malevoli.

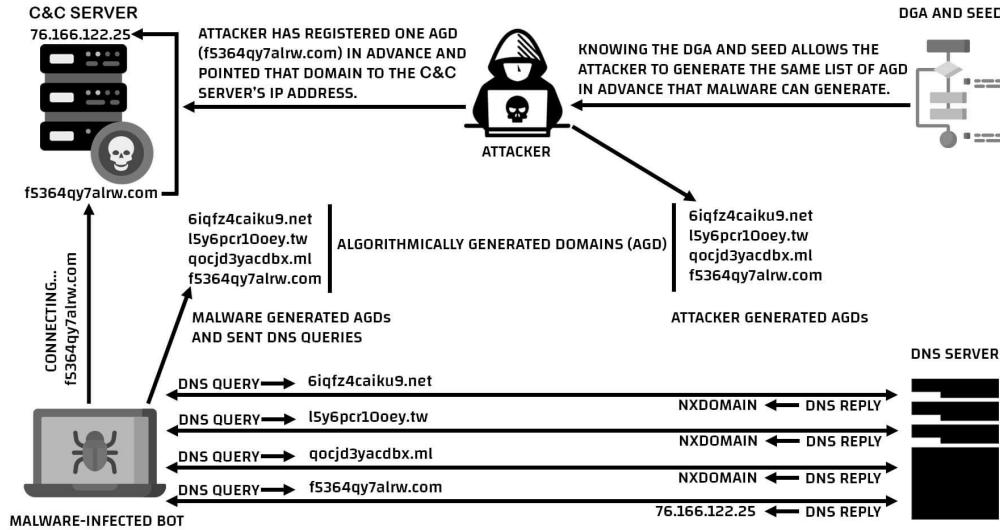


Figura 1.1: Algoritmi di generazione del dominio

1.1 Ambiente di sviluppo

Per la realizzazione del progetto, si è optato per Google Colab Pro e il linguaggio di programmazione Python. Colab è stato utilizzato come piattaforma per scrivere e eseguire il codice, sfruttando un backend con GPU Tesla T4, CUDA Toolkit 10 e cuDNN 7. L'utilizzo di Compute Units aggiuntive e delle risorse GPU hanno consentito di beneficiare dell'accelerazione hardware offerta da Tensorflow GPU, riducendo significativamente i tempi di addestramento dei modelli. Inoltre, per la gestione dei file, si è fatto affidamento su Google Drive, che ha facilitato l'organizzazione e l'accesso ai dati di progetto. Per il recupero dei dati di log dalla rete GARR, è stato sviluppato uno script Python da eseguire tramite linea di comando, consentendo un'interazione efficiente e il recupero dei dati desiderati.



Figura 1.2: Principali tecnologie utilizzate: Python e Google Colab

Capitolo 2

Pre-processing

Il presente capitolo si focalizza sulla fase di pre-processing, in cui verranno illustrati i passaggi svolti per ottenere i dati di log dalla rete GARR. Utilizzando uno script Python da riga di comando e i dati d'accesso SSH forniti, è stato possibile recuperare i logs necessari per l'analisi in questione. Una volta estratti, questi sono stati sottoposti ad un processo di elaborazione al fine di renderli adatti per l'allenamento della rete FastText. L'obiettivo era generare modelli FastText che potessero essere impiegati nella fase successiva di analisi e classificazione. Nel corso di questo capitolo, verranno esposti in dettaglio i metodi utilizzati per manipolare e preparare i dati, nonché le scelte adottate per ottimizzare l'efficacia dei modelli generati.

2.1 Recupero logs da server GARR

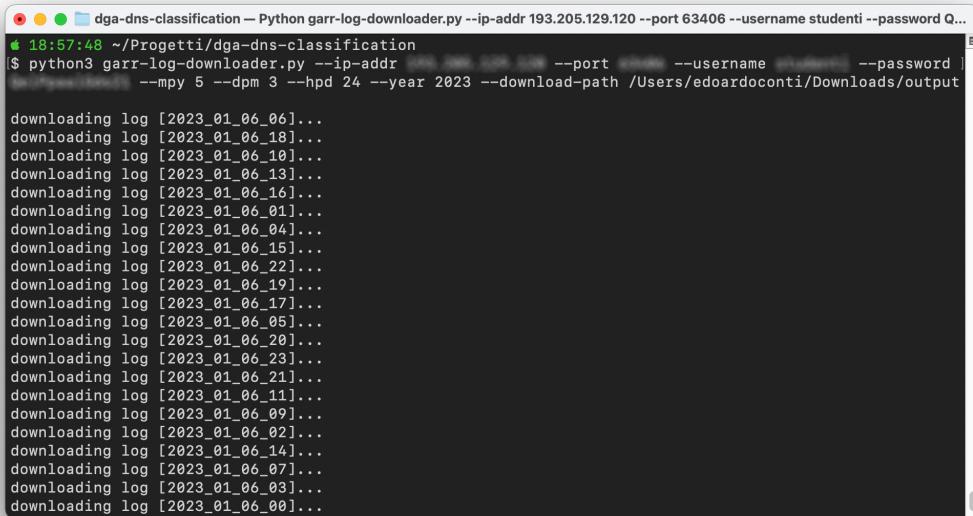
Lo script Python sviluppato permette il download dei dati di log dal server GARR in modo automatizzato. Per eseguire lo script, è necessario fornire i parametri di connessione al server, tra cui l'indirizzo IP, la porta, il nome utente e la password. Inoltre, vengono richieste alcune impostazioni opzionali per configurare i dettagli di filtraggio dei logs, come il numero di mesi per anno (mpy), il numero di giorni per mese (dpm), il numero di ore per giorno (hpd) e l'anno di riferimento.

Lo script utilizza il modulo `Paramiko` per stabilire una connessione SSH al server GARR, consentendo l'accesso remoto ai file di log. Successivamente, viene utilizzato il comando `SCP`

(Secure Copy) per scaricare i file di log desiderati. Gli argomenti forniti allo script determinano quali file di log verranno scaricati in base alle impostazioni specificate.

Durante l'esecuzione dello script, viene effettuato un controllo sui valori forniti per garantire che siano validi. Successivamente, utilizzando il modulo `calendar`, vengono generati in modo casuale i mesi da selezionare, seguiti dai giorni e dalle ore corrispondenti per ciascun mese. I file di log vengono quindi scaricati in base alle date generate. Al termine, viene chiusa la connessione SSH al server GARR.

Dunque, si è scelto di costruire il dataset di partenza scaricando i dati di log di 3 giorni casuali al mese, includendo tutte le 24 ore per ogni giorno, da Gennaio a Maggio 2023. Questa scelta ha comportato il download di un totale di 288 file con estensione ".log", compressi nel formato ".xz". L'insieme di file scaricati ha raggiunto una dimensione complessiva di circa 3 GB. Successivamente, tali file sono stati caricati su Google Drive per la fase vera e propria di preprocessing.



The screenshot shows a terminal window on a Mac OS X desktop. The title bar reads "dga-dns-classification — Python garr-log-downloader.py --ip-addr 193.205.129.120 --port 63406 --username studenti --password Q...". The command entered is "\$ python3 garr-log-downloader.py --ip-addr [REDACTED] --port [REDACTED] --username [REDACTED] --password [REDACTED] --mpy 5 --dpm 3 --hpd 24 --year 2023 --download-path /Users/edoardoconti/Downloads/output". The terminal output shows a series of messages indicating the download of log files from January 6, 2023, to January 23, 2023, with file names like "2023_01_06_06.log", "2023_01_06_18.log", etc.

```
$ python3 garr-log-downloader.py --ip-addr [REDACTED] --port [REDACTED] --username [REDACTED] --password [REDACTED] --mpy 5 --dpm 3 --hpd 24 --year 2023 --download-path /Users/edoardoconti/Downloads/output
downloading log [2023_01_06_06]...
downloading log [2023_01_06_18]...
downloading log [2023_01_06_10]...
downloading log [2023_01_06_13]...
downloading log [2023_01_06_16]...
downloading log [2023_01_06_01]...
downloading log [2023_01_06_04]...
downloading log [2023_01_06_15]...
downloading log [2023_01_06_22]...
downloading log [2023_01_06_19]...
downloading log [2023_01_06_17]...
downloading log [2023_01_06_05]...
downloading log [2023_01_06_20]...
downloading log [2023_01_06_23]...
downloading log [2023_01_06_21]...
downloading log [2023_01_06_11]...
downloading log [2023_01_06_09]...
downloading log [2023_01_06_02]...
downloading log [2023_01_06_14]...
downloading log [2023_01_06_07]...
downloading log [2023_01_06_03]...
downloading log [2023_01_06_00]...
```

