



UNIVERSITÀ DEGLI STUDI DI SALERNO

Università degli Studi di Salerno

Dipartimento di Informatica

Tecniche di Machine Learning per Intrusion Detection Systems

Analisi Comparativa su Dataset CIC-IDS-2017

Candidato

Emanuele Pascale

Matricola: NF22700002

Anno Accademico 2025–2026

Sommario

Gli Intrusion Detection Systems (IDS) sono oggi componenti centrali, fondamentali nelle infrastrutture di rete, ma la loro efficacia dipende dalla capacità di adattarsi a minacce che evolvono rapidamente e in maniera costante. I sistemi basati su regole statiche faticano a intercettare varianti nuove di attacchi noti, il che ha spinto la ricerca verso approcci di Machine Learning in grado di apprendere pattern anomali dai dati.

Questa relazione presenta un'analisi comparativa di tre algoritmi di apprendimento automatico — **Random Forest**, **LightGBM** e **Multilayer Perceptron (MLP)** — applicati al dataset CIC-IDS-2017, uno standard di riferimento per la valutazione di sistemi IDS. Il dataset originale è distribuito in **8 file CSV** giornalieri per un totale di circa **2.83 milioni** di flussi di rete e **79 colonne**; dopo il processo di analisi, pulizia e **TCP appendix filtering** con threshold ≥ 3 pacchetti, il dataset utilizzato contiene circa **1.73 milioni** di flussi etichettati, suddivisi in traffico benigno e sette tipologie di attacco: Bot, BruteForce, DDoS, DoS, Other, PortScan e WebAttack. Il dataset presenta un **class imbalance estremo** (ratio 31,077:1 tra Benign e Other).

La metodologia si sviluppa attraverso quattro fasi: (1) *Exploratory Data Analysis* per identificare distribuzioni, correlazioni e sbilanciamento delle classi; (2) *Feature Engineering e Selection* mediante Spearman Correlation, Mutual Information, ANOVA F-statistic e Random Forest Feature Importance, combinate in un consensus ranking che riduce le feature da 46 (dopo preprocessing) a 20; (3) gestione del class imbalance tramite **SMOTE gap-filling** al **70%** incrementando i samples da 1.21M a 1.27M ($+ \sim 5.2\%$); (4) training e valutazione dei modelli con split stratificato **70/15/15** in train, validation e test, per non introdurre informazioni future nel processo di apprendimento e dunque evitando data leakage.

I risultati sperimentali mostrano che il modello **Random Forest con SMOTE** rappresenta il compromesso migliore tra accuratezza globale e capacità di gestire lo sbilanciamento: sul test set raggiunge circa **99,85%** di accuracy e **90,43%** di F1-score macro, con un **miglioramento del +0.59%** rispetto al baseline RF (da 91.18% a 91.77% su validation) e del **0.54%** su test set (0.8989 \rightarrow 0.9043). L'analisi per-class rivela eccellente bilanciamento precision-recall su classi maggioritarie ($F1 \geq 0.99$) e performance robuste su classi minoritarie critiche (Bot 0.77, WebAttack 0.72, Other 0.77). Il gap fra validation e test del 1.34% conferma l'ottima generalizzazione. LightGBM, pur mantenendo un'accuracy del 77.46%, presenta un F1-score macro significativamente più basso (26.65%), mentre MLP ottiene performance intermedie (99.44% accuracy e 69.57% F1-macro baseline, migliorando a 99.48% accuracy e 76.91% F1-macro con SMOTE).

Dal punto di vista metodologico, questa relazione si distingue per tre aspetti principali: un preprocessing documentato passo-passo del dataset, un confronto su due configurazioni di modello anziché sul solo baseline, e una verifica esplicita della generalizzazione tramite gap validation-test.

Indice

1	Introduzione	5
1.1	Contesto e Motivazione	5
1.2	Dataset CIC-IDS-2017	6
1.2.1	Data Cleaning e Preprocessing	6
1.2.2	Distribuzione delle Classi	6
1.3	Obiettivi della Ricerca	7
1.4	Contributi Originali	8
1.5	Struttura della Relazione	8
2	Stato dell'Arte	9
2.1	Algoritmi di Machine Learning per IDS	9
2.1.1	Random Forest e Ensemble Methods	9
2.1.2	Gradient Boosting Machines	9
2.1.3	Deep Learning	9
2.2	Problematiche Metodologiche in Letteratura	10
2.2.1	Data Leakage Topologico	10
2.2.2	Class Imbalance e Metriche Inadeguate	10
2.2.3	Artefatti nel Dataset	10
2.3	Confronto Quantitativo con lo Stato dell'Arte	11
2.4	Contributi di Questa Ricerca	11
3	Caratterizzazione del Problema	11
3.1	Dataset CIC-IDS-2017 e Preprocessing	12
3.2	Distribuzione delle Classi nel Training Set	12
3.3	Feature Engineering	13
3.3.1	PacketRatio	13
3.3.2	ByteRatio	14
3.3.3	Validazione Empirica	14
3.4	Pipeline Anti-Leakage	14
3.5	Protocollo di Valutazione	15
3.6	Metriche di Valutazione	15
3.6.1	F1-Macro Score (Metrica Primaria)	15
3.6.2	Precision e Recall per-class	15
3.6.3	Metriche Secondarie	16
4	Metodologia	16
4.1	Overview della Pipeline	16
4.2	Fase 1: Exploratory Data Analysis	17
4.2.1	Caricamento e Aggregazione Dataset	17
4.2.2	Data Cleaning	17
4.2.3	Label Engineering: 15 → 8 Classi	18
4.3	Fase 2: Feature Engineering & Selection	19
4.3.1	Sanitizzazione Anti-Bias e Ratio Features	19
4.3.2	Consensus Feature Selection	20

4.3.3	Rimozione Ridondanza	21
4.4	Fase 3: Model Training & Evaluation	21
4.4.1	Data Splitting Stratificato	21
4.4.2	Gestione Class Imbalance: Baseline vs SMOTE	22
4.4.3	Cross-Validation Protocol	23
4.4.4	Algoritmi Selezionati	23
4.4.5	Model Selection Strategy	25
4.5	Fase 4: Test Finale	26
5	Implementazione	26
5.1	Fase 1: Caricamento Dati e Analisi Esplorativa	26
5.1.1	Aggregazione Multi-File e Ottimizzazione Memoria	27
5.1.2	Pipeline di Cleaning in 5 Step	27
5.1.3	Label Engineering e Encoding	28
5.1.4	Distribuzione delle Classi	29
5.1.5	Analisi Univariata delle Distribuzioni	30
5.1.6	Analisi Bivariata: Correlazione Feature-Target	31
5.1.7	Identificazione Feature Problematiche	33
5.2	Fase 2: Feature Engineering e Selection	35
5.2.1	Sanitizzazione Anti-Bias	35
5.2.2	Creazione Ratio Features	36
5.2.3	Split Stratificato 70/15/15	36
5.2.4	Consensus Feature Selection: Implementazione	37
5.2.5	Applicazione Trasformazioni a Val/Test (Anti-Leakage Policy)	40
5.3	Fase 3: Training e Valutazione Modelli	41
5.3.1	Configurazione Modelli: Doppia Strategia Esplorativa	41
5.3.2	SMOTE Gap-Filling Strategy: Implementazione Progressiva .	46
5.3.3	Data Scaling per MLP	47
5.3.4	Cross-Validation Rigorosa con SMOTE Inside Loop	48
5.3.5	Final Training e Test Set Evaluation	48
6	Risultati Sperimentali	49
6.1	Risultati Cross-Validation (5-Fold Stratified)	49
6.2	Risultati Validation Set	51
6.2.1	Panoramica Performance	51
6.2.2	Random Forest — Baseline	52
6.2.3	Random Forest + SMOTE (Best Model)	54
6.2.4	LightGBM — Baseline	56
6.2.5	LightGBM + SMOTE	58
6.2.6	Multilayer Perceptron (Rete Neurale) — Baseline	60
6.2.7	Multilayer Perceptron + SMOTE	62
6.3	Test Finale ed elezione del Best Model	64
6.3.1	Metriche Per-Class su Test Set	66
6.3.2	Analisi Errori	67
6.4	Confronto con Stato dell'Arte	68

7	Discussione	69
7.1	Perché la superiorità di Random Forest	69
7.1.1	Perché LightGBM Fallisce	70
7.1.2	L'Effetto di SMOTE Varia tra gli Algoritmi	70
7.2	I Limiti dell'Approccio Basato su Flussi di Rete	71
7.2.1	WebAttack: Difficoltà a distinguere con Benign	71
7.2.2	Bot: Difficoltà ad individuare la classe	72
7.2.3	Efficacia IDS per classe:	72
7.3	Confronto con Altri Studi	72
7.3.1	Feature Topologiche	73
7.3.2	Scelta delle Metriche di Valutazione	73
7.3.3	Applicazione Corretta di SMOTE	73
7.3.4	Split Train/Validation/Test	73
7.4	Applicabilità in Ambiente Reale	74
7.4.1	Costo degli errori	74
7.4.2	Architettura a Più Livelli	74
8	Conclusioni e Sviluppi Futuri	75
8.1	Riepilogo dei risultati	75
8.2	Limiti del Lavoro	75
8.3	Sviluppi Futuri	76
Riferimenti Bibliografici		77

1 Introduzione

1.1 Contesto e Motivazione

La sicurezza informatica rappresenta una delle sfide attuali più critiche nell'ambito tecnologico. Secondo il Cybersecurity Ventures Report 2025 [?], i danni globali derivanti dal cybercrime hanno raggiunto i 10.5 trilioni di dollari annui, con una crescita del 15% annuo fino a 12 trilioni nel 2031. Con la crescente digitalizzazione dei servizi e l'espansione dell'Internet of Things (IoT), le superfici d'attacco si ampliano esponenzialmente, rendendo le infrastrutture critiche, le aziende e gli utenti vulnerabili a minacce sempre più sofisticate.

Gli **Intrusion Detection Systems (IDS)** costituiscono una linea di difesa fondamentale per identificare e prevenire attacchi informatici in tempo reale. Tradizionalmente, gli IDS si basano su due approcci principali:

- **Signature-based:** Efficaci contro minacce note, identificano attacchi confrontando il traffico con pattern predefiniti. Tuttavia, risultano inefficaci per attacchi zero-day o varianti di attacchi esistenti.
- **Anomaly-based:** Più flessibili, costruiscono un modello del comportamento “normale” della rete e segnalano deviazioni significative. Questo approccio sfrutta il Machine Learning per distinguere anomalie benigne da attacchi reali, ma richiede calibrazione accurata per gestire il trade-off tra detection rate e falsi positivi.

L'applicazione del ML agli IDS ha mostrato risultati promettenti, ma persistono sfide critiche:

- **Class Imbalance:** Il traffico di rete è prevalentemente benigno, con attacchi che rappresentano meno del 20% dei flussi. Nel dataset CIC-IDS-2017, alcune classi critiche come Infiltration e Heartbleed sono presenti in meno dello 0.01% dei campioni, come documentato nell'analisi esplorativa.
- **High Dimensionality:** I dataset IDS contengono decine di feature estratte da pacchetti di rete, molte delle quali ridondanti o irrilevanti, richiedendo tecniche rigorose di feature selection.
- **Data-Induced Leakage:** Caratteristiche specifiche dell'ambiente di cattura, incluse informazioni legate alla configurazione di rete, possono introdurre bias che compromettono la capacità di generalizzazione dei modelli su reti diverse [7].
- **Real-time Constraints:** Gli IDS devono operare con latenze inferiori ai millisecondi per non compromettere le prestazioni della rete.

1.2 Dataset CIC-IDS-2017

Il dataset CIC-IDS-2017 rappresenta uno degli standard più utilizzati per la valutazione di IDS basati su ML. Generato dal Canadian Institute for Cybersecurity nel 2017, è ampiamente adottato dalla comunità scientifica per la sua completezza e qualità [8].

Caratteristiche principali:

- **Realismo:** Traffico di rete reale catturato durante 5 giorni lavorativi (3-7 luglio 2017) in un ambiente di laboratorio controllato, includendo sia attività utente legittime che attacchi informatici moderni. CICFlowMeter è stato utilizzato come tool di estrazione delle feature da pacchetti di rete effettivamente trasmessi.
- **Diversità degli attacchi:** Include 8 categorie di attacco per classificazione multiclass: DoS (Denial of Service), DDoS (Distributed DoS), Brute Force, Web Attacks (SQL Injection, XSS), Botnet, PortScan, e attacchi complessi ma rari (Infiltration, Heartbleed).
- **Feature estratte:** 79 feature base estratte tramite CICFlowMeter, basate su statistiche temporali (durate flusso, inter-arrival times), dimensioni pacchetti, flag TCP/IP, e parametri bidirezionali (forward/backward).
- **Dimensione:** 2.830.743 flussi originali, ridotti a 1.729.294 dopo data cleaning.

1.2.1 Data Cleaning e Preprocessing

Prima dell'analisi, il dataset è stato sottoposto a una pipeline di pulizia rigorosa

- Rimozione di 2.867 flussi con valori NaN o infiniti (errori di calcolo di CICFlowMeter)
- Eliminazione di 329.691 flussi duplicati
- Rimozione di 107 flussi con valori negativi non validi in feature temporali
- Eliminazione di 768.784 flussi con meno di 3 pacchetti (TCP appendix artifacts) [4]

1.2.2 Distribuzione delle Classi

Dopo il label engineering, il dataset presenta 8 categorie per classificazione multiclass con uno sbilanciamento critico:

Tabella 1: Distribuzione delle classi nel dataset CIC-IDS-2017 dopo preprocessing

Classe	Istanze	Percentuale
BENIGN	1,398,479	80.87%
DoS	189,262	10.94%
DDoS	127,928	7.40%
BruteForce	8,866	0.51%
WebAttack	2,007	0.12%
PortScan	1,479	0.09%
Bot	1,228	0.07%
Other	45	0.003%
Totale	1,729,294	100%

Come evidenziato nella Tabella 1, il dataset presenta un Imbalance Ratio di 31.077:1 tra la classe maggioritaria (BENIGN) e minoritaria (Other). Questo rispecchia la realtà operativa degli IDS ma pone una sfida per quanto riguarda l’addestramento dei modelli di ML.

1.3 Obiettivi della Ricerca

Questo lavoro si propone di:

- Condurre un’analisi esplorativa sistematica del dataset, identificando feature rilevanti e problematiche metodologiche (data leakage, multicollinearità, low-variance features).
- Implementare e confrontare tre algoritmi ML rappresentativi di approcci differenti:
 - Random Forest:** Ensemble method basato su alberi decisionali, robusto al rumore e interpretabile tramite feature importance [1].
 - LightGBM:** Gradient boosting ottimizzato per efficienza computazionale su dataset massivi [6].
 - Multilayer Perceptron:** Rete neurale feedforward per catturare relazioni non lineari complesse.
- Applicare **Feature Engineering** introducendo ratio features (*PacketRatio*, *ByteRatio*) per catturare asimmetrie nel traffico bidirezionale. Queste metriche misurano lo sbilanciamento tra pacchetti/byte in direzione forward (client→server) e backward (server→client), utili per identificare in particolar modo gli attacchi volumetrici (DDoS, DoS) e gli attacchi con traffico asimmetrico (Port Scan).
- Implementare **Consensus Feature Selection** combinando 4 metodi complementari (Spearman Correlation, Mutual Information, ANOVA F-test, Random Forest Importance) per ridurre la dimensionalità da 79 a 20 feature (74.7% di riduzione), migliorando efficienza computazionale e capacità di generalizzazione.

5. Valutare l'impatto di **SMOTE** (Synthetic Minority Over-sampling Technique) [3] sul rilevamento di classi minoritarie, confrontando configurazioni baseline vs configurazioni con oversampling sintetico.
6. Analizzare le performance dei modelli attraverso metriche robuste (F1-macro, precision/recall per-class, weighted F1) e identificare punti critici nel rilevamento delle diverse tipologie di attacco.

1.4 Contributi Originali

Questo studio punta a distinguersi dagli studi precedenti su CIC-IDS-2017 per diversi aspetti metodologici:

Primo, viene implementato un approccio di Consensus Feature Selection che aggredisce i ranking di 4 metodi complementari per identificare le 20 feature più robuste, riducendo il bias che affligge i metodi singoli. Questo approccio multi-metodo garantisce che le feature selezionate performino bene su criteri diversi (linearità, non-linearità, importanza contestuale).

Secondo, viene utilizzato uno split stratificato 70/15/15 con validation set completamente isolato e test set separato per il benchmark finale, rispetto al più comune split che non distingue tra valutazione e test.

Terzo, conduciamo un'analisi dettagliata delle performance per-class attraverso confusion matrix e metriche di classificazione, identificando le classi più problematiche (Bot, Other) e fornendo insights sui limiti dei modelli su classi fortemente sbilanciate

Infine, viene documentata in dettaglio la metodologia adottata per garantire la riproducibilità degli esperimenti, includendo parametri di training, configurazioni degli algoritmi e pipeline di preprocessing.

1.5 Struttura della Relazione

La relazione è organizzata come segue:

- **Sezione 2 – Stato dell’Arte:** Analisi della letteratura su ML per IDS e su CIC-IDS-2017 e analisi dei problemi metodologici identificati.
- **Sezione 3 – Caratterizzazione del Problema:** Formalizzazione del problema, definizione delle metriche di valutazione e analisi dello sbilanciamento delle classi.
- **Sezione 4 – Metodologia:** Descrizione dettagliata della pipeline sperimentale: EDA sistematica, Feature Engineering e Training dei modelli.
- **Sezione 5 – Implementazione:** Implementazione in termini di codice di quanto descritto nella metodologia.
- **Sezione 6 – Risultati Sperimentali:** Presentazione dei risultati con confusion matrix per-class e metriche di classificazione

- **Sezione 7 – Discussione:** Interpretazione dei risultati, analisi e discussione critica delle limitazioni metodologiche.
- **Sezione 8 – Conclusioni e Sviluppi Futuri:** Sintesi dei contributi chiave e direzioni per implementazioni future.

2 Stato dell'Arte

L'applicazione del Machine Learning alla sicurezza di rete ha visto un'evoluzione significativa negli ultimi venti anni, passando da approcci *signature-based*, efficaci solo contro minacce note, a paradigmi *anomaly-based* capaci di rilevare attacchi zero-day

2.1 Algoritmi di Machine Learning per IDS

2.1.1 Random Forest e Ensemble Methods

Gli algoritmi basati su alberi decisionali, in particolare Random Forest (RF), sono ampiamente utilizzati per la classificazione di traffico di rete [1]. RF funziona costruendo molti alberi decisionali su porzioni casuali del dataset e combinando le loro predizioni attraverso il voto di maggioranza. Questo approccio, chiamato *Bagging* (Bootstrap Aggregating), rende il modello robusto al rumore e all'overfitting.

Sharafaldin et al. (2018), nel paper di introduzione del CIC-IDS-2017, hanno ottenuto con Random Forest un weighted F1-score di 0.97, Precision 0.98 e Recall 0.97 confermando RF come baseline robusta per questo dataset.[8]

2.1.2 Gradient Boosting Machines

Algoritmi come LightGBM rappresentano un'evoluzione del Gradient Boosting, ottimizzati per l'efficienza computazionale. Ke et al. (2017) [6] hanno sviluppato LightGBM per gestire dataset massivi, utilizzando tecniche come il Gradient-based One-Side Sampling, ovvero "ignorare" i campioni con gradiente basso e raggruppamento delle feature correlate (Feature Bundling).

Tuttavia, il Gradient Boosting presenta una criticità importante: mentre RF combina alberi indipendenti riducendo la varianza, il Boosting costruisce alberi in sequenza, dove ogni albero cerca di correggere gli errori del precedente. Su dataset sbilanciati come CIC-IDS-2017 (classe minoritaria <0.01%), questo può portare a problemi se gli iperparametri non sono accuratamente ottimizzati.

2.1.3 Deep Learning

Le reti neurali profonde, come Multilayer Perceptron (MLP), possono apprendere rappresentazioni complesse dei dati attraverso strati multipli di neuroni. Tuttavia, su dati tabulari di rete, dove le feature hanno significati fisici eterogenei (tempi, conteggi, flag TCP), le reti neurali faticano a superare gli ensemble methods senza normalizzazione accurata e volumi massivi di training [5]. Inoltre, l'interpretabilità

limitata delle reti neurali pone problemi critici nel contesto della sicurezza, dove gli operatori devono comprendere le decisioni del sistema.

2.2 Problematiche Metodologiche in Letteratura

Engelen et al. [4] hanno identificato gravi difetti nella pipeline di generazione del CIC-IDS-2017 (TCP appendix, labelling errato). Più in generale, Ring et al. [7] evidenziano come la dipendenza da metadati di rete (IP, porta) limiti la generalizzabilità dei modelli addestrati su dataset pubblici.

2.2.1 Data Leakage Topologico

Anche Catillo et al. (2022) [2] dimostrano che modelli addestrati su CIC-IDS-2017 subiscono una significativa degradazione delle performance quando testati su reti diverse, evidenziando come l'inclusione di feature topologiche (IP, porte di destinazione) porti i modelli a memorizzare la struttura del testbed piuttosto che le caratteristiche comportamentali degli attacchi.

2.2.2 Class Imbalance e Metriche Inadeguate

Il dataset CIC-IDS-2017 presenta un Imbalance Ratio di 31.077:1 tra classe maggioritaria (BENIGN) e minoritaria (Other). In tali contesti di sbilanciamento estremo, l'accuratezza globale risulta inadeguata come metrica primaria (si veda la Sezione 3.5 per l'analisi dettagliata). Le principali strategie per gestire il class imbalance includono:

- **Resampling:** Undersampling (riduzione classe maggioritaria), oversampling (replicazione classe minoritaria), e SMOTE [3] (genera istanze sintetiche interpolando tra vicini minoritari).
- **Cost-Sensitive Learning:** Assegna pesi maggiori agli errori sulla classe minoritaria (es. `class_weight='balanced'` in scikit-learn).

In particolare, SMOTE deve essere applicato SOLO sul training set, all'interno di ciascun fold di cross-validation, per evitare data leakage tra train e validation.

2.2.3 Artefatti nel Dataset

Engelen et al. (2021) [4] identificano nel dataset ufficiale una quantità significativa di flussi problematici:

- **TCP Appendix Artifacts:** 768.784 flussi (<3 pacchetti totali), residui di connessioni incomplete che mancano di contenuto informativo per calcolare feature statistiche significative.
- **Valori Anomali:** Flussi con valori infiniti (divisioni per zero), e valori negativi in feature temporali.
- **Duplicati:** 329.691 flussi duplicati dovuti a errori di cattura pacchetti.

2.3 Confronto Quantitativo con lo Stato dell'Arte

La Tabella 2 confronta le performance del presente lavoro con lo studio baseline di Sharafaldin et al. (2018) [8], autori del dataset.

Tabella 2: Confronto con il baseline di Sharafaldin et al. (2018) su CIC-IDS-2017

Studio	Algoritmo	Features	Weighted F1	F1-Macro
Sharafaldin et al.	RF	80	0.97	-
Il mio studio	RF+SMOTE	20	0.998	0.904

L'approccio adottato raggiunge performance simili (F1 0.99 SMOTE vs 0.97 baseline) utilizzando solo 20 features (una riduzione del 74.7% rispetto al dataset originale), dimostrando l'efficacia della consensus feature selection. Il valore F1-Macro di 0.904 riflette la capacità del modello di rilevare anche le classi minoritarie, metrica non riportata nel paper originale del dataset.

2.4 Contributi di Questa Ricerca

Questo studio presenta diverse novità rispetto alla letteratura esistente:

1. **Consensus Feature Selection:** Le feature vengono scelte implementando un consensus ranking aggregando 4 metodi (Spearman Correlation, Mutual Information, ANOVA F-test, Random Forest Importance) per identificare le 20 features più robuste.
2. **SMOTE Partial Strategy Inside Cross-Validation:** Questo studio adotta SMOTE progressivo con gap-filling al 70%, applicato esclusivamente all'interno dei fold di training durante 5-fold stratified CV, garantendo che il validation set contenga solo dati originali.
3. **Baseline vs SMOTE con Validation-Test Gap:** Sono riportate le metriche sia per baseline che per SMOTE su tutti e tre gli algoritmi (RF, LightGBM, MLP), analizzando il gap validation-test (91.77% vs 90.43% F1-macro, gap 1.34%) per verificare la generalizzazione. Inoltre, investiga le ragioni del fallimento di LightGBM, fornendo insight sulle limitazioni del Gradient Boosting con gli iperparametri utilizzati.

3 Caratterizzazione del Problema

Il sistema di intrusion detection di rete tramite Machine Learning si configura come un problema di **classificazione multiclasse, ad alta dimensionalità e forte sbilanciamento**. L'obiettivo è far apprendere una funzione

$$f : \mathbb{R}^n \rightarrow \{\text{BENIGN}, \text{DoS}, \text{DDoS}, \text{BruteForce}, \text{WebAttack}, \text{PortScan}, \text{Bot}, \text{Other}\}$$

che mappi ogni flusso di rete-caratterizzato da n feature estratte dai pacchetti TCP/IP a una delle 8 classi di traffico.

Le sfide principali sono due: **(1)** l'elevato numero di features iniziali (79), e **(2)** il forte sbilanciamento tra classi, dove la classe Other (formata da Infiltration + Heartbleed) rappresenta solo lo 0.003% dei dati, contro l'80.87% di traffico benigno (Benign).

3.1 Dataset CIC-IDS-2017 e Preprocessing

Il dataset CIC-IDS-2017 è generato dal Canadian Institute for Cybersecurity e rappresenta uno scenario realistico di traffico di rete analizzato su un periodo di 5 giorni (3-7 luglio 2017). Come descritto nella Sezione 1, il dataset originale di 2.830.743 flussi è stato ridotto a 1.729.294 istanze (-38.9%) attraverso rimozione di artefatti TCP, valori anomali e duplicati.

Dopo il preprocessing e lo split stratificato 70/15/15, il Training Set contiene **1.210.461 istanze**, il Validation Set **259.438 istanze**, e il Test Set **259.395 istanze**. Lo split stratificato preserva la distribuzione proporzionale delle classi in ciascuna partizione, garantendo che anche le classi minoritarie siano rappresentate in tutti e tre i set.

3.2 Distribuzione delle Classi nel Training Set

La Tabella 3 riporta la distribuzione delle 8 classi nel Training Set con descrizioni comportamentali attese, evidenziando lo sbilanciamento estremo che costituisce la principale sfida modellistica.

Tabella 3: Distribuzione classi nel Training Set con caratteristiche comportamentali

Classe	Campioni	%	Descrizione e Comportamento Atteso
BENIGN	978.899	80.87	Traffico utente legittimo (HTTP, SSH, FTP). Alta varianza intrinseca, pattern interattivi simmetrici.
DoS	132.479	10.94	Denial of Service (Hulk, GoldenEye). Alta frequenza pacchetti, flow duration medio, IAT molto basso.
DDoS	89.546	7.40	Distributed DoS. Simile a DoS ma con origini multiple. Metriche rate-based elevate.
BruteForce	6.206	0.51	Tentativi password guessing (SSH/FTP). Flussi brevi, ripetitivi, alta frequenza flag FIN/RST.
WebAttack	1.405	0.12	XSS, SQL Injection. Difficili da rilevare con sole statistiche di flusso (payload-dependent).
PortScan	1.035	0.09	Scansione porte. Anomalie flag TCP (SYN senza ACK), conteggi flussi verso destinazioni multiple.
Bot	860	0.07	Traffico Command & Control. Mimetizzato come traffico legittimo (cifrato), beacon periodici.
Other	31	0.003	Infiltration + Heartbleed. Classe critica per rarità; richiede oversampling sintetico.

Nel training set, l’Imbalance Ratio raggiunge **31.577:1** tra classe maggioritaria (BENIGN: 978.899 campioni) e minoritaria (Other: 31 campioni). Questo sbilanciamento conferma che **l’accuracy non è una metrica affidabile**: un modello che ignora completamente le classi minoritarie (Other, Bot, PortScan, WebAttack, BruteForce) otterrebbe comunque accuracy elevata sommando solo Benign, DoS e DDoS.

3.3 Feature Engineering

Lo spazio delle feature originale comprende 79 variabili calcolate da CICFlowMeter. Per catturare l’**asimmetria del traffico**, fondamentale per distinguere comportamenti interattivi da attacchi automatici-sono state ingegnerizzate due feature ad alto potere discriminante:

3.3.1 PacketRatio

Definito come il rapporto tra pacchetti forward e backward:

$$\text{PacketRatio} = \frac{\text{Total_Fwd_Packets}}{\text{Total_Bwd_Packets}}$$

Interpretazione comportamentale:

- **Valore molto alto** (>10): Indica flood unidirezionale tipico di attacchi DoS/DDoS, dove l'attaccante invia migliaia di pacchetti senza ricevere risposte.
- **Valore bilanciato** (≈ 1): Tipico di connessioni interattive normali (HTTP request/response, SSH, DNS query/reply).
- **Invarianza temporale:** Non dipende dalla durata del flusso, rendendolo robusto a variazioni di scala temporale.

3.3.2 ByteRatio

Definito come il rapporto tra byte forward e backward:

$$\text{ByteRatio} = \frac{\text{Total_Length_Fwd_Packets}}{\text{Total_Length_Bwd_Packets}}$$

Interpretazione comportamentale:

- **ByteRatio $\gg 1$:** Indica upload massivo, tipico di data exfiltration (comandi piccoli in ingresso, dati estratti grandi in uscita).
- **ByteRatio $\ll 1$:** Indica download massivi, possibile botnet che scarica payload o C&C che invia comandi lunghi.
- **ByteRatio bilanciato:** Traffico simmetrico normale.

3.3.3 Validazione Empirica

L'analisi di correlazione di Spearman conferma l'efficacia di queste feature ingegnerizzate:

- **ByteRatio:** Correlazione **0.482** con le etichette di classe, posizionandosi tra i predittori più forti.
- L'analisi di Consensus Ranking (Sezione 4) colloca **ByteRatio** al 10° posto (Average Rank: 13.50) e **PacketRatio** al 16° posto (Average Rank: 19.50) su 79 feature totali.

3.4 Pipeline Anti-Leakage

In linea con le criticità identificate nello Stato dell'Arte (Sezione 2.2.1), è stata rimossa *a priori* l'unica feature topologica presente nel dataset originale :

- Destination Port

Studi sulla transferability di modelli IDS [2, 7] dimostrano che modelli addestrati su singoli dataset pubblici come CIC-IDS-2017 mostrano significativa degradazione delle performance quando testati su reti diverse o dataset correlati, evidenziando la criticità del data leakage indotto da feature topologiche.

Questa **data-induced leakage** si verifica perché il modello apprende pattern topologici specifici del testbed (ad esempio: “porta 22 verso servizio X è sempre BruteForce SSH”, “porta 80 è sempre WebAttack”) invece delle statistiche comportamentali.

La rimozione garantisce che il modello apprenda caratteristiche generalizzabili del traffico malevolo, non l’identità delle macchine coinvolte nel testbed, garantendo la trasferibilità su reti reali diverse.

3.5 Protocollo di Valutazione

Per garantire robustezza dei risultati e stime realistiche della capacità di generalizzazione, è adottato uno **split stratificato rigoroso 70/15/15**: (Training/Validazione/Test) con politica anti-leakage. I dettagli completi dello split, incluse dimensioni esatte delle partizioni e rationale della scelta sono descritti nella Sezione 4.4.1.

3.6 Metriche di Valutazione

Data la natura fortemente sbilanciata del problema, le metriche primarie sono state selezionate per dare eguale peso alle classi minoritarie critiche:

3.6.1 F1-Macro Score (Metrica Primaria)

Media aritmetica degli F1-score per-class, attribuendo peso uguale a tutte le classi indipendentemente dalla loro frequenza:

$$F1\text{-Macro} = \frac{1}{C} \sum_{i=1}^C F1_i$$

dove $C = 8$ è il numero di classi e

$$F1_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

è l’F1-score della classe i .

Rationale: Nel contesto cybersecurity, *ogni* classe, anche rara come Other (0.003%) o Bot (0.07%) è critica. La metrica F1-Macro non nasconde fallimenti su attacchi minoritari, a differenza di Accuracy o Weighted F1 dominate da BENIGN (80.87%).

3.6.2 Precision e Recall per-class

Metriche granulari per identificare performance su classi minoritarie:

- **Recall** (Sensitivity):

$$\text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}$$

Percentuale di attacchi reali della classe i correttamente rilevati. Un alto Recall minimizza i Falsi Negativi (attacchi non rilevati), questa è una metrica critica in cybersecurity.

- **Precision** (Positive Predictive Value):

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}$$

Percentuale di allarmi per classe i che corrispondono a veri attacchi. Un'alta Precision minimizza i Falsi Positivi, riducendo il numero di alert da controllare per i dipendenti.

3.6.3 Metriche Secondarie

- **Weighted F1-Score**: Media pesata per frequenza delle classi, fornisce indicazioni sulla performance globale.

$$F1\text{-Weighted} = \sum_{i=1}^C w_i \cdot F1_i \quad \text{dove} \quad w_i = \frac{n_i}{N}$$

- **Accuracy Globale**: Relegata a metrica terziaria in quanto può essere elevata ($>80\%$) anche per classificatori che ignorano classi minoritarie.

4 Metodologia

La metodologia adottata segue una pipeline strutturata in quattro fasi principali, progettata per garantire rigorosità scientifica e prevenire data leakage.

4.1 Overview della Pipeline

La pipeline si articola in quattro fasi sequenziali, ciascuna con output validati:

1. Fase 1: Exploratory Data Analysis (EDA)

Caricamento dei file CSV originali, data cleaning sistematico (gestione valori anomali, duplicati, artefatti TCP), label engineering per ridurre 15 classi granulari a 8 macro-categorie comportamentali.

2. Fase 2: Feature Engineering & Selection

Creazione di ratio features domain-driven (PacketRatio, ByteRatio), sanitizzazione anti-bias (rimozione identificatori topologici), consensus feature selection tramite aggregazione di 4 metodi complementari, riduzione dimensionalità da 79 a 20 features (74.7%).

3. Fase 3: Model Training & Evaluation

Split stratificato 70/15/15 (Train/Validation/Test), addestramento di 3 algoritmi (Random Forest, LightGBM, MLP) in 2 configurazioni (Baseline vs SMOTE), 5-fold stratified cross-validation, best model selection basata su F1-Macro.

4. Fase 4: Test Finale

Valutazione del modello ottimale su test set held-out, calcolo metriche finali, confusion matrix, analisi errori critici.

Riproducibilità: L'intero workflow è deterministico grazie al seed `random_seed=42`. Gli artefatti chiave (feature names selezionate, metriche di valutazione, risultati cross-validation) sono salvati in formato JSON/CSV per garantire tracciabilità.

4.2 Fase 1: Exploratory Data Analysis

4.2.1 Caricamento e Aggregazione Dataset

Il dataset CIC-IDS-2017 è distribuito in 8 file CSV giornalieri corrispondenti ai 5 giorni lavorativi di cattura (3-7 luglio 2017), con traffico benigno e attacchi distribuiti temporalmente. L'aggregazione sequenziale dei file richiede gestione di encoding non-standard e ottimizzazione della memoria per processare oltre 2.8M di flussi su hardware standard.

Pre-processing iniziale:

- **Gestione encoding:** I file sono caricati utilizzando encoding cp1252, questo step è necessario per gestire i caratteri speciali nelle label.
- **Normalizzazione nomi colonne:** Rimozione spazi leading/trailing e standardizzazione per uniformità del codice.
- **Ottimizzazione memoria:** Conversione tipi numerici (float64→float32, int64→int32) per ridurre footprint del 50% senza perdita di precisione significativa

4.2.2 Data Cleaning

La pipeline di pulizia adotta un approccio conservativo, rimuovendo solo istanze con evidenti anomalie che comprometterebbero l'addestramento, seguendo le linee guida metodologiche di Engelen et al. (2021) [4].

Pipeline di pulizia in 5 step sequenziali:

1. Gestione valori infiniti

I valori `np.inf` e `-np.inf`, causati da divisioni per zero nel calcolo di feature come *Flow Bytes/s* quando la durata del flusso è nulla, vengono sostituiti con `np.nan` per tracciabilità.

2. Rimozione valori mancanti

Righe contenenti `np.nan` in qualsiasi colonna vengono eliminate. Questo in quanto il dataset di default non dovrebbe avere missing values, oltre alla rimozione dei valori infiniti che in precedenza sono stati flaggati con `np.nan`.