Figura 2.1: Script python in esecuzione per il recupero dei dati di log

2.2 Decompressione ed Estrazione DNS

Da questo punto in poi le operazioni descritte saranno eseguite nell'ambiente Google Colab. In questa fase, i file di log vengono decompressi ed estratti per essere disponibili per l'elaborazione successiva. Durante la decompressione, i file risultanti vengono salvati nella cartella di output tenendo traccia del numero di file di log estratti. Successivamente, viene eseguita un'altra operazione di estrazione. Questa riguarda specificamente l'estrazione dei DNS dai file di log precedentemente decompressi. In questa fase vengono eseguite diverse operazioni di pulizia dei log, quali la rimozione di righe non valide e la selezione di un sottoinsieme casuale dei log. I DNS estratti ammontano a 13'859'029 e vengono salvati in nuovi file di log nella cartella di output corrispondente.

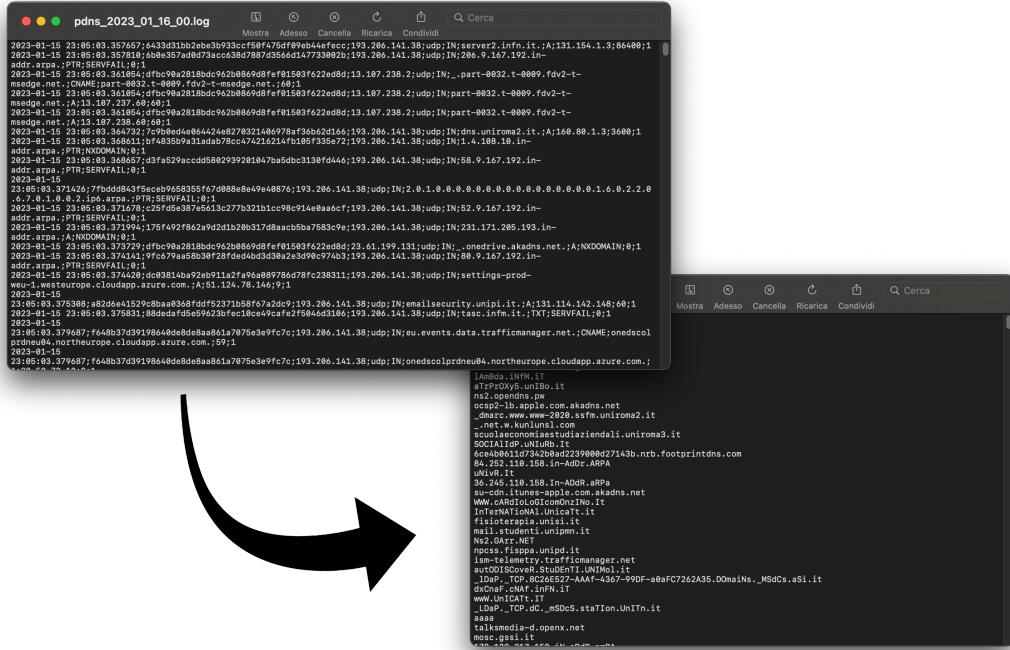


Figura 2.2: Decompressione file di log ed estrazione dei DNS

2.3 DNs N-gram Tokenizer

Successivamente si passa alla suddivisione dei nomi di dominio (DNs) in sequenze di caratteri chiamate n-grammi. Quest'ultimi rappresentano sottosequenze contigue di lunghezza specificata.

fica all'interno dei nomi di dominio. Questa operazione di suddivisione è fondamentale per consentire un'analisi più approfondita dei DNs utilizzando modelli di machine learning.

L'operazione di suddivisione degli n-grammi viene eseguita mediante l'utilizzo di una funzione che riceve in input una linea di testo e genera gli n-grammi a partire da essa. Prima di creare gli n-grammi, la linea di testo viene sottoposta a un processo di pulizia in cui viene rimossa la presenza del carattere ":" e degli eventuali spazi vuoti presenti all'inizio o alla fine della linea. Successivamente, gli n-grammi vengono restituiti come una stringa in cui ciascun n-gramma è separato dagli altri mediante uno spazio. Si precisa che è possibile specificare la quantità di DNS da analizzare in questa fase semplicemente indicando la percentuale di file di dns da utilizzare. Nel caso in esame sono stati utilizzati il 25% dei file disponibili, ovvero circa 3'464'757 di nomi di domini, per consentire tempi d'analisi accettabili in Google Colab.

Vengono così generati n-grammi di tre diverse lunghezze, unigrammi (1-gram), digrammi (2-grams) e trigrammi (3-grams). Questa varietà di lunghezze permette ai modelli di apprendere differenti pattern e relazioni presenti nei DNs.

The screenshot shows three separate Google Colab notebooks running simultaneously. Each notebook has a title bar indicating its name and a code cell below it.

- unigrams.log:** Contains the command `!nltk.download('stopwords')` and the resulting output showing the download of the stopwords package.
- digrams.log:** Contains the command `!nltk.download('punkt')` and the resulting output showing the download of the punkt package.
- trigrams.log:** Contains the command `!nltk.download('averaged_perceptron_tagger')` and the resulting output showing the download of the averaged_perceptron_tagger package.

The notebooks are running on a Mac OS X system, as indicated by the window title bars.

Figura 2.3: Unigrammi (1-gram), Diagrammi (2-grams) e Trigrammi (3-grams)

2.4 Addestramento modelli FastText

Dopo aver eseguito la suddivisione degli n-grams, i dati preprocessati vengono utilizzati per addestrare modelli FastText. Quest'ultima è una libreria di machine learning che sfrutta la struttura degli n-grammi per apprendere rappresentazioni vettoriali dei DNs. Queste rappresentazioni vettoriali catturano informazioni semantiche e relazioni tra i DNs, rendendo i modelli adatti per compiti come la classificazione, lo scopo ultimo di questo elaborato.



Figura 2.4: Libreria per l'apprendimento di incorporamenti di parole

Durante l'addestramento, vengono creati tre modelli FastText: uno per i unigrammi (1-grammi), uno per i bigrammi (2-grammi) e uno per i trigrammi (3-grammi). Ciascun modello viene addestrato utilizzando il metodo "skipgram" che consente di catturare le relazioni semantiche tra le parole prevedendo il contesto circostante delle stesse. Una volta addestrati, i modelli vengono salvati in formato binario e in formato VEC, come da requisito dell'architettura della rete di classificazione che verrà discussa nel capitolo successivo. La funzione adibita al salvataggio estrae le parole e le rispettive rappresentazioni vettoriali dai modelli FastText e le salva in un file ".vec". Quest'ultimo contiene l'header con il conteggio delle parole e la dimensione dei vettori, seguito da ogni parola e il suo vettore. I modelli FastText vengono addestrati utilizzando gli n-grammi generati durante la fase precedente. I modelli in formato ".vec" vengono salvati direttamente nella directory della repository "DGA-Mixed-Embeddings-Ensemble", che è stata precedentemente caricata in Google Drive, per essere poi impiegata in seguito nella fase d'analisi.

Per l'addestramento di modelli FastText i tempi variano in base alla quantità di dati disponibili e al numero di combinazioni possibili di n-grammi. Quindi è importante considerare la

dimensione del corpus di testo e la complessità degli n-grammi scelti al fine di ottenere risultati ottimali in un tempo ragionevole. I tempi d'attesa aumentano progressivamente per i diversi tipi di n-grammi: uni-grammi < bi-grammi < tri-grammi. Di seguito sono riportati i tempi computazionali di addestramento dei tre modelli FastText:

Modello	Tempi d'addestramento
uni-grammi	~ 6m
bi-grammi	~ 2h 30m
tri-grammi	~ 3h 15m

Capitolo 3

Rete di Classificazione

Dopo aver completato la fase di preprocessing nel notebook `dga-dns-preprocessing`, si procede con l'analisi e la classificazione dei domain names malevoli utilizzando una rete di classificazione.