3. Rimozione duplicati esatti

Flussi con valori identici in tutte le 79 features vengono deduplicati, mantenendo solo la prima occorrenza. Questi duplicati derivano da errori di cattura pacchetti o ridondanza nella pipeline di feature extraction.

4. Rimozione valori negativi non validi

Flussi con valori negativi in feature che devono essere non-negative (`Flow Duration`, `Flow Bytes/s`, `Flow Packets/s`, `Flow IAT Mean`) vengono rimossi. Questi valori negativi sono errori di calcolo di CICFlowMeter.

5. TCP Appendix Filtering [4]

Threshold: Total Fwd Packets + Total Bwd Packets ≥ 3

Rationale: Il minimo teorico per un flusso TCP informativo è il 3-way hand-shake completo (SYN, SYN-ACK, ACK). Flussi con meno di 3 pacchetti sono frammenti incompleti (TCP appendix artifacts) che mancano di contenuto statistico significativo. Engelen et al. dimostrano che questi artefatti costituiscono il 25.9% del dataset originale e introducono rumore nelle predizioni. Nel nostro caso, dopo la rimozione preliminare di valori anomali e duplicati, il filtro elimina 768.784 flussi (30.8% del dataset pre-pulito)

Il numero esatto di istanze rimosse in ciascun step e l'impatto percentuale sul dataset finale sono riportati nella Tabella ?? (Sezione 5.1).

4.2.3 Label Engineering: 15 \rightarrow 8 Classi

Il dataset originale presenta 15 label granulari che descrivono varianti specifiche di attacchi. Per ridurre la frammentazione delle classi minoritarie e aumentare la robustezza statistica, le label sono state aggregate in 8 macro-categorie comportamentali.

Tabella 4: Mapping label originali → 8 classi finali per classificazione multiclass

Classe Finale	Label Originali Aggregate
BENIGN	BENIGN
DoS	DoS Hulk, DoS GoldenEye, DoS Slowloris, DoS Slowhttptest
DDoS	DDoS
BruteForce	FTP-Patator, SSH-Patator
PortScan	PortScan
Bot	Bot
WebAttack	Web Attack - Brute Force, Web Attack - XSS, Web Attack - SQL Injection
Other	Heartbleed, Infiltration

Motivazione aggregazioni:

- **DoS variants:** Le 4 varianti condividono lo stesso pattern comportamentale (flood unidirezionale con alta frequenza di pacchetti) e differiscono solo nell’implementazione tecnica del tool di attacco. Da un punto di vista IDS, sono indistinguibili a livello di feature statistiche.
- **BruteForce:** FTP-Patator e SSH-Patator sono entrambi attacchi di password guessing, distinguibili solo dalla porta di destinazione (rimossa in fase di sanitizzazione anti-bias).
- **WebAttack:** Le 3 varianti operano sullo stesso protocollo (HTTP/HTTPS) e presentano pattern di traffico simili. La distinzione richiede analisi del payload, non disponibile nelle feature statistiche di CICFlowMeter.
- **Other:** Heartbleed e Infiltration sono attacchi estremamente rari (combined support < 0.003% del dataset). L’aggregazione in una classe **Other** permette di applicare SMOTE senza eccessiva frammentazione.

La distribuzione finale delle classi è documentata nella Tabella 1 (Sezione 3, Caratterizzazione del Problema), che evidenzia l’imbalance ratio critico di 31,077:1 tra classe maggioritaria (**BENIGN**, 80.87%) e minoritaria (**Other**, 0.003%).

4.3 Fase 2: Feature Engineering & Selection

4.3.1 Sanitizzazione Anti-Bias e Ratio Features

Come descritto in sottosottosezione 4.3.1, la sanitizzazione anti-bias rimuove **Destination Port** per prevenire data-induced leakage [2]. Le ratio features (**PacketRatio**, **ByteRatio**), definite in sottosezione 4.3, catturano asimmetrie di traffico bidirezionale. La validità empirica di queste feature è confermata dall’analisi di correlazione: **ByteRatio**

ottiene $\rho = 0.482$ con le etichette di classe, posizionandosi tra i predittori più forti del dataset (Sezione 5.2).

4.3.2 Consensus Feature Selection

Per identificare le feature più robuste e ridurre la dimensionalità senza bias di metodo singolo, è stato adottato un approccio di *consensus ranking* che aggrega i risultati di 4 tecniche complementari di feature importance.

4.3.2.1 Metodi di Ranking Individuali

1. Spearman Correlation (ρ)

Misura l'associazione monotonica (non necessariamente lineare) tra ciascuna feature e la variabile target.

2. Mutual Information (MI)

Quantifica la dipendenza statistica (lineare e non lineare) tra feature e target. Cattura relazioni complesse non rilevate da correlazione lineare.

3. ANOVA F-test

Testa se le medie della feature differiscono significativamente tra le classi. Assume distribuzione normale, quindi complementare a MI per identificare separabilità lineare.

4. Random Forest Feature Importance

Calcola la riduzione media di impurità (Gini) quando la feature è usata per split negli alberi.

4.3.2.2 Aggregazione tramite Consensus Ranking

Per ciascuna feature f_i , ogni metodo m produce un ranking $\text{Rank}_m(f_i) \in [1, 79]$ (1 = migliore, 79 = peggiore). Il consensus ranking è definito come:

$$\text{Rank}_{\text{consensus}}(f_i) = \frac{1}{4} \sum_{j=1}^4 \text{Rank}_j(f_i) \quad (1)$$

Le top-20 feature con ranking medio più basso (cioè 1 = migliore) vengono selezionate, riducendo la dimensionalità da 79 a 20 features (riduzione del 74.7%).

Rationale: L'aggregazione multi-metodo è un approccio per ridurre il bias intrinseco di ciascuna tecnica di feature selection, offrendo maggiore stabilità della selezione rispetto a metodi singoli, riducendo il rischio di overfitting sul training set.

La lista completa delle top-20 feature con i loro rank individuali e consensus è riportata nella Tabella 6 (Sezione 5.2).

4.3.3 Rimozione Ridondanza

Prima dell'applicazione del consensus ranking, vengono applicate due tecniche di pre-filtering per rimuovere feature con basso contenuto informativo o altamente correlate:

1. Variance Threshold

Feature con varianza < 0.01 vengono rimosse. Rationale: varianza quasi nulla indica valori quasi costanti, che non contribuiscono alla discriminazione tra classi.

2. Multicollinearity Filter

Per ogni coppia di feature con correlazione di Pearson $|r| > 0.95$, viene rimosso quella con correlazione più bassa rispetto al target. Rationale: feature altamente correlate portano informazione ridondante, aumentando il rischio di instabilità numerica in modelli lineari senza migliorare le performance.

Il numero esatto di feature rimosse e l'elenco dettagliato sono documentati nella Sezione 5.2 (paragrafi Low-Variance Features e Multicollinearità).

4.4 Fase 3: Model Training & Evaluation

4.4.1 Data Splitting Stratificato

Il dataset post-cleaning (dimensioni esatte in Sezione 5.1) viene suddiviso in tre partizioni disgiunte tramite split stratificato, che preserva le proporzioni originali delle classi in ciascun subset.

Split 70/15/15:

- **Training Set (70%):** Utilizzato per l'addestramento dei modelli e per il calcolo delle statistiche necessarie alla feature selection (correlazione, mutual information, ANOVA, RF importance). Le trasformazioni (StandardScaler, label encoding) vengono fittate esclusivamente su questo set.
- **Validation Set (15%):** Utilizzato per model selection basata su F1-Macro. Questo set *non* viene toccato durante la feature selection per prevenire data leakage-le feature sono selezionate usando solo statistiche del training set, poi applicate a validation/test.
- **Test Set (15%):** Held-out finale, completamente non visto durante training e validation. Utilizzato esclusivamente per la valutazione finale del modello ottimale, simulando un deployment real-world su dati nuovi.

Rationale dello split 70/15/15:

La suddivisione in *tre* set distinti (anziché due come in molti studi) permette di:

- Separare chiaramente *model development* (train+validation) da *final evaluation* (test), evitando il bias di riportare metriche su dati "visti" durante l'ottimizzazione.

- Calcolare il *validation-test gap* (differenza tra F1-Macro su validation e test), che quantifica il grado di generalizzazione: $\text{gap} < 0.02$ indica ottima generalizzazione, $\text{gap} > 0.05$ segnala possibile overfitting.

Nota: StandardScaler viene fissato *esclusivamente* sul training set, poi applicato tramite `transform()` a validation e test set. Questo prevede data leakage derivante dall'uso di media e deviazione standard calcolate su validation/test durante la normalizzazione.

4.4.2 Gestione Class Imbalance: Baseline vs SMOTE

L'imbalance ratio critico di 31,077:1 tra classe maggioritaria e minoritaria richiede strategie dedicate per evitare che i modelli collassino su predizioni triviali (sempre "BENIGN"). Vengono confrontate due configurazioni per ogni algoritmo:

4.4.2.1 Baseline: Class Weighting

Approccio algoritmico nativo disponibile in Random Forest e LightGBM tramite parametro `class_weight='balanced'`. Il peso di ciascuna classe k è calcolato come:

$$w_k = \frac{n_{\text{total}}}{n_{\text{classes}} \times n_k} \quad (2)$$

dove n_{total} è il numero totale di campioni, $n_{\text{classes}} = 8$, e n_k è il numero di campioni della classe k . Durante il training, la loss function viene pesata per penalizzare maggiormente gli errori sulle classi minoritarie.

Vantaggi: Zero overhead computazionale, nessuna sintesi di dati artificiali, mantiene distribuzione originale.

Limitazioni: Efficace per imbalance moderato (10:1), meno per sbilanciamenti estremi (>1000:1) dove i pesi diventano numericamente instabili.

4.4.2.2 SMOTE Progressive Gap-Filling

SMOTE (Synthetic Minority Over-sampling Technique) [3] genera istanze sintetiche per le classi minoritarie tramite interpolazione lineare tra k-nearest neighbors. Per evitare l'overfitting su dati sintetici, adottiamo una strategia *progressive gap-filling* al 70%.

Formula strategia parziale:

Per ciascuna classe i ordinata per frequenza crescente, il target di oversampling è:

$$\text{target}_i = \text{count}_i + 0.70 \times (\text{count}_{i+1} - \text{count}_i) \quad (3)$$

Cioè, ogni classe viene portata al 70% della distanza (*gap*) verso la classe successiva in frequenza, anziché bilanciarla completamente con la maggioritaria.

Safety Constraints per stabilità numerica:

- **Max 20× increment:** Nessuna classe può essere aumentata più di 20 volte la sua dimensione originale, per prevenire dominanza di dati sintetici.

- **Target $\leq 95\%$ next class:** Il target non può superare il 95% della classe immediatamente superiore, preservando l'ordinamento relativo.
- **No SMOTE se $\geq 5\%$ majority:** Classi che già rappresentano almeno il 5% della classe maggioritaria non vengono oversample.

Rationale: Il full balancing (100%) causa overfitting su dati sintetici, degradando le performance sul test set. Strategie parziali (30-70% gap-filling) migliorano il recall sulle classi minoritarie senza sacrificare precision sulla maggioritaria.

Implementazione rigorosa: SMOTE viene applicato *esclusivamente* ai training folds durante cross-validation, *mai* ai validation folds, per prevenire data leakage. Il validation set contiene solo dati reali, simulando la distribuzione del test set.

L'impatto quantitativo di SMOTE (numero di istanze generate per classe, dimensione finale del training set) è documentato nella Tabella 7 (Sezione 5.3).

4.4.3 Cross-Validation Protocol

Per stimare la variabilità delle performance e prevenire overfitting, ogni modello (algoritmo \times configurazione) viene valutato tramite **5-Fold Stratified Cross-Validation** sul training set.

Stratificazione: Ciascun fold mantiene le proporzioni originali delle 8 classi, garantendo che anche le classi minoritarie (Bot, Other) siano presenti in ogni training/validation partition.

Metriche monitorate:

- **F1-Macro (PRIMARY):**
Metrica primaria definita in sottosottosezione 4.4.3: media aritmetica degli F1-score per-class, che tratta tutte le classi con peso uguale indipendentemente dalla frequenza.
- **F1-Weighted:** Media pesata per support, riflette la distribuzione reale.
- **Accuracy:** Percentuale di predizioni corrette globali. Metrica secondaria, può essere ingannevole con imbalance estremo.

4.4.4 Algoritmi Selezionati

Sono stati selezionati 3 algoritmi rappresentativi di paradigmi di apprendimento differenti: diversi: Random Forest (ensemble bagging), LightGBM (gradient boosting sequenziale) e Multilayer Perceptron (deep learning). Per la descrizione teorica dettagliata di ciascun algoritmo, vedere la Sezione 2. Di seguito si motivano le ragioni della selezione dei modelli nel contesto specifico del problema.

4.4.4.1 Random Forest (RF)

Motivazioni della scelta:

- **Baseline di riferimento:** Sharafaldin et al. (2018), autori del dataset CIC-IDS-2017, ottengono $F1\ weighted = 0.97$ con RF su 80 features [8]. Questo stabilisce RF come benchmark contro cui confrontare le innovazioni metodologiche implementate (consensus feature selection, SMOTE progressivo).
- **Robustezza al class imbalance:** Il bagging riduce il rischio di overfitting sulla classe maggioritaria e la randomizzazione delle feature favorisce la diversità dell'ensemble, migliorando il recall su minority classes.
- **Interpretabilità:** Feature importance tramite MDI permette validazione delle scelte di feature engineering.
- **Assenza di preprocessing:** Non richiede normalizzazione e gestisce nativamente feature eterogenee (tempi, conteggi, flag).

4.4.4.2 LightGBM (LGB)

Motivazioni della scelta:

- **Efficienza computazionale:** GOSS (Gradient-based One-Side Sampling) e EFB (Exclusive Feature Bundling) rendono LightGBM $5-10\times$ più veloce di XGBoost su dataset massivi ($>1M$ samples), rilevante per deployment operativo con 1.2M training instances post-SMOTE.
- **Comparazione paradigmatica:** Confrontare boosting sequenziale (riduce bias) vs bagging parallelo (riduce varianza) per identificare quale approccio sia più adatto al problema IDS.
- **Test di robustezza:** Verificare se il boosting, noto per sensibilità a iperparametri su dataset sbilanciati, mantiene performance competitive con configurazione di default o richiede tuning intensivo.

4.4.4.3 Multilayer Perceptron (MLP)

Motivazioni della scelta:

- **Apprendimento non-lineare complesso:** Valutare se interazioni feature-feature di ordine superiore, non catturabili da metodi tree-based, giustificano la maggiore complessità computazionale.
- **Sensibilità a dati sintetici:** Confrontare MLP Baseline vs MLP+SMOTE per quantificare l'impatto di istanze sintetiche (gap-filling 70%) sulle performance delle reti neurali.
- **Trade-off interpretabilità-performance:** Verificare se la perdita di interpretabilità (assenza di feature importance nativa) sia compensata da miglioramenti quantitativi nelle metriche di classificazione.

Configurazioni Testate:

Ogni algoritmo viene testato in 2 configurazioni:

- **Baseline:** Class weighting nativo (`class_weight='balanced'` per RF/LGB, loss pesata per MLP)
- **Enhanced (SMOTE):** Progressive gap-filling al 70%, senza `class_weight`, rimosso per evitare doppia compensazione dello sbilanciamento

Totale: **6 modelli confrontati** (3 algoritmi \times 2 configurazioni).

4.4.5 Model Selection Strategy

La strategia di model selection adotta un protocollo rigoroso in 3 step per garantire stime non biased della capacità di generalizzazione:

1. Step 1: Cross-Validation su Training Set (70%)

Ogni modello (algoritmo \times configurazione) viene valutato tramite 5-fold stratified CV sul training set. Questo fornisce:

- Stima della variabilità delle performance (mean \pm std su 5 folds)
- Early detection di overfitting (alta varianza tra folds indica instabilità)

Output: Tabella con F1-Macro mean/std per ogni modello (Sezione 6.1).

2. Step 2: Model Selection su Validation Set (15%)

Ogni modello viene riaddestrato sul training set *completo* (non più foldato), poi valutato sul validation set. Il modello con **F1-Macro più alto su validation** viene selezionato come *best model*.

3. Step 3: Final Evaluation su Test Set (15%)

Tutti i 6 modelli candidati vengono valutati sul test set held-out, mai visto durante training, feature selection, cross-validation o model selection. La valutazione comparativa permette di calcolare il *validation-test gap* per ciascun modello, verificando la capacità di generalizzazione dell'intera pipeline. L'analisi dettagliata (confusion matrix, per-class metrics, analisi errori) viene condotta sul best model selezionato da validation.

4.4.5.1 Validation-Test Gap: Metrica di Generalizzazione

La differenza tra F1-Macro su validation e test quantifica il grado di generalizzazione:

$$\text{Gap}_{\text{val-test}} = \text{F1-Macro}_{\text{val}} - \text{F1-Macro}_{\text{test}} \quad (4)$$

Interpretazione:

- $\text{Gap} < 0.02$ (2%): Generalizzazione eccellente, modello robusto
- $0.02 \leq \text{Gap} < 0.05$ (2-5%): Generalizzazione accettabile
- $\text{Gap} \geq 0.05$ (>5%): Possibile overfitting, metriche validation sovrastimate

Questo gap è cruciale per verificare le performance reali del modello.

4.5 Fase 4: Test Finale

La valutazione finale del best model selezionato da validation viene eseguita sul test set held-out, simulando un deployment real-world su dati completamente nuovi.

Protocollo di test:

- **Dimensione:** 15% del dataset post-cleaning (259,395 samples, dettagli in Sezione 5.1)
- **Preprocessing:** Applicazione delle trasformazioni (StandardScaler, feature selection) fitted su training set
- **Nessun retraining:** I modelli non vengono riaddestrati, vengono utilizzati i modelli già addestrati sul training set.

Metriche finali calcolate:

- **Aggregate:** Accuracy, F1-Macro, F1-Weighted
- **Per-class:** Precision, Recall, F1-score per ciascuna delle 8 classi
- **Confusion Matrix:** Matrice 8x8 dettagliata per analisi errori (quali classi vengono confuse, pattern di misclassificazione)
- **Analisi istanze critiche:** Identificazione di falsi negativi su classi critiche (Bot, Other) per capire quali pattern sfuggono al modello

I risultati completi del test finale sono riportati nella Sezione 6.3 (Risultati Sperimentali), con confronto quantitativo rispetto allo stato dell'arte nella Sezione 7.4 (Discussione).