Nel notebook `dga-dns-analysis`, vengono importate le librerie necessarie per preparare l'ambiente colab e l'esecuzione della rete di classificazione. Vengono anche definiti i percorsi dei file e delle directory che verranno utilizzate durante l'analisi. Ad esempio, si definiscono il path degli script DGA caricati in precedenza dalla repository analoga e la directory dei report. In seguito si installa Python 3.7 insieme ai pacchetti Python richiesti utilizzando il file dei requisiti `requirements.txt`. Successivamente, si procede con l'installazione di CUDA 10.0, che è necessario per `tf-gpu 1.15`. Si copia il file di installazione CUDA da Google Drive a Colab e lo si installa tramite una serie di comandi dedicati. Infine, viene impostata la directory di lavoro al percorso degli script DGA così da lanciare comodamente il codice Python della rete.

3.1 Dataset

In questo lavoro, si è sfruttato il dataset UMUDGA (Unified Malicious Domain Generation Algorithm) per condurre lo studio di classificazione dei domini malevoli generati tramite classi di DGA. Il dataset UMUDGA è stato creato specificamente per fornire una collezione rappre-

sentativa di domini malevoli generati da vari algoritmi DGA, consentendo così di effettuare ricerche approfondite e valutare l'efficacia dei modelli di classificazione.

Il dataset utilizzato, nonché disponibile nella cartella "Dataset" del repository, è un sottosinsieme di quello originale: UMUDGA 1K. Questo subset è stato progettato per fornire una diversa dimensione di campionamento, al fine di supportare sia le analisi a piccola scala che quelle su larga scala.

UMUDGA 1K è costituito da 51 classi bilanciate, di cui 50 rappresentano diverse tipologie di algoritmi DGA, mentre la classe restante rappresenta nomi di dominio legittimi. Questo contiene 1000 record per ciascuna classe, ovvero 51'000 righe. Ogni dominio è stato anche associato all'algoritmo DGA specifico utilizzato per generarlo.

Il file `umudga_1k.csv` è strutturato in diversi campi. Sono presenti: una label binaria che indica se il dominio è generato da un algoritmo DGA o è legittimo, una label multiclass che rappresenta il tipo specifico di algoritmo DGA con cui è stato generato il dominio e i campi 1-grams, 2-grams, 3-grams. Quest'ultimi sono ottenuti dal nome di dominio senza punti e servono come rappresentazioni aggiuntive per l'analisi dei dati.

16985	dga	kraken_v_fotmyvcgmamooo	fotmyvcgmamooo	f o t m y v c m g a m o o o	f o t t m y v v c m g a g a m m o o o	f o t o t m y v v c m g a g a m a m o o o	f o t m y v c m g a m o o o
16986	dga	kraken_v_ckuociowxttcom	ckuociowxttcom	c k u o c i o w o x t t c o m	c k u o u c o c i o o w o o x t t c o o r	c k u o u c o c i o l o w o w o x t t c o o r	c k u o c i o w o x t t c o o r
16987	dga	kraken_v_ixjcbhjxexcom	ixjcbhjxex.com	i x j c b h j x e x c o m	i x j c b h j x e x c o o m	i x j c b h b j h j x e x c o o m	i x j c b h j x e x c o o m
16988	dga	kraken_v_vwmilcidyiorg	vwmilcid.yi.org	v w m i c i d y i o r g	v w m i l c i d y i o r g	v w m i l c i d i d y i o r g	v w m i c i d y i o r g
16989	dga	kraken_v_xawgfkavt	xawgfkavt	x a w g f k a t v	x a w g w g f a k t v	xaw w g g f a k a k t k v	x a w g f a k t v
16990	dga	kraken_v_pqoabbaelwntv	pqoabbaelwntv	q p o a b b a e l w n t v	q p o a b b a e e e l w n t n t v	q p o a a b b a b b a a e e e l w n w n t n t v	q p o a a b b a b b a a e e e l w n w n t n t v
16991	dga	kraken_v_dcqturnnmoocon	dcqturnnm.mooo.co	d c q t u r n n m o o o o	d c q t u r n n m n m o o o o o	d c q t q t u r n r n n n m o o o o o o	d c q t u r n n m o o o o o
16992	dga	kraken_v_fkwywkjocom	fkwywkj.com	f k w y w k o j c o m	f w k y w k y w k o o j c o m	f w k y w k y w k o j o j c o m	f w k y w k y w k o j c o m
16993	dga	kraken_v_gtmrrttsiyorg	gtmrrtts.yi.org	g t m r r t t s y i o r g	g t t m r r t t t s y i o r g	g t m r r t t t t s t s y i o r g	g t m r r t t s y i o r g
16994	dga	kraken_v_benhywchnixtv	benhywchnix.tv	b e n h y w c h n i x t v	b e n h n y w c h n h i n i x t v	b e n h n y w c h n h i n i x t v	b e n h y w c h n i x t v
16995	dga	kraken_v_lymoqmfdmbooc	lymoqmfdmbooc	i y m o q f m d b m o o o	i y m o q f m d b m o o o	i y m o q f m d b m o o o	i y m o q f m d b m o o o
16996	dga	kraken_v_ifdymptagmoooco	ifdymptag.mooo.co	i f d y m p t a g m o o o o	i f d y m p t a g m o o o o	i f d y m p t a g m o o o o	i f d y m p t a g m o o o o
16997	dga	kraken_v_jhkoofxjt	jhkoofxjt	j h k o o f x j t v	j h k o o f x j t v	j h k o o f x j t v	j h k o o f x j t v
16998	dga	kraken_v_qzquuzudjautv	qzquuzudjautv	q z q u u z u d j a u t v	q z q u u z u d j a u t v	q z q u u z u d j a u t v	q z q u u z u d j a u t v
16999	dga	kraken_v_kessonjknmoooo	kessonjkn.mooo.co	k e s s o n j k n m o o o	k x e s s s o n o n j j k n m o o o	k x e s s s o n o n j j k n m o o o	k x e s s s o n o n j j k n m o o o
17000	dga	kraken_v_nyqtchgbmoooo	nyqtchgb.mooo.co	n y q t c h g b m o o o o	n y q t c h g b m o o o o	n y q t c h g b m o o o o	n y q t c h g b m o o o o o
17001	legit	googlecom	google.com	g o o g l e . c o m	g o o g l e . c o m	g o o g l e . c o m	g o o g l e . c o m
17002	legit	youtubecom	youtube.com	y o u t u b e . c o m	y o u t u b e . c o m	y o u t u b e . c o m	y o u t u b e . c o m
17003	legit	facebookcom	facebook.com	f a c e b o o k . c o m	f a c e c e b b o o o k k c o o m	f a c e c e b b o o o k k c o o m	f a c e c e b b o o o k k c o o m
17004	legit	baiducom	baidu.com	b a i d u . c o m	b a i d u d u u c o o m	b a i d u d u u c o o m	b a i d u . c o m
17005	legit	wikipediaorg	wikipedia.org	w i k i p e d i a o r g	w i k i k i p e p d i i a o a o r g	w i k i k i p e p d i i a o a o r g	w i k i p e d i a o r g

Figura 3.1: Estratto dataset UMUDGA 1K

3.2 Architettura Stacked Generalization

L'architettura Stacking è caratterizzata da un processo di addestramento che si divide in due fasi distintive: la fase interna e la fase esterna. Durante la fase interna, i modelli individuali vengono addestrati e le loro predizioni sono impiegate per creare un "dataset" utilizzato per l'addestramento del meta-modello, che consiste in una regressione logistica. Questa procedura viene ripetuta per cinque volte. Nella fase esterna, i modelli individuali vengono addestrati usando l'insieme originale di dati. Successivamente, le predizioni ottenute da tali modelli e il meta-modello vengono combinati per generare la predizione finale dell'architettura Stacking. In altre parole, i modelli singoli effettuano le loro predizioni, che vengono poi passate al meta-modello per ottenere la predizione finale dell'intera architettura Stacking. L'architettura Stacked comprende un totale di cinque modelli: tre di essi sono ottenuti tramite l'utilizzo di FastText, mentre gli altri due sono un modello random-multiclasse e un modello ELMo-multiclasse. Di seguito il comando d'esempio utilizzato per eseguire lo script:

```
!python3 Main.py
--dataset-path ../Dataset/umudga_1k.csv
--output-path {reports_dir + "/stacking_1k_0.10"}
--model Stacking
--with-training
--epochs 20
--train-perc 0.1
```

- **dataset-path** specifica il percorso del file del dataset UMUDGA 1K.
- **output-path** specifica la cartella di destinazione in cui verranno salvati i risultati della classificazione.
- **model** specifica il tipo di modello di classificazione da utilizzare, in questo caso "Stacking".
- **with-training** indica che il modello deve essere addestrato.
- **epochs** specifica il numero di epoche per l'addestramento.
- **train-perc** specifica la percentuale dei dati di addestramento da utilizzare.

A causa dei tempi di addestramento sulla piattaforma Colab, è stato necessario sfruttare i parametri disponibili nello script Python al fine di gestire eventuali interruzioni non volontarie durante l'addestramento o situazioni impreviste. Tra questi parametri, occasionalmente si è fatto uso di `starting-ext-fold-nr`, che consente di specificare il numero del fold a partire dal quale riprendere l'addestramento. Questo è particolarmente utile quando si utilizza Colab e l'addestramento viene interrotto dopo aver completato un certo numero di fold. È possibile utilizzare questo parametro grazie alla suddivisione deterministica dei fold, che garantisce la stessa divisione anche in esecuzioni diverse. Segue lo schema dell'architettura Stacked Generalization fornita direttamente dallo sviluppatore della repository:

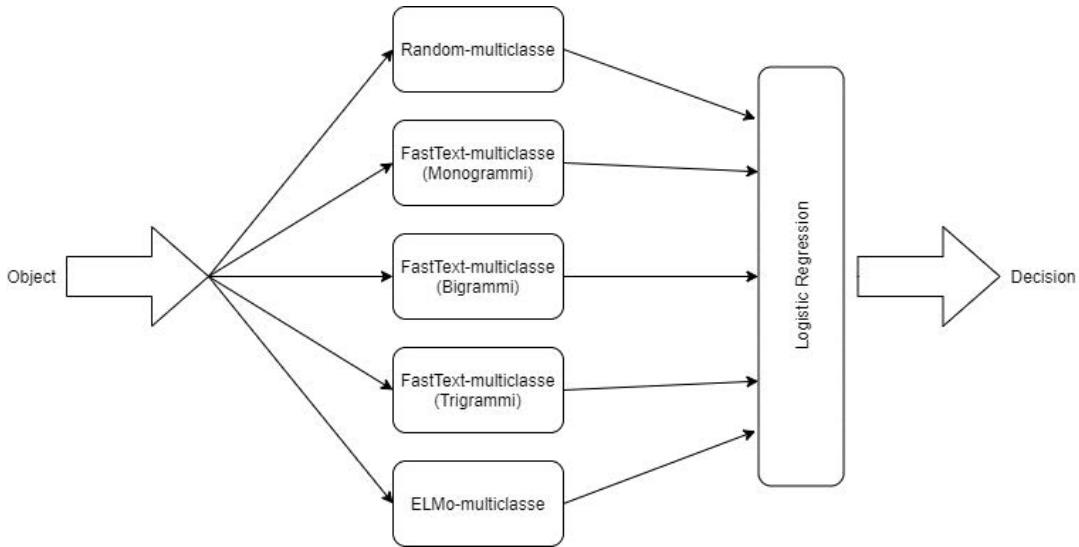


Figura 3.2: Schema architettura Stacking

3.3 Architettura REFS

Oltre all'architettura Stacked precedentemente descritta, è stata implementata un'altra architettura chiamata REFS (Random, ELMO, FastText Stacked). L'architettura REFS è composta da tre modelli: un modello random binario, un modello ELMO multiclasse e un modello FastText multiclasse. Nella fase finale dell'architettura, si prende la decisione finale considerando i risultati di questi modelli. Il modello random binario viene preso in considerazione solo se il campione viene classificato come "benevolo", mentre il modello ELMO multiclasse viene con-

siderato solo se supera una specifica soglia di attendibilità. I risultati ottenuti da questi modelli vengono quindi utilizzati per determinare la predizione finale dell'intera architettura REFS.

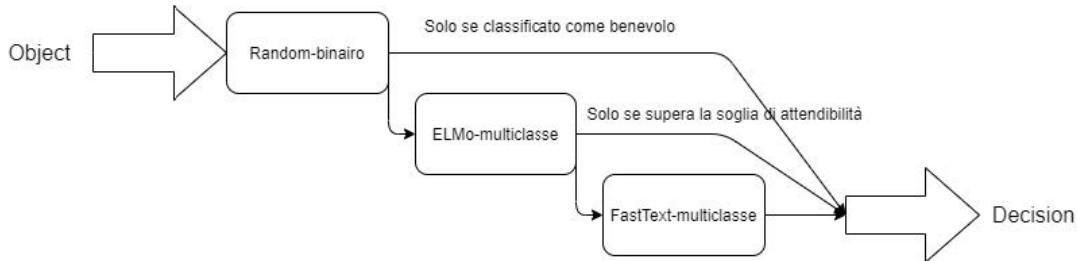


Figura 3.3: Schema architettura REFS

3.4 Architettura EFRP

L'ultima architettura, denominata EFRP, è costituita da quattro componenti: un modello FastText binario, un modello ELMO binario, un modulo Fuser Average (che non è un modello a sé stante) e un modello Random multiclass. Durante il processo di valutazione, gli output dei modelli FastText e ELMO vengono passati al modulo Fuser Average, che combina le loro predizioni. L'esito prodotto da Fuser Average viene poi preso in considerazione insieme all'output del modello Random multiclass, ma solo se il campione in questione è classificato come "benevolo". L'utilizzo di questa combinazione di modelli e la considerazione di determinati esiti sono finalizzati a fornire una decisione complessiva come nelle precedenti architetture.

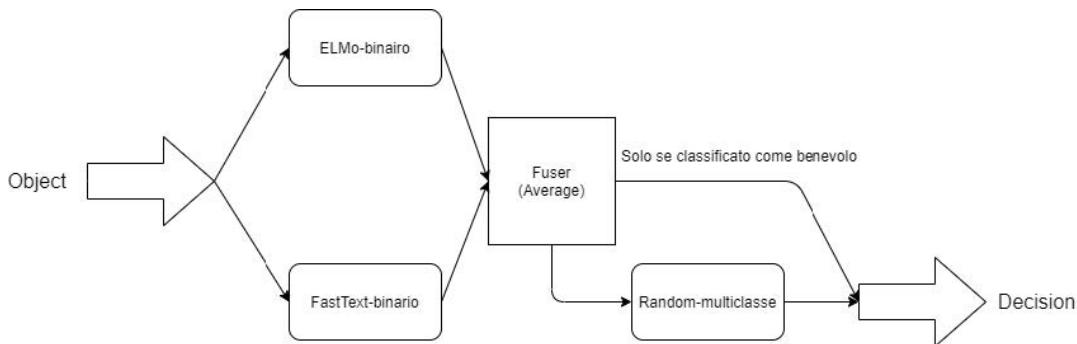


Figura 3.4: Schema architettura EFRP

Capitolo 4

Risultati

Le metriche finali di ogni architettura di rete studiata, presentate in questo capitolo, sono ottenute mediante l'aggregazione dei risultati derivanti dalle diverse fasi di test eseguite. Ogni fase di test, infatti, viene condotta per ciascuno dei 5 fold presenti nel dataset, utilizzando i campioni che non sono stati impiegati durante l'addestramento della rete. Tale procedura consente di valutare le prestazioni delle diverse architetture ottenendo così una valutazione globale delle loro capacità predittive.

I modelli sottostanti sono stati addestrati utilizzando due diversi tagli del dataset UMUDGA (vedi Tab. 4.1). Nella tabella 4.2 sono riportati i tempi di addestramento per ciascun modello eseguito.

Dataset	Tagli	
UMUDGA 1k	10%	25%

Tabella 4.1: Percentuali Dateset UMUDGA utilizzate

	Stacking (10%)	Stacking (25%)	REFS	EFRP
Tempi	~ 5h 45m	~ 9h 30m	~ 5h	~ 6h

Tabella 4.2: Tempi di addestramento

4.1 Stacked Generalization

Nella presente sezione, si pone l'accento sulla prima architettura di rete, la Stacking (Sez.3.2). Tale rete viene addestrata utilizzando entrambi i tagli disponibili del dataset. Questa selezione mira a evitare un sovraccarico eccessivo nel processo di addestramento, con l'ulteriore vantaggio di ridurre le tempistiche coinvolte.