5 Implementazione

Questa sezione documenta come la metodologia descritta nella Sezione 4 è stata tradotta in codice eseguibile, seguendo il flusso dei tre notebook Jupyter:

1. `01_dataloading_and_eda.ipynb` — Caricamento dati e analisi esplorativa
2. `02_feature_engineering_selection.ipynb` — Feature engineering e selezione
3. `03_model_training_evaluation.ipynb` — Training e valutazione modelli

5.1 Fase 1: Caricamento Dati e Analisi Esplorativa

Il primo notebook parte dai file CSV grezzi del dataset CIC-IDS-2017 e produce un dataset pulito pronto per la feature engineering. L'obiettivo principale è rimuovere gli artefatti documentati da Engelen et al. [4] e caratterizzare le distribuzioni dei dati.

5.1.1 Aggregazione Multi-File e Ottimizzazione Memoria

Il dataset è distribuito in **8 file CSV** (uno per giorno di cattura, dal 3 al 7 luglio 2017). I file sono stati caricati con `pd.read_csv()`, specificando l'encoding `cp1252` per gestire i caratteri speciali nelle label originali.

Per non saturare la RAM è stata implementata una conversione automatica dei tipi numerici: tutte le colonne `float64` vengono convertite in `float32`, e `int64` in `int32`. Questo dimezza il footprint in memoria senza perdita significativa di precisione (errore relativo $< 10^{-6}$).

```
1 for filepath in all_csv_files:
2     df_temp = pd.read_csv(filepath, encoding='cp1252',
3                           low_memory=False)
4     df_temp.columns = df_temp.columns.str.strip()    # Rimuovi
5                                         spazi extra
6
7     # Conversione float64 -> float32, int64 -> int32
8     for col in
9         df_temp.select_dtypes(include='float64').columns:
10            df_temp[col] = df_temp[col].astype('float32')
11     for col in df_temp.select_dtypes(include='int64').columns:
12            df_temp[col] = df_temp[col].astype('int32')
13
14     dfs.append(df_temp)
15
16 df_raw = pd.concat(dfs, ignore_index=True)
```

Listing 1: Caricamento e ottimizzazione memoria

Output: Dataset concatenato: 2,830,743 righe x 79 colonne.

5.1.2 Pipeline di Cleaning in 5 Step

Sono state applicate quattro operazioni di pulizia in sequenza, salvando le statistiche di ogni passaggio in `eda_results.json` per tracciabilità:

1. **Step 1: Gestione valori infiniti.** CICFlowMeter produce valori $\pm\infty$ quando calcola rate (es. Flow Bytes/s) su flussi con durata zero. Questi vengono sostituiti con `NaN` per poterli rimuovere nel passo successivo.
2. **Step 2: Rimozione NaN.** Vengono eliminate tutte le righe con almeno un valore mancante. Il dataset originale non ha missing values strutturali, quindi i `NaN` indicano errori di calcolo del tool di estrazione.
3. **Step 3: Rimozione duplicati.** Vengono eliminati flussi con valori identici in tutte le 79 feature, mantenendo solo la prima occorrenza. Questi duplicati derivano probabilmente da errori nella cattura pacchetti.

4. **Step 4: Rimozione valori negativi.** Feature come Flow Duration e Flow Bytes/s non possono assumere valori negativi.
5. **Step 5: TCP Appendix Filtering.** Viene applicata la soglia Total Fwd Packets + Total Bwd Packets ≥ 3 , come raccomandato da Engelen et al. [4]. Flussi con meno di 3 pacchetti totali sono frammenti di handshake TCP incompleti che non contengono abbastanza informazione per calcolare statistiche temporali affidabili (IAT, durata).

```

1 # Step 1-3: Inf, NaN, duplicati
2 df_raw.replace([np.inf, -np.inf], np.nan, inplace=True)
3 df_raw.dropna(inplace=True)
4 df_raw.drop_duplicates(inplace=True)

5
6 # Step 4: Rimozione valori negativi non validi
7 cols_positive = ['Flow Duration', 'Flow Bytes/s',
8                   'Flow Packets/s', 'Flow IAT Mean']
9 neg_mask = (df_raw[cols_positive] < 0).any(axis=1)
10 df_raw = df_raw[~neg_mask].copy()

11
12 # Step 5: TCP filtering
13 MIN_PACKETS = 3
14 total_packets = df_raw['Total Fwd Packets'] + df_raw['Total
15 Backward Packets']
df_clean = df_raw[total_packets >= MIN_PACKETS].copy()

```

Listing 2: Pipeline cleaning sequenziale

Output finale:

Righe iniziali:	2,830,743
Dopo Step 1-4:	2,497,980 (circa -12%)
Dopo TCP filter:	1,729,294 (-30.8% rispetto a Step 3)

Step	Operazione	Righe rimosse	Righe rimanenti
—	Dataset originale	—	2,830,743
1	Inf → NaN	0	2,830,743
2	Rimozione NaN	2,867	2,827,876
3	Rimozione duplicati	329,691	2,498,185
4	Rimozione negativi	107	2,498,078
5	TCP Appendix Filtering	768,784	1,729,294

Il TCP filtering elimina quasi un terzo dei flussi rimanenti, confermando l'entità del problema degli artefatti nel dataset originale.

5.1.3 Label Engineering e Encoding

Le 15 label originali sono state raggruppate in 8 macro-categorie usando il dizionario definito nella Sezione 3. Per garantire consistenza tra tutti i notebook, è stato usato

`LabelEncoder` di scikit-learn, che assegna automaticamente codici numerici 0–7 in ordine alfabetico.

```
1 LABEL_MAPPING = {  
2     'BENIGN': 'Benign',  
3     'Bot': 'Bot',  
4     'FTP-Patator': 'BruteForce',  
5     'SSH-Patator': 'BruteForce',  
6     'DDoS': 'DDoS',  
7     'DoS GoldenEye': 'DoS',  
8     'DoS Hulk': 'DoS',  
9     'DoS Slowhttptest': 'DoS',  
10    'DoS slowloris': 'DoS',  
11    'PortScan': 'PortScan',  
12    'Web Attack - Brute Force': 'WebAttack',  
13    'Web Attack - Sql Injection': 'WebAttack',  
14    'Web Attack - XSS': 'WebAttack',  
15    'Heartbleed': 'Other',  
16    'Infiltration': 'Other'  
17 }  
18  
19 df_clean['Label'] =  
20     df_clean['Label'].str.strip().map(LABEL_MAPPING)  
21 # Encoding numerico (0-7)  
22 le = LabelEncoder()  
23 df_clean['Label_Encoded'] =  
24     le.fit_transform(df_clean['Label'])
```

Listing 3: Mapping e encoding consistente

Questo produce il mapping: 0 → Benign, 1 → Bot, ..., 7 → WebAttack.

5.1.4 Distribuzione delle Classi

La distribuzione finale delle classi (dettagliata nella Sezione 3) viene confermata dopo il cleaning: 1,398,479 campioni Benign (80.87%), 45 campioni Other (0.003%), con imbalance ratio di 31,077:1.

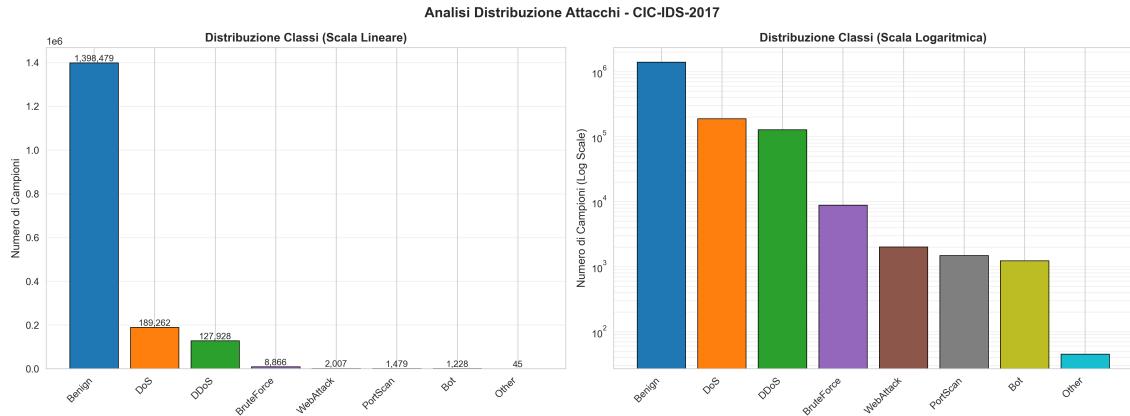


Figura 1: Distribuzione delle 8 classi su scala logaritmica . La classe *Benign* domina con 80.9% dei campioni, mentre *Other* rappresenta solo lo 0.003%.

5.1.5 Analisi Univariata delle Distribuzioni

Prima di procedere con la feature selection, vengono calcolate statistiche descrittive (media, deviazione standard, percentili, skewness) per tutte le 78 feature, salvandole in `descriptive_statistics.csv`.

```

1  from scipy.stats import skew, kurtosis
2
3  stats_list = []
4  for col in numeric_features:
5      stats = {
6          'Feature': col,
7          'Mean': df_clean[col].mean(),
8          'Std': df_clean[col].std(),
9          'Median': df_clean[col].median(),
10         'P75': df_clean[col].quantile(0.75),
11         'Max': df_clean[col].max(),
12         'Skewness': skew(df_clean[col]),
13         'Zeros_pct': 100 * (df_clean[col] == 0).sum() /
14             len(df_clean)
15     }
16     stats_list.append(stats)
17
18 stats_df = pd.DataFrame(stats_list)
19 stats_df.to_csv('descriptive_statistics.csv', index=False)

```

Listing 4: Calcolo statistiche descrittive

L'analisi rivela tre pattern principali:

1. **Skewness elevata:** Le distribuzioni sono fortemente asimmetriche verso destra. Ad esempio, `Flow Duration` ha mediana 262k ma media 24M, indicando che la maggior parte dei flussi è breve ma alcuni durano decine di secondi.

Questo è tipico del traffico di rete e viene gestito nativamente dai modelli tree-based tramite split su soglie.

2. **Outliers estremi:** Valori massimi eccezionalmente alti rispetto ai percentili centrali. Ad esempio, **Total Fwd Packets** ha $P75=8$ ma $\text{max}=220k$, questo è probabilmente causato da attacchi volumetrici come i DDoS floods.
3. **Zero-inflation:** Molte feature hanno una percentuale significativa di valori nulli. Ad esempio, **Bwd IAT Total** ha 14.1% di zeri, flussi unidirezionali senza pacchetti backward, tipici di flood attacks).

Queste caratteristiche motivano la scelta di:

- usare Random Forest e LightGBM *senza trasformazioni* (gestiscono nativamente skewness e zeri),
- applicare **StandardScaler** *obbligatoriamente* per MLP (neural networks sono sensibili a scale diverse).

5.1.6 Analisi Bivariata: Correlazione Feature-Target

Per identificare feature con forte capacità discriminante, viene calcolata la correlazione di Spearman tra ciascuna feature e il target binario (Benign vs Any Attack). La correlazione di Spearman è preferita a Pearson perché:

- robusta agli outliers (usa solo l'ordine relativo dei valori),
- cattura relazioni monotone (anche non lineari),
- non assume normalità delle distribuzioni.

```

1 from scipy.stats import spearmanr
2
3 y_binary = (df_clean['Label'] != 'Benign').astype(int)
4
5 target_corr_list = []
6 for col in numeric_features:
7     corr, pval = spearmanr(df_sample[col], y_sample)
8     target_corr_list.append({
9         'Feature': col,
10        'SpearmanCorr': abs(corr),
11        'pvalue': pval
12    })

```

Listing 5: Correlazione di Spearman con target binario

Le top-10 feature per correlazione assoluta sono riportate nella Tabella 5.

Tabella 5: Top-10 feature per correlazione di Spearman con target binario

Feature	$ \rho $
ByteRatio	0.4823
Fwd IAT Std	0.4492
Bwd Packets/s	0.4255
Flow IAT Max	0.4211
Flow IAT Std	0.4184
Flow IAT Mean	0.4130
Flow Packets/s	0.4125
Bwd Packet Length Min	0.4116
Fwd Packets/s	0.4079
Packet Length Variance	0.3705

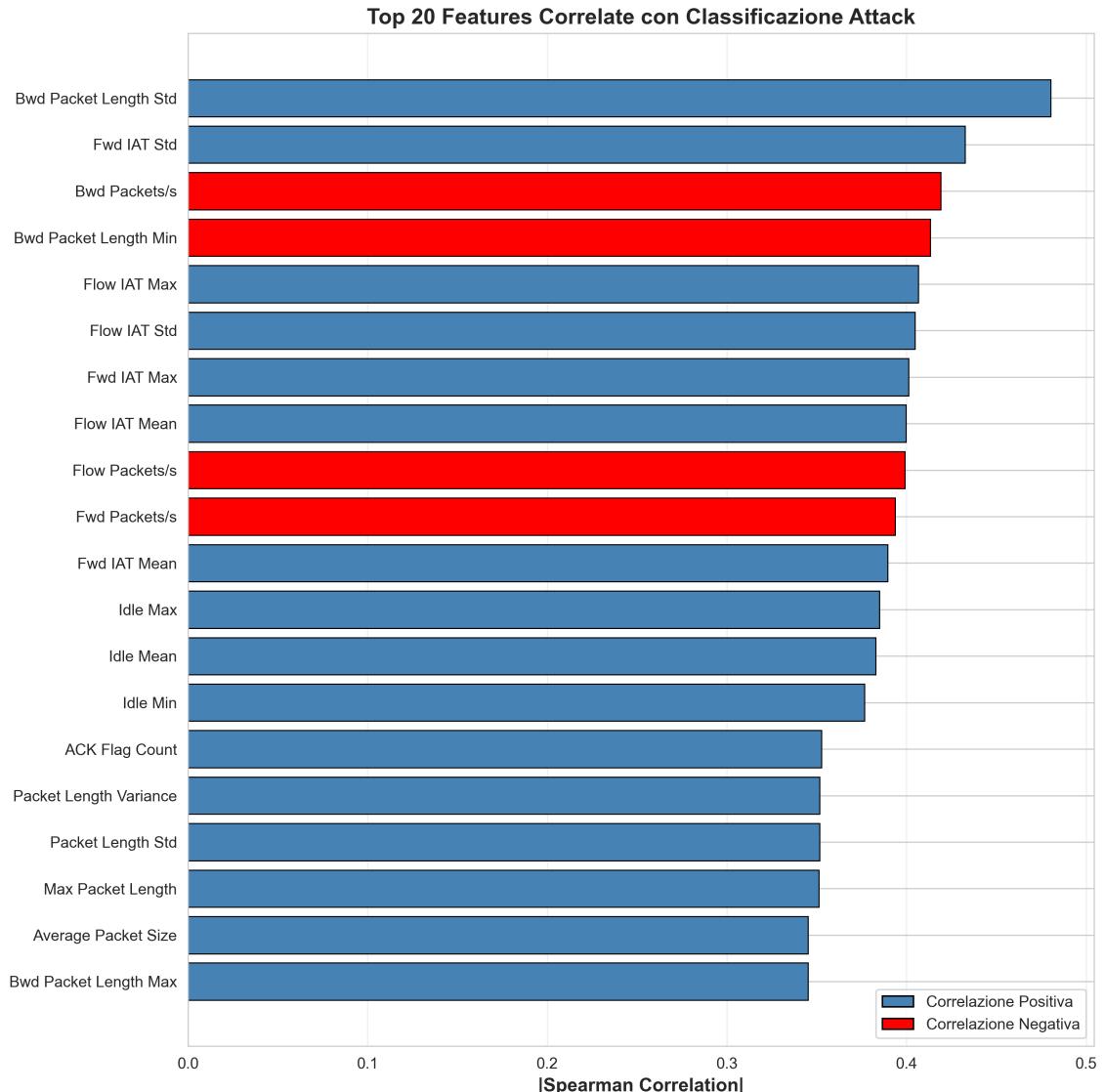


Figura 2: Correlazione di Spearman (top-20 feature) con target binario . Feature rate-based e IAT mostrano le associazioni più forti ($|\rho| > 0.4$).

Nota sulle correlazioni negative: Alcune feature mostrano correlazione negativa, ma il valore assoluto indica comunque forte associazione.

5.1.7 Identificazione Feature Problematiche

Prima di passare alla feature selection vera e propria, vengono identificate le feature problematiche quali: varianza quasi nulla e multicollinearità estrema.

5.1.7.1 Low-Variance Features.

Viene usato `VarianceThreshold(threshold=0.01)` per trovare feature con meno dell'1% di valori unici.

```
1 | from sklearn.feature_selection import VarianceThreshold
```

```

2 selector = VarianceThreshold(threshold=0.01)
3 selector.fit(X_sample[numeric_features])
4
5 low_var = [f for f, keep in zip(numeric_features,
6     selector.get_support()) if not keep]

```

Listing 6: Rilevamento low-variance

Output: 12 feature con varianza < 0.01, tra cui:

- *Bulk transfer metrics* (8 feature): sempre zero perché il pattern bulk transfer (sequenze di pacchetti >512 bytes) è assente nel dataset
- *Rare TCP flags* (4 feature): RST, CWE, ECE, URG (frequenza < 0.04%, troppo rari per essere affidabili)

5.1.7.2 Multicollinearità.

Viene calcolata la matrice di correlazione di Pearson e vengono identificate tutte le coppie con $r > 0.95$.

```

1 corr_matrix = X_sample[numeric_features].corr().abs()
2 upper_tri = np.triu(np.ones(corr_matrix.shape),
3     k=1).astype(bool)
4 upper = corr_matrix.where(upper_tri)
5
6 multicol_pairs = []
7 for col in upper.columns:
8     high_corr = upper[upper[col] > 0.95].index.tolist()
9     if high_corr:
10         for feat in high_corr:
11             multicol_pairs.append((col, feat,
12                 upper.loc[feat, col]))

```

Listing 7: Rilevamento coppie multicollineari

Output: 56 coppie multicollineari, coinvolgenti 21 feature candidate alla rimozione.
Esempi:

- Subflow Fwd Bytes \leftrightarrow Total Length of Fwd Packets ($r = 1.00$)
- Fwd Header Length.1 \leftrightarrow Fwd Header Length ($r = 1.00$)
- Avg Bwd Segment Size \leftrightarrow Bwd Packet Length Mean ($r = 1.00$)

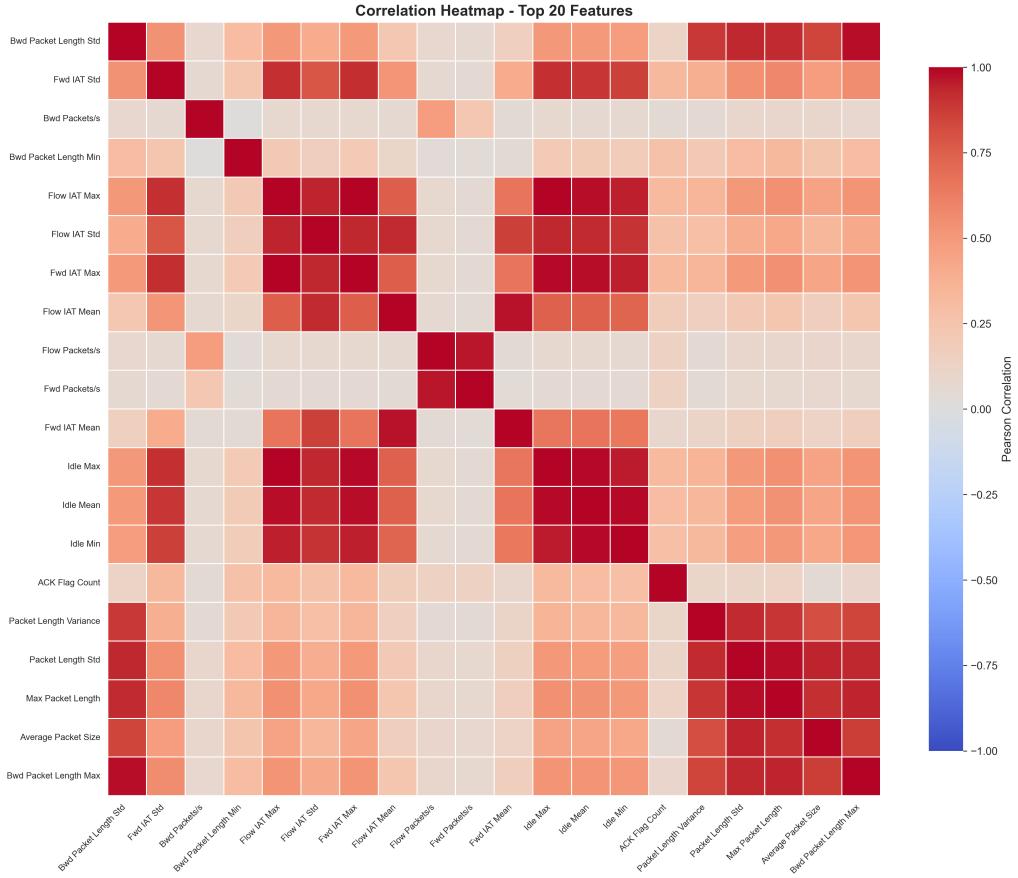


Figura 3: Heatmap correlazione di Pearson . Blocchi rossi intensi ($r > 0.95$) identificano 56 coppie ridondanti da rimuovere nella feature selection.

5.2 Fase 2: Feature Engineering e Selection

Il secondo notebook parte dal dataset pulito e produce i tre set finali (train/val/test) con solo le 20 feature selezionate. Le decisioni implementative principali riguardano: (1) rimozione degli identificatori topologici, (2) creazione delle ratio features, (3) split stratificato, (4) consensus feature selection.

5.2.1 Sanitizzazione Anti-Bias

Come discusso nella Sezione 4.3.1, viene rimossa tassativamente `Destination Port` per evitare che il modello memorizzi la topologia del testbed. Viene implementato anche un safety check per assicurarsi che la label encodata non finisca mai nel feature set X .

```

1 to_drop = ['Destination Port', 'Label_Encoded']
2 X = df_clean.drop(columns=to_drop) # Rimuovi colonne
   pericolose
3 y = df_clean['Label']
4

```

```

5 # Safety check
6 assert 'Label' not in X.columns
7 assert 'Label_Encoded' not in X.columns
8 assert 'Destination Port' not in X.columns

```

Listing 8: Rimozione identificatori e safety check (Notebook 02)

5.2.2 Creazione Ratio Features

Vengono ingegnerizzate due feature per catturare l’asimmetria bidirezionale del traffico (definite nella Sezione 4.3):

```

1 X['PacketRatio'] = np.where(
2     X['Total Backward Packets'] == 0, 0,
3     X['Total Fwd Packets'] / X['Total Backward Packets']
4 )
5
6 X['ByteRatio'] = np.where(
7     X['Total Length of Bwd Packets'] == 0, 0,
8     X['Total Length of Fwd Packets'] / X['Total Length of Bwd
9     Packets']
)

```

Listing 9: Ratio features per asimmetria traffico (Notebook 02)

La divisione per zero (tipica di attacchi flood unidirezionali) viene gestita assegnando valore 0, che semanticamente significa “traffico completamente sbilanciato verso forward”.

5.2.3 Split Stratificato 70/15/15

Vengono usati due `train_test_split` consecutivi per ottenere tre set con proporzioni esatte 70/15/15, mantenendo la stratificazione per garantire che anche la classe *Other* (45 campioni totali) sia presente in tutti e tre i set.

```

1 # Split 1: (Train+Val) vs Test
2 X_temp, X_test, y_temp, y_test = train_test_split(
3     X, y, test_size=0.15, stratify=y, random_state=42
4 )
5
6 # Split 2: Train vs Val (15% / 85% = 0.1765)
7 X_train, X_val, y_train, y_val = train_test_split(
8     X_temp, y_temp, test_size=0.1765, stratify=y_temp,
9     random_state=42
)

```

Listing 10: Double split stratificato (Notebook 02)

Output: Train: 1,210,461 (70%), Val: 259,438 (15%), Test: 259,395 (15%)

5.2.4 Consensus Feature Selection: Implementazione

Questa è la parte più complessa dell'implementazione. L'obiettivo è ridurre le 79 feature a 20, usando un approccio ensemble che aggrega 4 metodi statistici diversi per ridurre il bias di metodo singolo.

5.2.4.1 I quattro metodi di ranking.

Ogni metodo assegna un punteggio a ciascuna feature; poi le feature vengono ordinate dalla migliore alla peggiore (rank 1 = migliore, 46 = peggiore, dopo aver rimosso low-variance e multicollineari):

1. **Correlazione di Spearman:** Misura se esiste una relazione monotona (non necessariamente lineare) tra la feature e il target.

Esempio: Se `Flow Duration` cresce, anche la probabilità di essere un attacco cresce in modo consistente? Se sì, Spearman darà un punteggio alto.

2. **Mutual Information (MI):** Misura quanta informazione la feature fornisce sul target, in senso probabilistico. Cattura anche relazioni non lineari complesse.

Esempio: Se `Packet Length Mean` ha una distribuzione completamente diversa tra Benign e attacchi (anche se non in modo monotono), MI lo rileverà.

3. **ANOVA F-test:** Testa se le medie della feature sono significativamente diverse tra le 8 classi. Funziona bene quando le distribuzioni dentro ogni classe sono approssimativamente normali.

Esempio: Se `Fwd IAT Std` ha media 100 per Benign, 500 per DoS, 1000 per DDoS, ANOVA darà un punteggio alto perché le classi sono ben separate.

4. **Random Forest Feature Importance:** Calcola quanto ciascuna feature riduce l'impurità (Gini) negli split degli alberi. È l'unico metodo *context-aware*: tiene conto delle interazioni tra feature.

Esempio: `PacketRatio` potrebbe non essere correlata al target da sola, ma combinata con `Flow Duration` potrebbe diventare decisiva per distinguere DDoS da traffico normale.

5.2.4.2 Calcolo dei punteggi.

```
1 from scipy.stats import spearmanr, f_oneway
2 from sklearn.feature_selection import mutual_info_classif
3 from sklearn.ensemble import RandomForestClassifier
4
5 # 1. Spearman Correlation
6 spearman_scores = {}
7 for col in X_train.columns:
8     corr, _ = spearmanr(X_train[col], y_train_binary)
9     spearman_scores[col] = abs(corr)
```

```

10
11 # 2. Mutual Information
12 mi_scores = mutual_info_classif(X_train, y_train_encoded,
13     random_state=42)
13 mi_dict = {col: score for col, score in zip(X_train.columns,
14     mi_scores)}
14
15 # 3. ANOVA F-test
16 anova_scores = {}
17 for col in X_train.columns:
18     groups = [X_train[y_train == cls][col] for cls in
19         y_train.unique()]
20     fstat, _ = f_oneway(*groups)
21     anova_scores[col] = fstat
22
22 # 4. Random Forest Importance
23 rf_temp = RandomForestClassifier(n_estimators=50,
24     random_state=42)
24 rf_temp.fit(X_train, y_train_encoded)
25 rf_dict = {col: imp for col, imp in zip(X_train.columns,
26     rf_temp.feature_importances_)}

```

Listing 11: Calcolo ranking per ciascun metodo (Notebook 02)

5.2.4.3 Aggregazione consensus.

Per ciascun metodo, gli score vengono convertiti in ranking (la feature con score più alto prende rank 1). Poi viene calcolato il ranking medio per ogni feature:

$$\text{Rank}_{\text{consensus}}(f_i) = \frac{1}{4} (\text{Rank}_{\text{Spearman}}(f_i) + \text{Rank}_{\text{MI}}(f_i) + \text{Rank}_{\text{ANOVA}}(f_i) + \text{Rank}_{\text{RF}}(f_i))$$

```

1 # Converti score in rank (1 = best)
2 rank_spearman =
3     pd.Series(spearman_scores).rank(ascending=False)
4 rank_mi = pd.Series(mi_dict).rank(ascending=False)
5 rank_anova = pd.Series(anova_scores).rank(ascending=False)
6 rank_rf = pd.Series(rf_dict).rank(ascending=False)
7
7 # Calcola ranking medio
8 consensus = pd.DataFrame({'Feature': X_train.columns})
9 consensus = consensus.merge(rank_spearman, on='Feature',
10     how='left').rename(columns={0: 'Rank_Spear'})
10 consensus = consensus.merge(rank_mi, on='Feature',
11     how='left').rename(columns={0: 'Rank_MI'})
11 consensus = consensus.merge(rank_anova, on='Feature',
12     how='left').rename(columns={0: 'Rank_ANOVA'})
12 consensus = consensus.merge(rank_rf, on='Feature',
13     how='left').rename(columns={0: 'Rank_RF'})

```

```

13
14 consensus['AverageRank'] = consensus[['Rank_Spear',
15   'Rank_MI', 'Rank_ANOVA', 'Rank_RF']].mean(axis=1)
16 consensus = consensus.sort_values('AverageRank')
17 # Seleziona top-20
18 top20_features = consensus.nsmallest(20,
19   'AverageRank')['Feature'].tolist()

```

Listing 12: Aggregazione consensus e selezione top-20 (Notebook 02)

Tabella 6: Top-20 feature selezionate tramite consensus ranking:

#	Feature	Spear.	MI	ANOVA	RF	Cons.
1	Fwd IAT Std	2	8	1	5	5.5
2	Flow IAT Max	4	3	3	8	7.0
3	Packet Length Mean	14	1	8	3	7.2
4	Bwd Packet Length Max	13	4	2	10	7.5
5	Packet Length Variance	10	2	5	15	8.0
6	Max Packet Length	12	5	4	17	9.5
7	Flow Duration	16	13	9	12	12.5
8	Init Win Bytes backward	21	7	20	3	12.8
9	Flow IAT Std	5	19	7	23	13.5
10	ByteRatio	1	3	41	9	13.5
11	Flow IAT Mean	6	20	10	19	13.8
12	Total Length Fwd Packets	39	6	16	1	15.5
13	Bwd Packets/s	3	15	31	24	18.2
14	Fwd Packets/s	9	18	33	14	18.5
15	Flow Packets/s	7	21	30	18	19.0
16	PacketRatio	28	25	19	6	19.5
17	Fwd Packet Length Max	44	8	27	4	20.8
18	Init Win Bytes forward	33	9	22	20	21.0
19	Fwd Packet Length Mean	42	10	26	7	21.2
20	Bwd IAT Std	19	29	12	26	21.5

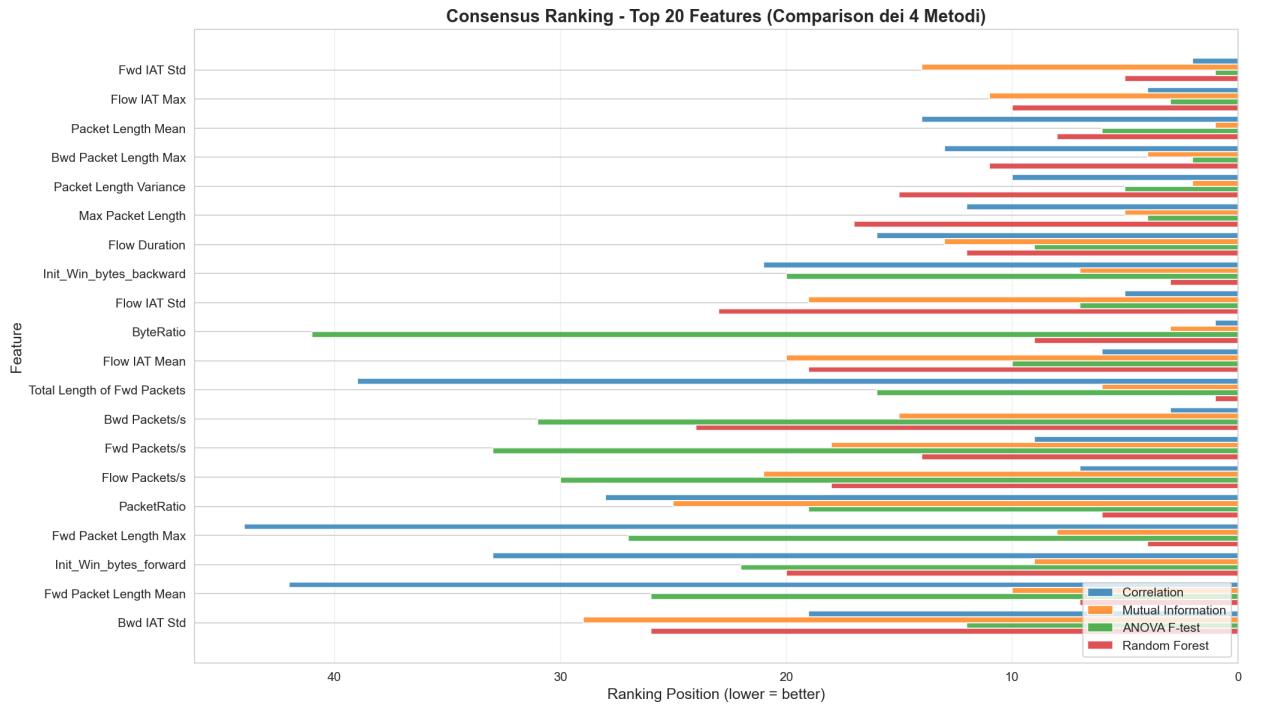


Figura 4: Confronto ranking dei 4 metodi per le top-20 feature (Notebook 02). Feature con ranking uniformemente basso su tutti i metodi (es. Fwd IAT Std, rank medio 5.5) sono i candidati più robusti.

Output: Le 20 feature finali vengono salvate in `selected_features.json` e applicate a train/val/test.

5.2.5 Applicazione Trasformazioni a Val/Test (Anti-Leakage Policy)

Applichiamo le stesse trasformazioni (rimozione feature) a Val e Test usando le maschere calcolate su Train.

Nota: Val e Test NON hanno influenzato la Feature Selection. Questo garantisce:

1. *No Data Leakage*: Statistiche calcolate solo su Train
2. *Valutazione Onesta*: Val/Test simulano dati mai visti
3. *Generalizzazione Reale*: Performance su Val/Test riflette deployment

```

1 X_val_final = X_val.loc[:, top20_features]
2 X_test_final = X_test.loc[:, top20_features]
3
4 # Verifica coerenza
5 assert list(X_train_final.columns) ==
   list(X_val_final.columns) == list(X_test_final.columns)

```

Listing 13: Applicazione mask a Val/Test (Notebook 02)

Output: Train: (1,210,461, 20), Val: (259,438, 20), Test: (259,395, 20)

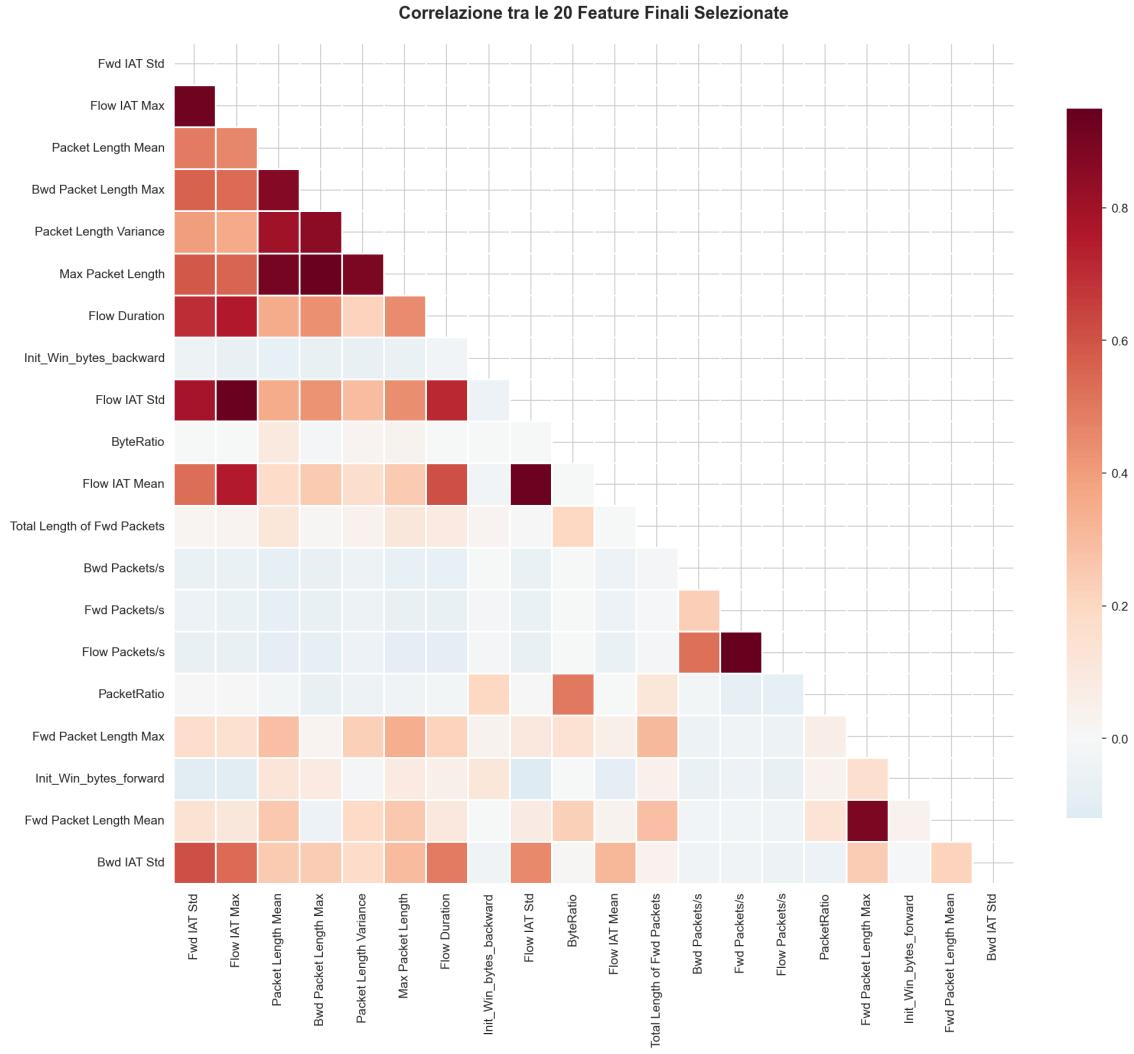


Figura 5: Heatmap correlazione tra le 20 feature finali selezionate (Notebook 02). Il consensus ranking ha eliminato efficacemente le coppie multicollineari ($r < 0.95$ per tutte le coppie).