4.1.1 Stacking UMUDGA 10%

La prima tabella 4.3 mostra le metriche ottenute testando il modello stacking addestrandolo con il 10% del dataset UMUDGA 1k, la tabella 4.4 quelle ottenute per ogni classe di algoritmi che generano domini malevoli.

	Macro	Micro
F1 Score	0.6800	0.6883
Precision	0.7093	0.6883
Recall	0.6883	0.6883
Accuratezza		0.6883

Tabella 4.3: Metriche Stacking UMUDGA 1k (10%)

Tabella 4.4: Risultati Stacking UMUDGA 1k (10%)

	Precision	Recall	F1 Score	Support
alureon	0.259729	0.388842	0.295260	950.00000
banjori	1.000000	1.000000	1.000000	950.00000
bedep	0.532584	0.610316	0.555645	950.00000
ccleaner	0.992301	0.997684	0.994973	950.00000
chinad	0.929879	0.882947	0.904204	950.00000
corebot	0.995803	0.988000	0.991875	950.00000
cryptolocker	0.402649	0.291368	0.326767	950.00000
Continua nella pagina successiva				

Tabella 4.4

	Precision	Recall	F1 Score	Support
dircrypt	0.260482	0.234737	0.2225807	950.00000
dyre	0.996874	0.999368	0.998113	950.00000
fobber_v1	0.759076	0.968421	0.850461	950.00000
fobber_v2	0.284435	0.469263	0.339214	950.00000
gozi_gpl	0.928345	0.904632	0.908909	950.00000
gozi_luther	0.854838	0.712421	0.730823	950.00000
gozi_nasa	0.634261	0.701263	0.654869	950.00000
gozi_rfc4343	0.642710	0.689263	0.663246	950.00000
kraken_v1	0.838471	0.601263	0.686959	950.00000
kraken_v2	0.423418	0.384421	0.401974	950.00000
locky	0.371157	0.198105	0.257281	950.00000
matsnu	0.941057	0.888000	0.909880	950.00000
murofet_v1	0.983196	0.963579	0.972988	950.00000
murofet_v2	0.795215	0.900842	0.837498	950.00000
murofet_v3	1.000000	0.992211	0.996089	950.00000
necurs	0.400380	0.128842	0.175289	950.00000
nymaim	0.914850	0.793263	0.844448	950.00000
padcrypt	0.979382	0.977474	0.978400	950.00000
pizd	0.646611	0.734737	0.668252	950.00000
proslikefan	0.394556	0.553895	0.453876	950.00000
pushdo	0.846146	0.779789	0.775772	950.00000
pykspa	0.240788	0.174947	0.191320	950.00000
pykspa_noise	0.304122	0.188211	0.211286	950.00000
qadars	0.817109	0.833895	0.822195	950.00000
qakbot	0.597667	0.376632	0.428724	950.00000
ramdo	0.960768	0.992421	0.976225	950.00000
ramnit	0.207973	0.227368	0.213083	950.00000

Continua nella pagina successiva

Tabella 4.4

	Precision	Recall	F1 Score	Support
ranbyus_v1	0.486733	0.574105	0.488685	950.00000
ranbyus_v2	0.634767	0.616421	0.616174	950.00000
rovnix	0.880186	0.844000	0.853733	950.00000
shiotob	0.745531	0.613474	0.668117	950.00000
simda	0.789821	0.657684	0.634347	950.00000
sisron	0.997676	0.850316	0.886230	950.00000
suppobox_1	0.745146	0.627158	0.599096	950.00000
suppobox_2	0.890784	0.886737	0.887121	950.00000
suppobox_3	0.981433	0.974105	0.977367	950.00000
symmi	0.989386	0.998947	0.994138	950.00000
tempedreve	0.395799	0.440000	0.408563	950.00000
tinba	0.330134	0.304211	0.276606	950.00000
vawtrak_v1	0.990433	0.962526	0.974537	950.00000
vawtrak_v2	0.804194	0.711579	0.747065	950.00000
vawtrak_v3	0.706821	0.860211	0.773334	950.00000
zeus-newgoz	0.999366	0.985263	0.992201	950.00000
legit	0.671477	0.669684	0.664968	950.00000

A seguire viene presentata la matrice di confusione che descrive le correlazioni tra i risultati previsti dal processo di classificazione, evidenziando sia Falsi/Veri positivi che Falsi/Veri negativi.

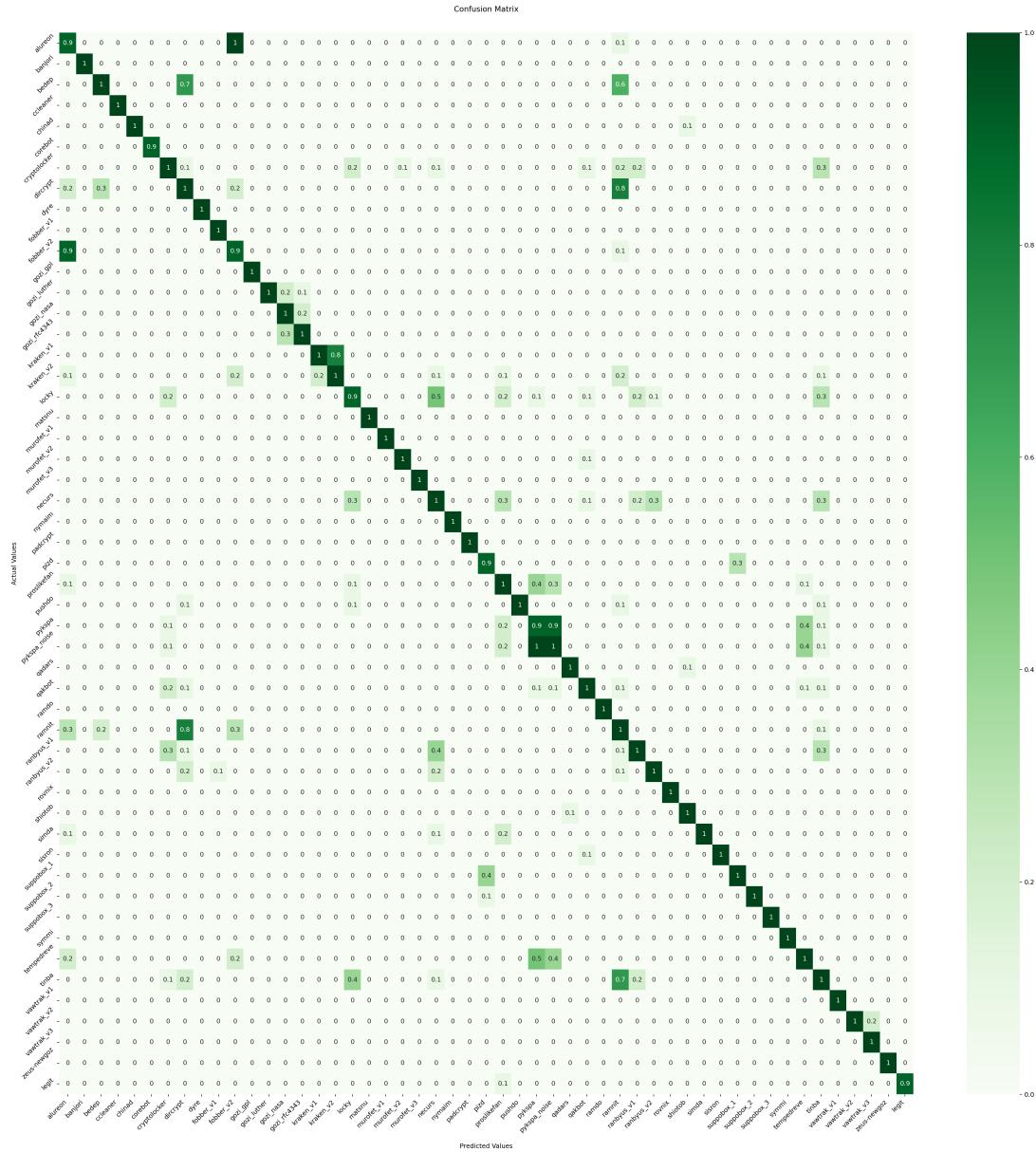


Figura 4.1: Confusion matrix modello Stacking UMUDGA 1K (10%)

4.1.2 Stacking UMUDGA 25%

Al fine di evitare ridondanze descrittive, a partire da questo momento, i risultati, ottenuti da ogni architettura di rete, saranno presentati seguendo la seguente sequenza:

- Tabella metriche del test complessivo
- Tabella metriche per ogni classe
- Matrice di confusione

	Macro	Micro
F1 Score	0.7742	0.7755
Precision	0.7894	0.7755
Recall	0.7755	0.7755
Accuratezza		0.7755

Tabella 4.5: Metriche stacking UMUDGA 1K (25%)

Tabella 4.6: Risultati Stacking UMUDGA 1k (25%)

	Precision	Recall	F1 Score	Support
alureon	0.373149	0.414905	0.382356	4750.00000
banjori	0.982276	0.863705	0.887494	4750.00000
bedep	0.682503	0.620505	0.645338	4750.00000
ccleaner	0.998233	0.997053	0.997634	4750.00000
chinad	0.969813	0.970147	0.969950	4750.00000
corebot	0.999576	0.992674	0.996113	4750.00000
cryptolocker	0.536311	0.463242	0.494873	4750.00000
dircrypt	0.370183	0.254526	0.297525	4750.00000
dyre	0.999495	0.999242	0.999368	4750.00000
fobber_v1	0.855221	0.935411	0.893055	4750.00000
Continua nella pagina successiva				

Tabella 4.6

	Precision	Recall	F1 Score	Support
fobber_v2	0.373295	0.549011	0.435817	4750.00000
gozi_gpl	0.953295	0.954863	0.953316	4750.00000
gozi_luther	0.859641	0.891116	0.874504	4750.00000
gozi_nasa	0.797946	0.819242	0.807795	4750.00000
gozi_rfc4343	0.782426	0.764421	0.772642	4750.00000
kraken_v1	0.768096	0.734442	0.709013	4750.00000
kraken_v2	0.597751	0.382947	0.464551	4750.00000
locky	0.636192	0.506358	0.562861	4750.00000
matsnu	0.939685	0.957558	0.948276	4750.00000
murofet_v1	0.991105	0.985895	0.988469	4750.00000
murofet_v2	0.868317	0.943200	0.903977	4750.00000
murofet_v3	1.000000	0.995874	0.997932	4750.00000
necurs	0.739419	0.559158	0.629000	4750.00000
nymaim	0.913997	0.872884	0.892863	4750.00000
padcrypt	0.990736	0.988211	0.989360	4750.00000
pizd	0.822255	0.867368	0.837247	4750.00000
proslikefan	0.573459	0.573011	0.566240	4750.00000
pushdo	0.920524	0.934989	0.926722	4750.00000
pykspa	0.302071	0.343874	0.320267	4750.00000
pykspa_noise	0.282801	0.267453	0.274248	4750.00000
qadars	0.933309	0.903747	0.917846	4750.00000
qakbot	0.736276	0.464758	0.557409	4750.00000
ramdo	0.988195	0.956000	0.969380	4750.00000
ramnit	0.316279	0.418863	0.351881	4750.00000
ranbyus_v1	0.736574	0.806526	0.766285	4750.00000
ranbyus_v2	0.737398	0.781895	0.756322	4750.00000
rovnix	0.942714	0.912505	0.926947	4750.00000

Continua nella pagina successiva

Tabella 4.6

	Precision	Recall	F1 Score	Support
shiotob	0.884017	0.815242	0.848036	4750.00000
simda	0.886449	0.863074	0.845713	4750.00000
sisron	0.996405	0.941389	0.963880	4750.00000
suppobox_1	0.831449	0.805011	0.809342	4750.00000
suppobox_2	0.940561	0.917600	0.925866	4750.00000
suppobox_3	0.986816	0.992547	0.989609	4750.00000
symmi	0.993558	0.999916	0.996727	4750.00000
tempedreve	0.415974	0.400674	0.404753	4750.00000
tinba	0.566902	0.754947	0.644025	4750.00000
vawtrak_v1	0.992921	0.886695	0.917677	4750.00000
vawtrak_v2	0.889698	0.836463	0.840326	4750.00000
vawtrak_v3	0.873718	0.940211	0.903771	4750.00000
zeus-newgoz	0.998198	0.998189	0.998191	4750.00000
legit	0.735636	0.751495	0.732908	4750.00000

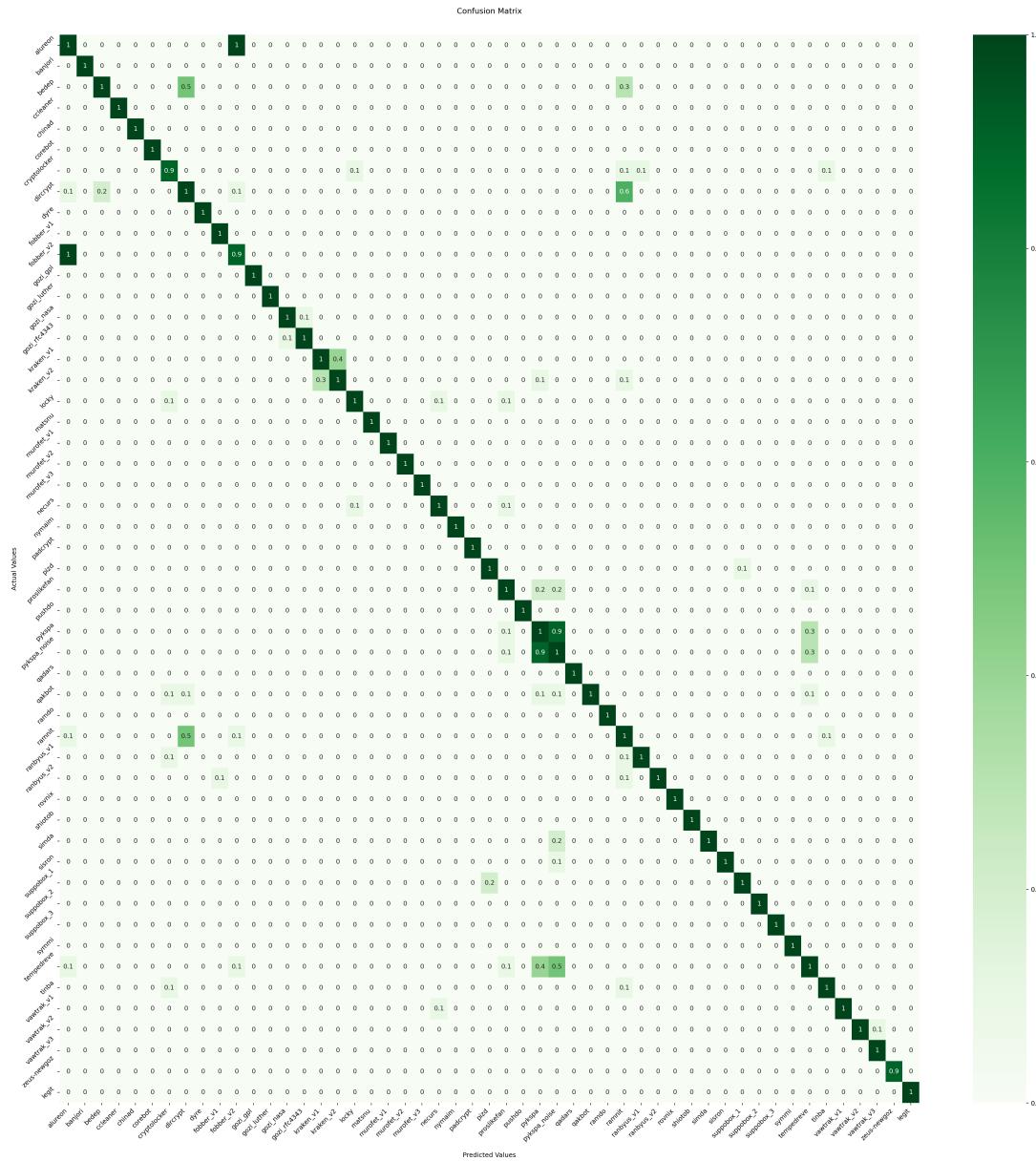


Figura 4.2: Confusion matrix modello Stacking UMUDGA 1K (25%)

4.2 REFS

Il seguente modello è stato addestrato utilizzando il taglio da 10% del dataset UMUDGA 1K.