5.3 Fase 3: Training e Valutazione Modelli

Il training dei modelli utilizza una doppia strategia (Baseline vs SMOTE) e valutazione sistematica. Le decisioni implementative chiave riguardano: (1) configurazione modelli conservativa vs aggressiva, (2) SMOTE gap-filling progressivo, (3) scaling per MLP, (4) cross-validation rigorosa.

5.3.1 Configurazione Modelli: Doppia Strategia Esplorativa

Per ciascun algoritmo sono state utilizzate *due configurazioni distinte* in fasi diverse della pipeline, seguendo un approccio esplorativo che bilancia vincoli hardware e ricerca di performance ottimali.

5.3.1.1 Strategia 1: Configurazione Conservativa (Cross-Validation).

Utilizzata nella fase di cross-validation per garantire training rapido e memoria gestibile durante l'esplorazione iterativa dei 5 fold.

Random Forest - Conservativa:

```
1 rf_config = {  
2     'n_estimators': 50,  
3     'max_depth': 20,  
4     'min_samples_split': 10,  
5     'min_samples_leaf': 5,  
6     'class_weight': 'balanced',  
7     'max_samples': 0.8,  
8     'random_state': 42,  
9     'n_jobs': 4  
10 }
```

Listing 14: Configurazione RF conservativa (Notebook 03, Section 4)

Motivazioni:

- `n_estimators=50`: Training time ~ 220 s, gestibile su hardware consumer
- `max_depth=20`: Profondità moderata, previene overfitting
- `max_samples=0.8`: Bootstrap ridotto, risparmio memoria

LightGBM - Conservativa:

```
1 lgb_config = {  
2     'n_estimators': 100,  
3     'max_depth': 7,  
4     'learning_rate': 0.05,  
5     'min_child_samples': 50,  
6     'num_leaves': 31,  
7     'is_unbalance': True,  
8     'random_state': 42,  
9     'n_jobs': 4  
10 }
```

Listing 15: Configurazione LGB conservativa (Notebook 03, Section 4)

Motivazioni:

- `max_depth=7`: Best practice LightGBM [6], alberi shallow riducono overfitting
- `learning_rate=0.05`: Conservativo, stabilizza boosting su minority classes
- `num_leaves=31`: $2^5 - 1$, cap standard per leaf-wise growth

5.3.1.2 Strategia 2: Configurazione Aggressiva (Validation/Test).

Utilizzata nella fase di model selection su validation set per testare se configurazioni più potenti potessero migliorare le performance finali.

Random Forest - Aggressiva:

```
1 rf_config = {  
2     'n_estimators': 200,  
3     'max_depth': 25,  
4     'min_samples_split': 5,  
5     'min_samples_leaf': 2,  
6     'class_weight': 'balanced',  
7     'random_state': 42,  
8     'n_jobs': -1  
9 }
```

Listing 16: Configurazione RF aggressiva (Notebook 03, Section 5)

Motivazioni:

- `n_estimators=200`: 4× più alberi, maggior potenza predittiva
- `max_depth=25`: Alberi più profondi, catturano pattern complessi
- `min_samples_split=5, min_samples_leaf=2`: Regularizzazione ridotta, maggior fitting capacity

LightGBM - Aggressiva:

```
1 lgb_config = {  
2     'n_estimators': 200,  
3     'max_depth': 25,  
4     'learning_rate': 0.1,  
5     'num_leaves': 50,  
6     'class_weight': 'balanced',  
7     'random_state': 42,  
8     'n_jobs': -1  
9 }
```

Listing 17: Configurazione LGB aggressiva (Notebook 03, Section 5)

Motivazioni:

- `max_depth=25`: Boosting profondo, massima capacità discriminante
- `learning_rate=0.1`: 2× più veloce, convergenza più rapida
- `num_leaves=50`: $\approx 2^{5.6}$, maggior complessità foglie

Risultato della strategia aggressiva: Come analizzato in dettaglio nella Sezione 6.2.4, la configurazione aggressiva LightGBM ha prodotto *peggioramenti significativi* rispetto alla baseline conservativa (Accuracy validation: 0.44 vs atteso >0.90,

F1-Macro: 0.21 vs atteso >0.60). Questo conferma le raccomandazioni della letteratura [6] che sconsigliano alberi troppo profondi per il boosting gradient-based. La configurazione Random Forest aggressiva ha invece mantenuto performance stabili, dimostrando la maggior robustezza dell'ensemble bagging rispetto al boosting su configurazioni estreme.

5.3.1.3 Multilayer Perceptron (MLP)

Per il neural network sono state utilizzate due implementazioni distinte nelle due fasi della pipeline, questo perché la fase di cross-validation richiede compatibilità con `cross_validate` di scikit-learn, mentre per la valutazione finale si è scelto di passare a Keras per avere maggiore controllo sull'architettura e sul processo di training.

Configurazione Cross-Validation: Utilizza `MLPClassifier` di scikit-learn, compatibile con `cross_validate` e la funzione `smote_cv` manuale.

```

1 mlp_config = {
2     'hidden_layer_sizes': (256, 128, 64),
3     'activation': 'relu',
4     'solver': 'adam',
5     'alpha': 0.0001,
6     'batch_size': 512,
7     'max_iter': 50,
8     'early_stopping': True,
9     'validation_fraction': 0.1,
10    'random_state': 42
11 }
12
13 mlp_pipeline = Pipeline([
14     ('scaler', StandardScaler()),
15     ('mlp', MLPClassifier(**mlp_config))
16 ])

```

Listing 18: Configurazione MLP per CV

Configurazione Validation/Test (Section 5): Utilizza `tensorflow.keras`, che offre maggiore controllo su architettura (Dropout per layer), callback (`EarlyStopping` con `restore_best_weights`), e validazione su set esterno invece che su split interno.

```

1 mlp_config_dict = {
2     'hidden_layers': [256, 128, 64],
3     'dropout_rate': 0.3,
4     'learning_rate': 0.001,
5     'batch_size': 64,
6     'epochs': 100,
7     'patience': 15
8 }
9
10 mlp = keras.Sequential()
11 mlp.add(layers.Input(shape=(X_train.shape[1],)))

```

```

12 for units in mlp_config_dict['hidden_layers']:
13     mlp.add(layers.Dense(units, activation='relu'))
14     mlp.add(layers.Dropout(mlp_config_dict['dropout_rate']))
15 mlp.add(layers.Dense(8, activation='softmax'))
16
17 mlp.compile(
18     optimizer=keras.optimizers.Adam(
19         learning_rate=mlp_config_dict['learning_rate']),
20         loss='sparse_categorical_crossentropy',
21         metrics=['accuracy'])
22 )
23
24 early_stop = EarlyStopping(
25     monitor='val_loss',
26     patience=mlp_config_dict['patience'],
27     restore_best_weights=True
28 )

```

Listing 19: Configurazione MLP Keras per Validation/Test (Notebook 03, Section 5)

Motivazioni del passaggio a Keras per la fase finale:

- **Dropout(0.3)**: durante il training, il 30% dei neuroni viene disattivato casualmente ad ogni aggiornamento. Questo obbliga la rete a non dipendere da singoli neuroni, riducendo l'overfitting.
- **EarlyStopping(patience=15, restore_best_weights=True)**: il training si interrompe se la loss sul validation set non migliora per 15 epoche consecutive. Con `restore_best_weights=True`, i pesi utilizzati per la predizione finale sono quelli dell'epoca con loss minima, non quelli dell'ultima epoca.
- **batch_size=64**: ad ogni epoca, i parametri della rete vengono aggiornati ogni 64 campioni. Batch più piccoli rispetto alla CV (512) producono più aggiornamenti per epoca aumentando la probabilità che la rete incontri e impari dalle classi minoritarie.
- **StandardScaler esterno**: la normalizzazione viene calcolata esclusivamente sui dati di training (`fit`), poi applicata a validation e test (`transform`). Se lo scaler venisse calcolato anche su validation o test, le statistiche di quei set (media e deviazione standard) influenzerebbero il training, introducendo data leakage.

Nota: I risultati di cross-validation (Sezione 6.1) si riferiscono alla configurazione sklearn, mentre i risultati su validation e test set (Sezioni 6.2 e 6.3) si riferiscono alla configurazione Keras. Questa differenza spiega, in parte la variazione di performance tra CV e validation.

5.3.2 SMOTE Gap-Filling Strategy: Implementazione Progressiva

L'implementazione della strategia SMOTE (descritta nella Sezione 4.4.2.2) segue un approccio progressivo order-preserving per evitare oversynthesis.

5.3.2.1 Formula Gap-Filling.

Per ogni classe minoritaria c_i :

$$\text{target}(c_i) = \text{count}(c_i) + 0.70 \times [\text{count}(c_{i+1}) - \text{count}(c_i)]$$

con vincoli di sicurezza: (1) max $20\times$ incremento, (2) $<95\%$ della classe successiva.

```

1 majority_count = class_counts.max()
2 smote_targets = {}

3
4 # Sort classes by count (ascending)
5 class_order = sorted([(class_counts[idx], cls, idx)
6                      for idx, cls in enumerate(class_names)])
7
8 for i, (current_count, current_cls, current_idx) in
9     enumerate(class_order):
10    percent_of_majority = (current_count / majority_count) *
11        100
12
13    # Classi >5% majority: no change
14    if percent_of_majority > 5.0:
15        target_count = current_count
16    else:
17        # Trova la classe immediatamente successiva
18        if i < len(class_order) - 1:
19            next_count = class_order[i + 1][0]
20            gap = next_count - current_count
21
22            # Target: current + 70% del gap
23            target_count = current_count + int(gap * 0.70)
24
25            # Safety caps:
26            # 1. Max 20x increment (overfitting protection)
27            max_by_increment = current_count * 20
28            target_count = min(target_count, max_by_increment)
29
30            # 2. Deve rimanere <95% della classe successiva
31            max_by_order = int(next_count * 0.95)
32            target_count = min(target_count, max_by_order)
33
34        else:
35            # la classe pi grande (Benign)
36            target_count = current_count
37
38    smote_targets[current_cls] = target_count

```

Listing 20: SMOTE gap-filling progressivo (Notebook 03)

Output SMOTE strategy:

Tabella 7: SMOTE gap-filling strategy: samples originali → target (Notebook 03)

Classe	Originale	Post-SMOTE	Incremento	Motivazione
Other	31	611	19.7×	Extreme minority (0.003%)
Bot	860	982	1.1×	Minimal intervention
PortScan	1,035	1,294	1.3×	Light boosting
WebAttack	1,405	4,765	3.4×	Moderate synthesis
BruteForce	6,206	64,543	10.4×	Gap-fill to DDoS
DDoS	89,546	89,546	1.0×	>5% majority
DoS	132,479	132,479	1.0×	>5% majority
Benign	978,899	978,899	1.0×	Majority class
Total	1,210,461	1,273,119	+5.2%	

L'incremento complessivo del +5.2% (vs 100–300% tipici di full balancing) minimizza il rischio di overfitting su dati sintetici, mantenendo l'ordine perfettamente preservato.

5.3.3 Data Scaling per MLP

Neural networks richiedono scaling obbligatorio. `StandardScaler` viene fittato solo su train e poi applicato a val/test per evitare data leakage.

```

1 from sklearn.preprocessing import StandardScaler
2
3 scaler = StandardScaler()
4 X_train_scaled = scaler.fit_transform(X_train) # Fit SOLO su
      train
5 X_val_scaled = scaler.transform(X_val)          # Transform
      su val
6 X_test_scaled = scaler.transform(X_test)         # Transform
      su test
7
8 print(f"Train shape: {X_train_scaled.shape}")
9 print(f"Val shape: {X_val_scaled.shape}")
10 print(f"Test shape: {X_test_scaled.shape}")

```

Listing 21: Scaling rigoroso per MLP (Notebook 03)

5.3.4 Cross-Validation Rigorosa con SMOTE Inside Loop

Per evitare data leakage, SMOTE viene applicato SOLO ai fold di training, MAI ai fold di validation durante la cross-validation.

```
1 from sklearn.model_selection import StratifiedKFold
2 from imblearn.over_sampling import SMOTE
3
4 skf = StratifiedKFold(n_splits=5, shuffle=True,
5   random_state=42)
6
7 macro_f1_scores = []
8 for train_idx, val_idx in skf.split(X_train, y_train_encoded):
9   X_train_fold, X_val_fold = X_train.iloc[train_idx],
10    X_train.iloc[val_idx]
11   y_train_fold, y_val_fold = y_train_encoded[train_idx],
12    y_train_encoded[val_idx]
13
14   # SMOTE applicato SOLO a training fold
15   smote = SMOTE(sampling_strategy=smote_strategy_config,
16     random_state=42)
17   X_train_resampled, y_train_resampled =
18     smote.fit_resample(X_train_fold, y_train_fold)
19
20   # Training
21   model = RandomForestClassifier(**rf_config)
22   model.fit(X_train_resampled, y_train_resampled)
23
24   # Evaluation (NO SMOTE su validation fold!)
25   y_pred = model.predict(X_val_fold)
26   f1 = f1_score(y_val_fold, y_pred, average='macro')
27   macro_f1_scores.append(f1)
28
29 print(f"Macro F1: {np.mean(macro_f1_scores):.4f}
30       {np.std(macro_f1_scores):.4f}")
```

Listing 22: 5-Fold CV con SMOTE inside loop (Notebook 03)

Questa implementazione garantisce che validation fold rimanga completamente isolato da qualsiasi operazione di resampling, simulando dati mai visti.

5.3.5 Final Training e Test Set Evaluation

Dopo la cross-validation esplorativa, i modelli vengono riaddestrati sull'intero training set e valutati su validation set per model selection, e infine testati sul test set held-out.

```
1 # 1. Training su intero train set (con SMOTE)
2 smote = SMOTE(sampling_strategy=smote_strategy_config,
3   random_state=42)
```

```

3 X_train_smote, y_train_smote = smote.fit_resample(X_train,
4     y_train_encoded)
5
6 rf_smote = RandomForestClassifier(**rf_config)
7 rf_smote.fit(X_train_smote, y_train_smote)
8
9 # 2. Evaluation su validation set (per model selection)
10 y_val_pred = rf_smote.predict(X_val)
11 val_acc = accuracy_score(y_val_encoded, y_val_pred)
12 val_f1 = f1_score(y_val_encoded, y_val_pred, average='macro')
13
14 print(f"Validation Accuracy: {val_acc:.4f}")
15 print(f"Validation F1-Macro: {val_f1:.4f}")
16
17 # 3. FINAL TEST
18 y_test_pred = rf_smote.predict(X_test)
19 test_acc = accuracy_score(y_test_encoded, y_test_pred)
20 test_f1 = f1_score(y_test_encoded, y_test_pred,
21     average='macro')
22
23 print(f"Test Accuracy: {test_acc:.4f}")
24 print(f"Test F1-Macro: {test_f1:.4f}")

```

Listing 23: Training finale e test evaluation (Notebook 03)

Best model: RF + SMOTE con Validation F1-Macro = 0.9177, Test F1-Macro = 0.9043 (analisi dettagliata nella Sezione 6).

6 Risultati Sperimentali

Questa sezione presenta i risultati sperimentali ottenuti dalla pipeline implementata nella Sezione 5. L'analisi procede in tre fasi sequenziali:

1. **Cross-validation (CV):** Valutazione su 5 partizioni diverse del training set per stimare la stabilità dei modelli
2. **Validation set:** Selezione del best model su dati held-out (mai visti durante training)
3. **Test set:** Valutazione di tutti i 6 modelli per verificare la capacità di generalizzazione, con analisi dettagliata del best model

Tutte le metriche sono calcolate sulle 8 classi (Benign, Bot, BruteForce, DDoS, DoS, Other, PortScan, WebAttack) definite nella Sezione 3.

6.1 Risultati Cross-Validation (5-Fold Stratified)

La cross-validation fornisce una stima della *variabilità* delle performance: ogni modello viene valutato 5 volte su partizioni diverse del training set. Questo permette di capire quanto siano stabili i risultati.

Tabella 8: Risultati Cross-Validation 5-Fold Stratified (Notebook 03)

Modello	F1-Macro	Std	Weighted F1	Accuracy	Tempo (s)
RF Baseline	0.8751	± 0.0237	0.9979	0.9976	220
RF + SMOTE	0.8882	± 0.0110	0.9978	0.9974	3,275
LGB Baseline	0.6211	± 0.0673	0.9833	0.9750	129
LGB + SMOTE	0.6834	± 0.0337	0.9902	0.9883	3,068
MLP Baseline	0.8113	± 0.0180	0.9954	0.9959	1,815
MLP + SMOTE	0.8303	± 0.0351	0.9949	0.9944	5,258

Legenda metriche:

- **F1-Macro:** Media semplice degli F1-Score delle 8 classi (misura la performance su *tutte* le classi, incluse quelle piccole)
- **Std (Deviazione Standard):** Misura la variabilità tra i 5 fold (più bassa = modello più stabile)
- **Weighted F1:** F1-Score pesato per il numero di campioni per classe (dominato dalle classi grandi come Benign)
- **Accuracy:** Percentuale di predizioni corrette sul totale (può essere ingannevole su dataset sbilanciati)

Osservazioni:

- **RF + SMOTE:** Migliore F1-Macro (0.8882) e stabilità più elevata (std 0.0110, dimezzata rispetto a Baseline). SMOTE rende il modello più robusto.
- **LGB:** Performance basse (F1-Macro < 0.70) con elevata instabilità (std 0.0673, la più alta) anche se migliora con SMOTE. Questo anticipa problemi successivi, verificatosi sul validation set.
- **MLP + SMOTE:** Guadagno +1.90 punti percentuali in F1-Macro, ma aumento varianza (std da 0.0180 a 0.0351). SMOTE aiuta ma rende il modello più sensibile alla composizione dei fold.
- **Costo SMOTE:** Il tempo di training aumenta 10–15× per RF/LGB e 3× per MLP.
- **Gap Accuracy vs F1-Macro:** LGB Baseline mostra Accuracy 97.5% ma F1-Macro 62.1%, dunque una differenza del differenza 35.4 %. Questo dimostra che Accuracy è ingannevole: un classificatore "sempre Benign" raggiungerebbe 80.9% accuracy su questo dataset anche senza rilevare alcun attacco.

6.2 Risultati Validation Set

6.2.1 Panoramica Performance

Ogni modello viene valutato sul **validation set** (259,438 campioni completamente separati dal training set). Questa fase serve a selezionare il best model prima di valutarlo sul test set finale.

Tabella 9: Performance complessiva su Validation Set — 6 modelli confrontati (Notebook 03)

Modello	Accuracy	F1-Macro	Interpretazione
RF + SMOTE	0.9985	0.9177	Best model
RF Baseline	0.9984	0.9118	Baseline solido
MLP + SMOTE	0.9946	0.7989	SMOTE efficace (+9.80 %)
MLP Baseline	0.9944	0.7009	Difficoltà su classi piccole
LGB + SMOTE	0.7768	0.2670	Performance critiche
LGB Baseline	0.4366	0.2097	Worst model

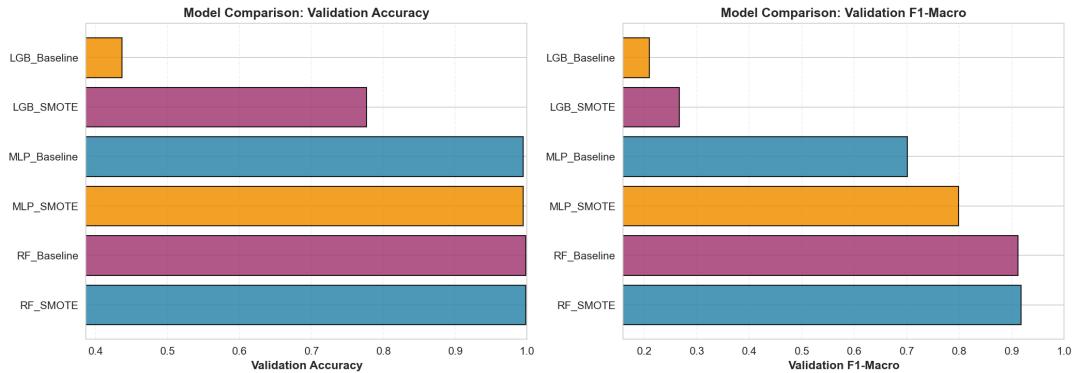


Figura 6: Confronto performance su validation set: Accuracy (sinistra) e F1-Macro (destra). RF SMOTE emerge chiaramente come best model.

6.2.2 Random Forest — Baseline

Tabella 10: Metriche per-class — RF Baseline su Validation Set

Classe	Precision	Recall	F1-Score	Support
Benign	0.9999	0.9981	0.9990	209,807
Bot	0.6128	0.9891	0.7568	184
BruteForce	1.0000	1.0000	1.0000	1,330
DDoS	0.9997	0.9998	0.9998	19,193
DoS	0.9983	0.9992	0.9988	28,394
Other	1.0000	0.7143	0.8333	7
PortScan	0.9693	0.9955	0.9822	222
WebAttack	0.5736	0.9834	0.7246	301
Macro Avg	0.8942	0.9599	0.9118	—

Interpretazione risultati:

RF Baseline raggiunge performance eccellenti su classi grandi (Benign, DDoS, DoS, BruteForce: F1 > 0.99). Le criticità sono:

- **Bot (precision 61.3%)**: Il modello genera molti falsi allarmi. Su 184 Bot reali, ne rileva 182 (recall 98.9%), ma in totale predice 297 "Bot", quindi 115 sono traffico Benign classificato erroneamente.
- **WebAttack (precision 57.4%)**: Simile a Bot: il modello rileva quasi tutti gli attacchi WebAttack (recall 98.3%), ma genera 220 falsi positivi da traffico Benign normale.

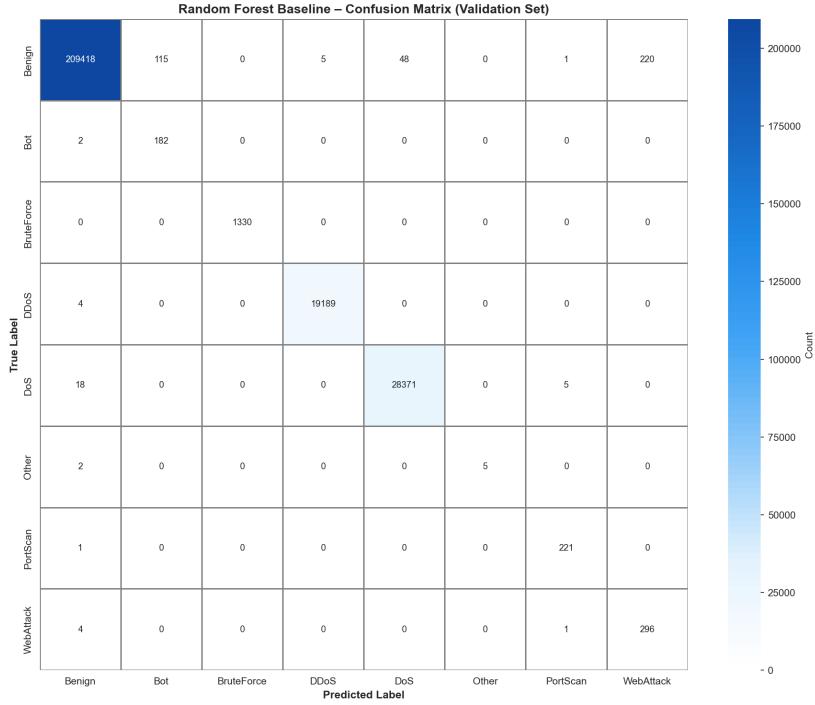


Figura 7: Confusion Matrix RF Baseline. Gli errori principali sono Benign→WebAttack (220 flussi normali classificati come attacco) e Benign→Bot (115).

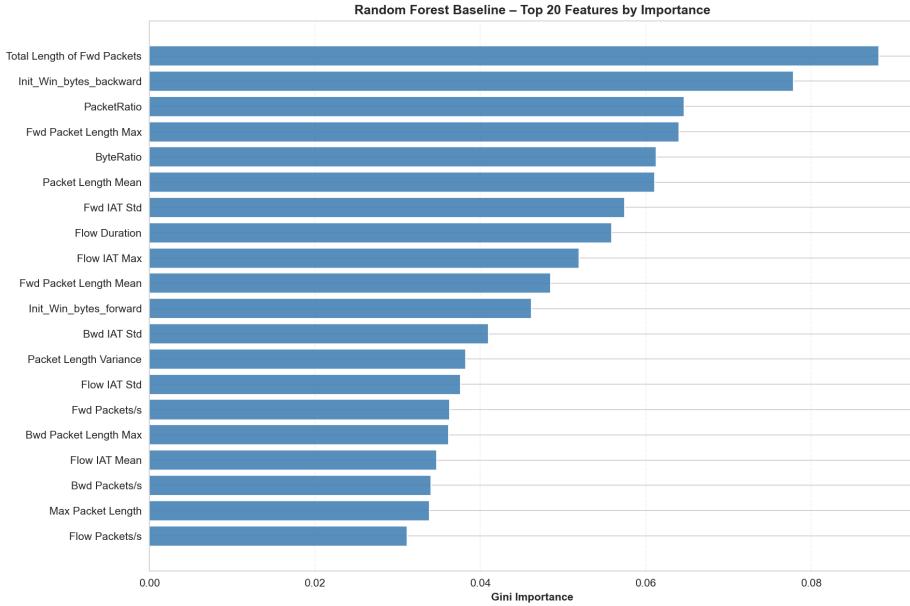


Figura 8: Feature Importance RF Baseline (top-20). Mostra quanto ogni feature sia importante nelle decisioni del modello. Le due feature ingegnerizzate PacketRatio (rank 3) e ByteRatio (rank 5) entrano nella top-5, confermando l’efficacia del feature engineering.

6.2.3 Random Forest + SMOTE (Best Model)

Tabella 11: Metriche per-class — RF + SMOTE su Validation Set

Classe	Precision	Recall	F1-Score	Support
Benign	0.9996	0.9986	0.9991	209,807
Bot	0.8688	0.7554	0.8081	184
BruteForce	1.0000	1.0000	1.0000	1,330
DDoS	0.9998	0.9995	0.9997	19,193
DoS	0.9986	0.9991	0.9988	28,394
Other	1.0000	0.7143	0.8333	7
PortScan	0.9910	0.9910	0.9910	222
WebAttack	0.5562	0.9867	0.7114	301
Macro Avg	0.9268	0.9305	0.9177	—

Confronto con Baseline: SMOTE migliora significativamente la detection sulla classe Bot, con precision che aumenta da 61% a 87% (+ 26%), riducendo i falsi allarmi da 115 a 21. Il trade-off è una diminuzione del recall da 99% a 76% (-23%): Quindi il modello rileva meno Bot, ma quando predice "Bot" è molto più accurato.

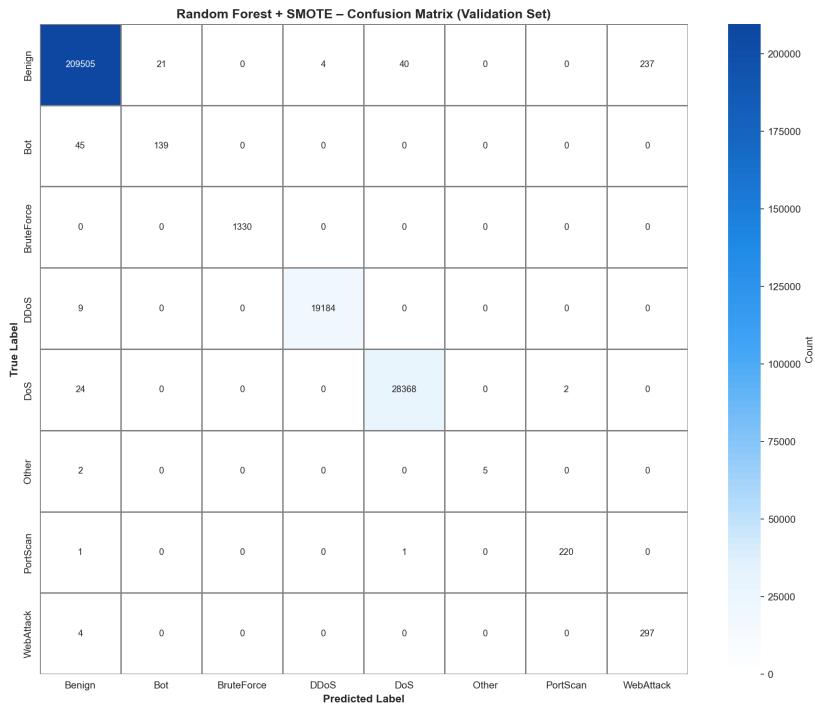


Figura 9: Confusion Matrix RF+SMOTE. Confrontando con la confusion matrix del modello baseline 7, si nota la drastica riduzione dei falsi positivi Benign→Bot: da 115 a 21 (-82%).

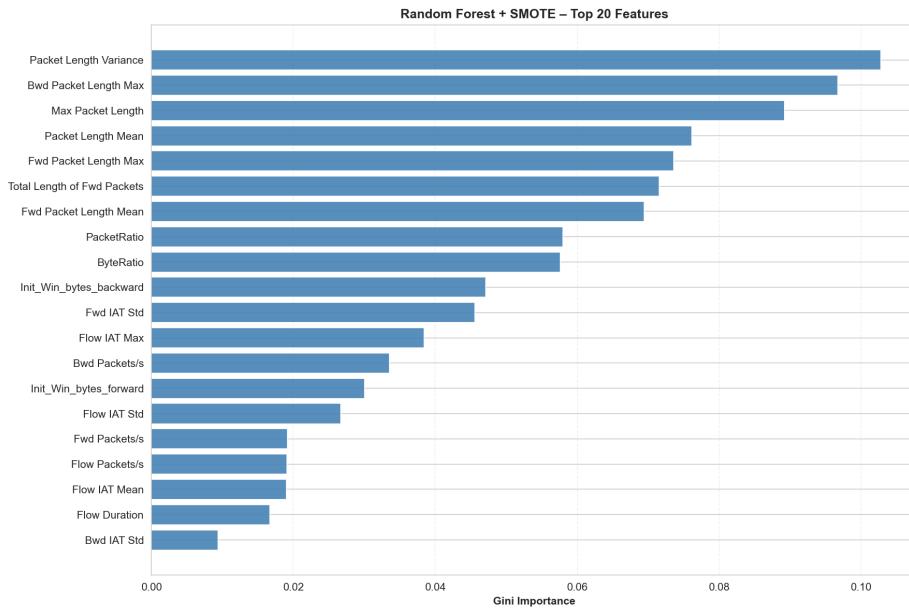


Figura 10: Feature Importance RF+SMOTE (top-20)

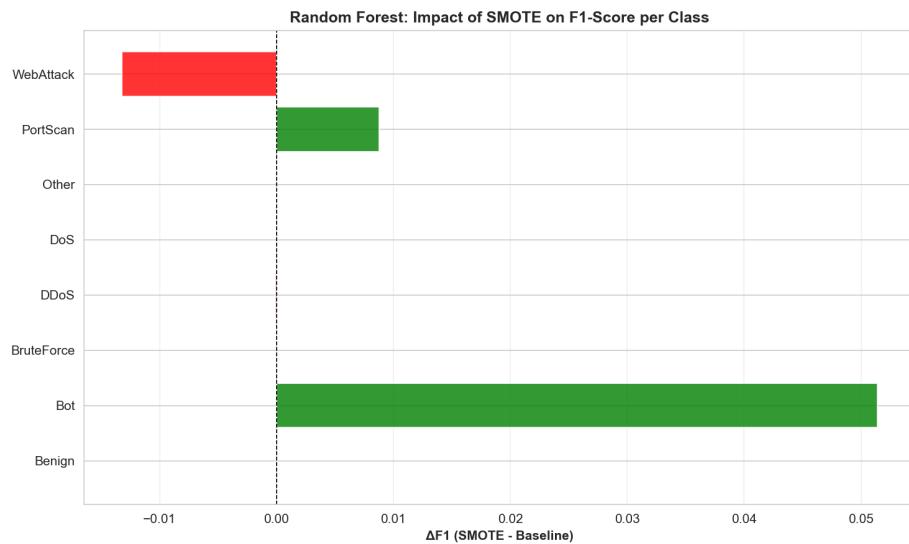


Figura 11: Variazione F1-Score per classe dopo applicazione di SMOTE. Bot mostra il maggior guadagno (+5.1%), WebAttack lieve degradazione (-1.3 %).

Tabella 12: Impatto di SMOTE su Random Forest: variazione F1 per-class (validation)

Classe	F1 Baseline	F1 SMOTE	$\Delta F1$ (punti %)
Bot	0.7568	0.8081	+5.1
PortScan	0.9822	0.9910	+0.9
WebAttack	0.7246	0.7114	-1.3
BruteForce	1.0000	1.0000	0.0
Other	0.8333	0.8333	0.0
Benign	0.9990	0.9991	+0.0
DDoS	0.9998	0.9997	-0.0
DoS	0.9988	0.9988	+0.0

6.2.4 LightGBM — Baseline

Tabella 13: Metriche per-class — LightGBM Baseline su Validation Set

Classe	Precision	Recall	F1-Score	Support
Benign	0.8818	0.4283	0.5766	209,807
Bot	0.0057	0.6250	0.0114	184
BruteForce	0.0000	0.0000	0.0000	1,330
DDoS	0.5533	0.5151	0.5335	19,193
DoS	0.6541	0.4624	0.5418	28,394
Other	0.0002	0.2857	0.0003	7
PortScan	0.0051	0.8919	0.0101	222
WebAttack	0.0019	0.2558	0.0037	301
Macro Avg	0.2628	0.4330	0.2097	—

Failure completo: LightGBM Baseline mostra un collasso su quasi tutte le classi:

- **Accuracy globale solo 43.66%:** Peggio di un classificatore casuale (che raggiungerebbe 12.5% su 8 classi uniformi).
- **Precision quasi zero** su Bot (0.6%), PortScan (0.5%), WebAttack (0.2%): il modello genera decine di migliaia di falsi allarmi.
- **BruteForce F1 = 0:** Nessun attacco BruteForce viene rilevato (su 1,330 attacchi reali, 0 predetti correttamente).
- **Benign F1 solo 0.58:** Il modello classifica erroneamente la metà del traffico normale come attacco, generando 120,000+ falsi positivi.