	Macro	Micro
F1 Score	0.7126	0.7209
Precision	0.7214	0.7209
Recall	0.7209	0.7209
Accuratezza		0.7209

Tabella 4.7: Metriche REFS UMUDGA 1k (10%)

Tabella 4.8: Risultati REFS UMUDGA 1k (10%)

	Precision	Recall	F1 Score	Support
alureon	0.283	0.423	0.327	900
banjori	0.999	1.000	0.999	900
bedep	0.500	0.587	0.531	900
ccleaner	0.986	0.997	0.992	900
chinad	0.849	0.835	0.841	900
corebot	0.996	0.981	0.988	900
cryptolocker	0.375	0.427	0.391	900
dircrypt	0.317	0.271	0.274	900
dyre	0.996	0.997	0.996	900
fobber_v1	0.755	0.919	0.827	900
fobber_v2	0.275	0.292	0.275	900
gozi_gpl	0.904	0.959	0.930	900
gozi_luther	0.799	0.852	0.823	900
gozi_nasa	0.640	0.755	0.692	900
gozi_rfc4343	0.762	0.547	0.632	900
Continua nella pagina successiva				

Tabella 4.8

	Precision	Recall	F1 Score	Support
kraken_v1	0.763	0.688	0.720	900
kraken_v2	0.489	0.488	0.482	900
locky	0.497	0.375	0.421	900
matsnu	0.870	0.952	0.909	900
murofet_v1	0.961	0.990	0.975	900
murofet_v2	0.761	0.974	0.854	900
murofet_v3	0.998	0.992	0.995	900
necurs	0.634	0.255	0.352	900
nymaim	0.857	0.844	0.850	900
padcrypt	0.932	0.964	0.948	900
pizd	0.762	0.776	0.766	900
proslikefan	0.536	0.475	0.498	900
pushdo	0.868	0.891	0.876	900
pykspa	0.276	0.212	0.231	900
pykspa_noise	0.296	0.249	0.257	900
qadars	0.804	0.850	0.824	900
qakbot	0.659	0.336	0.439	900
ramdo	0.891	0.992	0.938	900
ramnit	0.216	0.172	0.183	900
ranbyus_v1	0.587	0.599	0.591	900
ranbyus_v2	0.611	0.626	0.614	900
rovnix	0.873	0.841	0.853	900
shiotob	0.696	0.612	0.649	900
simda	0.817	0.856	0.834	900
sisron	0.983	1.000	0.992	900
suppobox_1	0.802	0.805	0.800	900
suppobox_2	0.842	0.955	0.894	900

Continua nella pagina successiva

Tabella 4.8

	Precision	Recall	F1 Score	Support
suppobox_3	0.934	0.987	0.959	900
symmi	0.982	0.998	0.990	900
tempedreve	0.419	0.455	0.428	900
tinba	0.433	0.370	0.389	900
vawtrak_v1	0.966	0.999	0.982	900
vawtrak_v2	0.881	0.886	0.881	900
vawtrak_v3	0.824	0.866	0.842	900
zeus-newgoz	0.980	0.979	0.979	900
legit	0.658	0.616	0.634	900

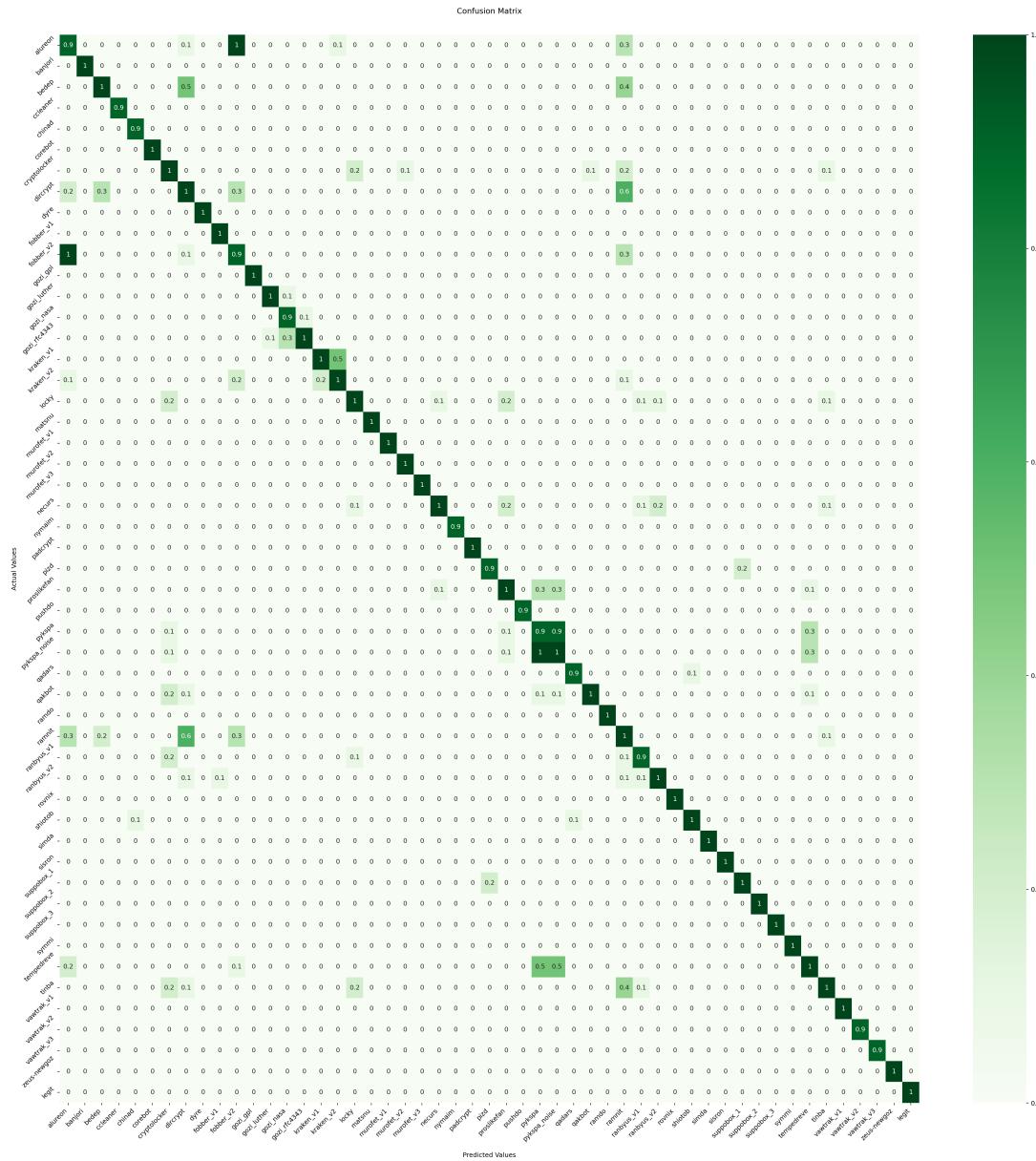


Figura 4.3: Confusion matrix modello REFS UMUDGA 1K (10%)

4.3 EFRP

Il seguente modello è stato addestrato utilizzando il taglio da 10% del dataset UMUDGA 1K. Inoltre, non ha completato il suo addestramento a causa di un possibile errore interno del codice. Di conseguenza non si dispone del risultato aggregato ottenuto dai diversi fold, ma onde evitare di vanificare il tempo impiegato, si è scelto comunque di includerlo tra i risultati considerando il fold migliore tra quelli disponibili.

	Macro	Micro
F1 Score	0.6611	0.6698
Precision	0.6805	0.6698
Recall	0.6698	0.6698
Accuratezza		0.6698

Tabella 4.9: Metriche EFRP UMUDGA 1k (10%)

Tabella 4.10: Risultati EFRP UMUDGA 1k (10%)

	Precision	Recall	F1 Score	Support
alureon	0.1399	0.0967	0.1143	900
banjori	0.9956	1.0000	0.9978	900
bedep	0.5907	0.5900	0.5903	900
ccleaner	0.9719	1.0000	0.9858	900
chinad	0.8909	0.9622	0.9252	900
corebot	0.9673	0.9867	0.9769	900
cryptolocker	0.3394	0.4989	0.4040	900
dircrypt	0.4045	0.2000	0.2677	900
dyre	1.0000	1.0000	1.0000	900
fobber_v1	0.7733	0.9856	0.8666	900
fobber_v2	0.3313	0.7967	0.4680	900
Continua nella pagina successiva				

Tabella 4.10

	Precision	Recall	F1 Score	Support
gozi_gpl	0.8513	0.7189	0.7795	900
gozi_luther	0.6579	0.5811	0.6171	900
gozi_nasa	0.3992	0.3167	0.3532	900
gozi_rfc4343	0.4518	0.4478	0.4498	900
kraken_v1	0.9382	0.4556	0.6133	900
kraken_v2	0.3622	0.4933	0.4177	900
locky	0.3441	0.1778	0.2344	900
matsnu	0.9319	0.8667	0.8981	900
murofet_v1	0.9587	0.9811	0.9698	900
murofet_v2	0.7788	0.9778	0.8670	900
murofet_v3	1.0000	0.9956	0.9978	900
necurs	0.4711	0.1989	0.2797	900
nymaim	0.8802	0.6122	0.7221	900
padcrypt	0.9692	0.9800	0.9746	900
pizd	0.5546	0.6211	0.5860	900
proslikefan	0.5868	0.4844	0.5307	900
pushdo	0.6335	0.9489	0.7598	900
pykspa	0.2534	0.1444	0.1840	900
pykspa_noise	0.3480	0.3067	0.3260	900
qadars	0.9441	0.7500	0.8359	900
qakbot	0.8729	0.3511	0.5008	900
ramdo	0.9803	0.9944	0.9873	900
ramnit	0.3112	0.3644	0.3357	900
ranbyus_v1	0.5447	0.6767	0.6036	900
ranbyus_v2	0.7586	0.6878	0.7214	900
rovnix	0.4975	0.7589	0.6010	900
shiotob	0.7563	0.7000	0.7271	900

Continua nella pagina successiva

Tabella 4.10

	Precision	Recall	F1 Score	Support
simda	0.6431	0.7467	0.6910	900
sisron	0.9202	0.9989	0.9579	900
suppobox_1	0.6325	0.4933	0.5543	900
suppobox_2	0.7410	0.8967	0.8115	900
suppobox_3	0.8353	0.9411	0.8851	900
symmi	0.9248	0.9978	0.9599	900
tempedreve	0.5210	0.4556	0.4861	900
tinba	0.4305	0.4989	0.4622	900
vawtrak_v1	0.9317	1.0000	0.9646	900
vawtrak_v2	0.4179	0.4356	0.4266	900
vawtrak_v3	0.4261	0.5833	0.4925	900
zeus-newgoz	0.9989	0.9789	0.9888	900
legit	0.8465	0.4289	0.5693	900

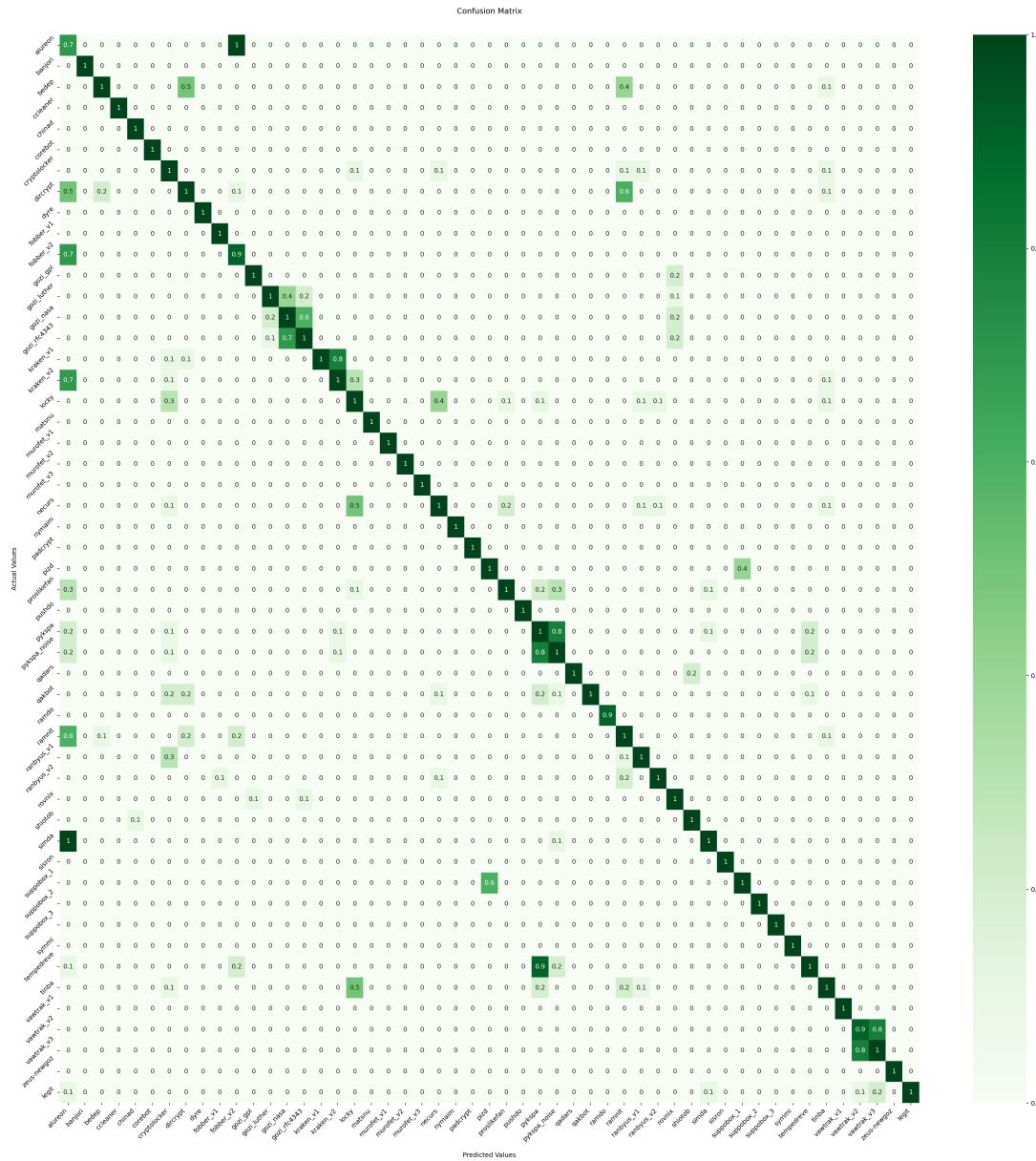


Figura 4.4: Confusion matrix modello EFRP UMUDGA 1K (10%)

Capitolo 5

Conclusioni

Per concludere, analizzando la tabella 5.1, in prima battuta si evince che i risultati migliori sono ottenuti dal modello Stacking allenato sul 25% del dataset UMUDGA. Quello che emerge però è che il modello REFS, addestrato utilizzando solo il 10% del dataset UMUDGA, è più performante di quello Stacked se rapportato alla quantità di dati con cui è stato addestrato. Nonostante la sua architettura più leggera, il modello REFS presenta prestazioni comparabili al modello Stacking, che è stato addestrato utilizzando una percentuale maggiore di dati, precisamente il 15% in più. Questa differenza di 5 punti percentuali tra i due modelli evidenzia l'efficacia del modello REFS nel migliorare sia le metriche di valutazione che i tempi di esecuzione. Ciò suggerisce che quest'ultimo modello potrebbe essere una scelta più conveniente in termini di risorse computazionali e tempo di addestramento senza compromettere la qualità delle prestazioni.

	Accuratezza	F1 Score	Precision	Recall
Stacking (10%)	0.6883	0.6800	0.7093	0.6883
Stacking (25%)	0.7755	0.7742	0.7894	0.7755
REFS	0.7209	0.7126	0.7214	0.7209
EFRP	0.6698	0.6611	0.6805	0.6698

Tabella 5.1: Confronto dei risultati ottenuti dai vari modelli