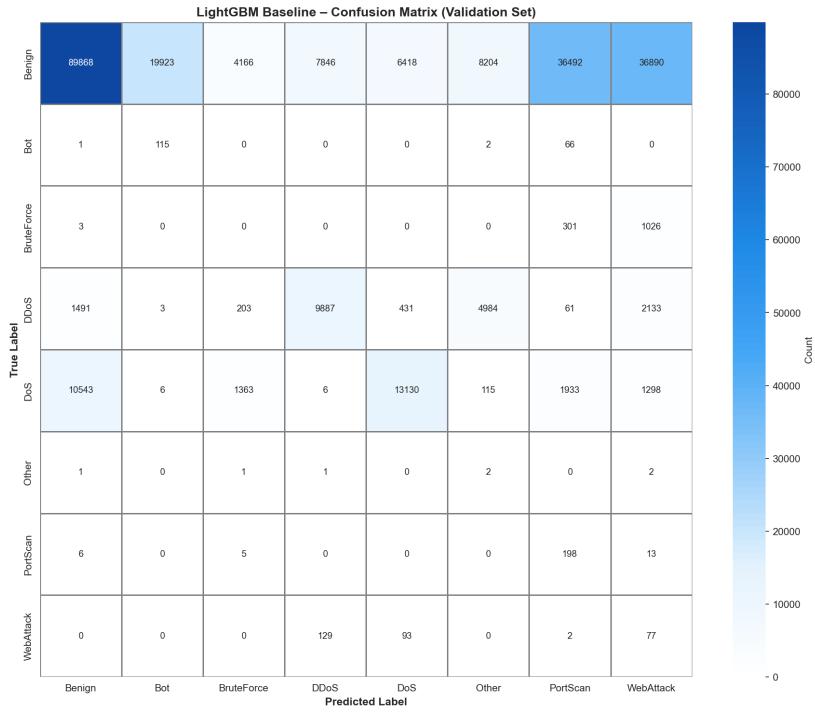


Figura 12: Confusion Matrix LGB Baseline. Predizioni completamente disperse: oltre 36,000 falsi e nessun BruteForce rilevato.

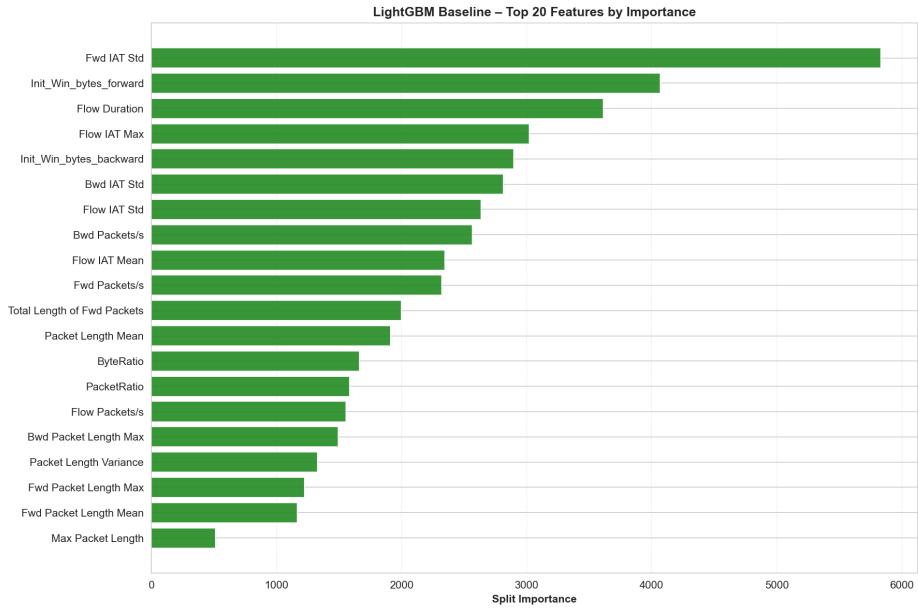


Figura 13: Feature Importance LGB Baseline. Nonostante l'importanza distribuita, il modello non riesce ad apprendere pattern discriminanti per le classi minoritarie.

6.2.5 LightGBM + SMOTE

Tabella 14: Metriche per-class — LightGBM + SMOTE su Validation Set

Classe	Precision	Recall	F1-Score	Support
Benign	0.9457	0.8200	0.8784	209,807
Bot	0.0000	0.0000	0.0000	184
BruteForce	0.0443	0.3586	0.0789	1,330
DDoS	0.5117	0.3450	0.4121	19,193
DoS	0.7447	0.7883	0.7659	28,394
Other	0.0000	0.0000	0.0000	7
PortScan	0.0002	0.0135	0.0003	222
WebAttack	0.0000	0.0000	0.0000	301
Macro Avg	0.2808	0.2907	0.2670	—

SMOTE migliora selettivamente solo 2 classi:

- **Benign:** F1 da 0.58 a 0.88 (+30 %) — il modello classifica abbastanza correttamente il traffico normale
- **DoS:** F1 da 0.54 a 0.77 (+23 %) — Migliorato anche su questa classe maggioritaria
- **Bot, Other, WebAttack:** F1 scende a 0 — Peggiorato invece su tutte le classi minoritarie, inclusa la classe DDoS non minoritaria.

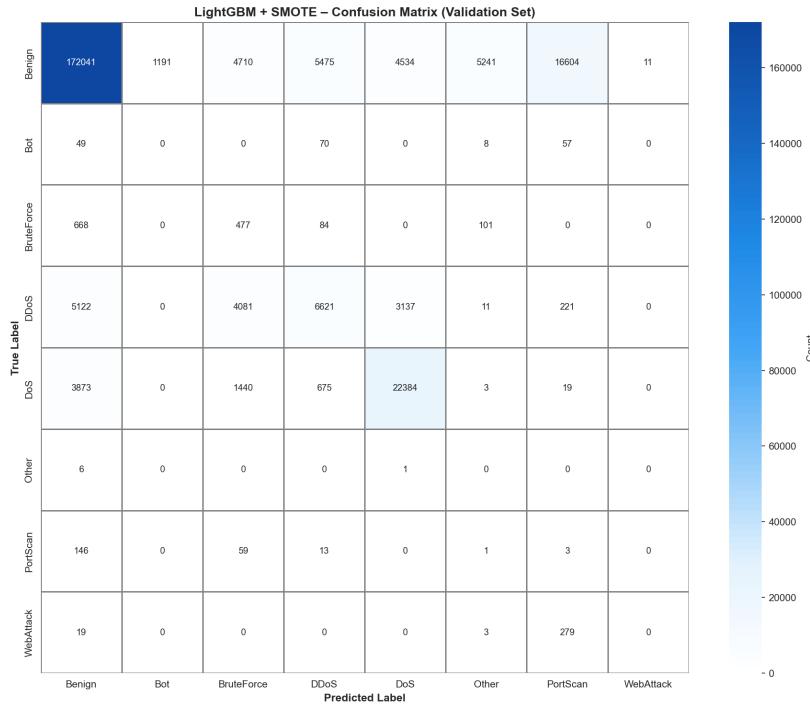


Figura 14: Confusion Matrix LGB+SMOTE. Miglioramento su Benign e DoS, ma ancora 16,600 falsi positivi Benign→PortScan.

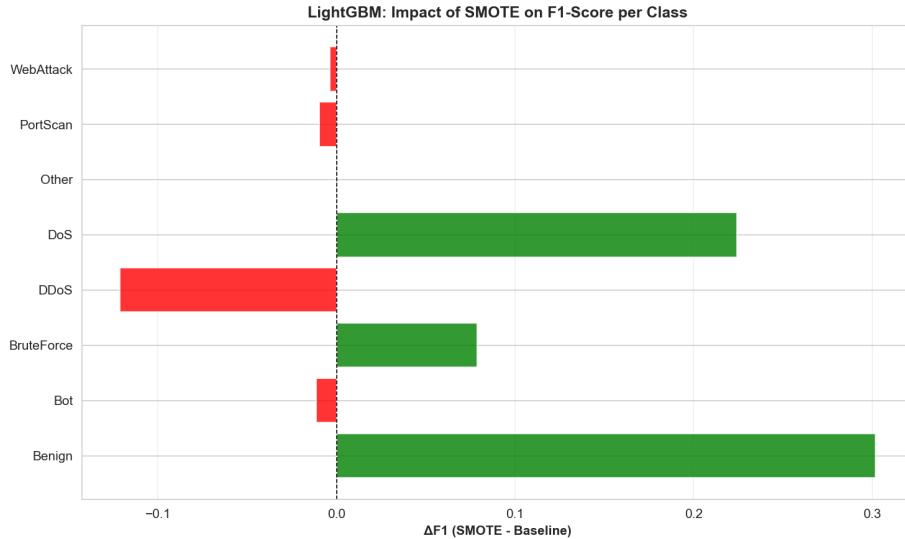


Figura 15: Variazione F1 LGB dopo SMOTE. Guadagni su Benign (+30 %) e DoS (+22 %), ma Bot/Other/WebAttack peggiorano o restano a 0.

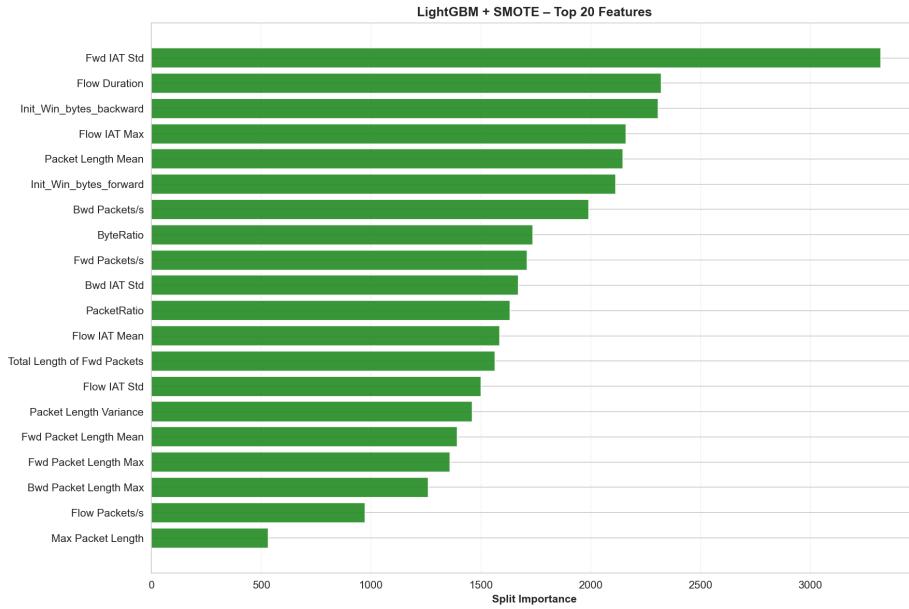


Figura 16: Feature Importance LGB+SMOTE. Il modello concentra l'importanza su feature per Benign e DoS, ignorando segnali per le classi piccole.

6.2.6 Multilayer Perceptron (Rete Neurale) — Baseline

Tabella 15: Metriche per-class — MLP Baseline su Validation Set

Classe	Precision	Recall	F1-Score	Support
Benign	0.9962	0.9970	0.9966	209,807
Bot	1.0000	0.6141	0.7609	184
BruteForce	0.9917	0.9925	0.9921	1,330
DDoS	0.9853	0.9983	0.9917	19,193
DoS	0.9867	0.9875	0.9871	28,394
Other	0.0000	0.0000	0.0000	7
PortScan	1.0000	0.7838	0.8788	222
WebAttack	0.0000	0.0000	0.0000	301
Macro Avg	0.7450	0.6716	0.7009	—

Pattern "dicotomico": MLP Baseline mostra ottimi risultati su classi grandi, ma fallimento totale su classi piccole:

- **Classi grandi ($F1 > 0.98$):** Benign, DDoS, DoS, BruteForce — quasi perfetto
- **Classi medie:** Bot (precision 100%, recall 61%), PortScan (precision 100%, recall 78%) — quando predice queste classi, precisione massima, ma ne perde molte

- **Classi piccole ($F1 = 0$):** Other (7 samples), WebAttack (301 samples) — nessun campione predetto correttamente

Motivazione: La rete neurale ottimizza una loss function (funzione di errore) dominata dalla classe Benign (80.87% del training set). Le classi piccole hanno un peso trascurabile nella loss totale, quindi la rete ignora questi pattern minoritari per concentrarsi su Benign.

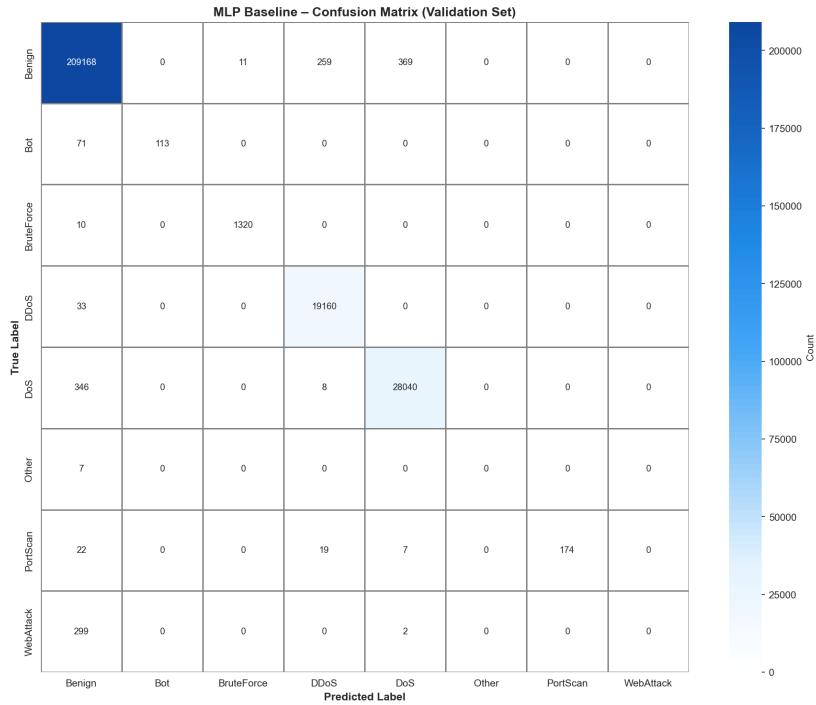


Figura 17: Confusion Matrix MLP Baseline. 299 su 301 WebAttack (99.3%) vengono classificati come Benign.



Figura 18: Training history MLP Baseline. La rete si ferma dopo 65 epoche per via dell'early stopping. La convergenza rapida su classi grandi non lascia tempo al modello di apprendere le minorities.

6.2.7 Multilayer Perceptron + SMOTE

Tabella 16: Metriche per-class — MLP + SMOTE su Validation Set

Classe	Precision	Recall	F1-Score	Support
Benign	0.9964	0.9970	0.9967	209,807
Bot	0.9826	0.6141	0.7559	184
BruteForce	0.9822	0.9947	0.9884	1,330
DDoS	0.9901	0.9980	0.9940	19,193
DoS	0.9846	0.9883	0.9865	28,394
Other	0.8000	0.5714	0.6667	7
PortScan	1.0000	0.8018	0.8900	222
WebAttack	1.0000	0.0598	0.1129	301
Macro Avg	0.9670	0.7531	0.7989	—

Effetto di SMOTE: Guadagno +9.80 % F1-Macro totale, grazie a:

- **Other:** Da F1=0 a F1=0.67 (+66.7 %) — con solo 31 campioni originali e 611 post-SMOTE (19.7× incremento), la rete riesce finalmente ad apprendere pattern per questa classe. Rileva 4 attacchi su 7 (recall 57%), con una precision del 80%.
- **WebAttack:** Da F1=0 a F1=0.11 (+11.3 %) — La recall sale da 0% a 6% (18 su 301 rilevati), ma con precision perfetta (1.0). Dunque quando MLP predice WebAttack, ha sempre ragione, ma lo fa troppo raramente.
- **Classi grandi invariate:** Benign, DDoS, DoS restano 0.99, confermando che SMOTE non introduce rumore dannoso per le classi maggioritarie.

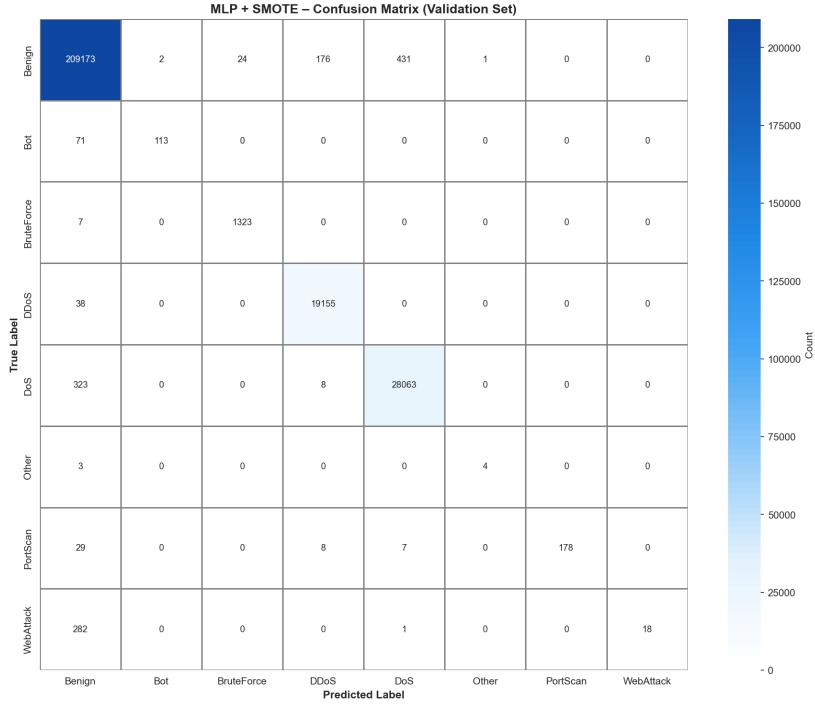


Figura 19: Confusion Matrix MLP+SMOTE. Miglioramento significativo rispetto a baseline.



Figura 20: Training history MLP+SMOTE. L'early stopping interviene a 62 epochhe (vs 65 baseline): il bilanciamento tramite SMOTE permette alla rete di convergere più rapidamente, raggiungendo una loss stabile in meno iterazioni grazie alla distribuzione più equilibrata delle classi.

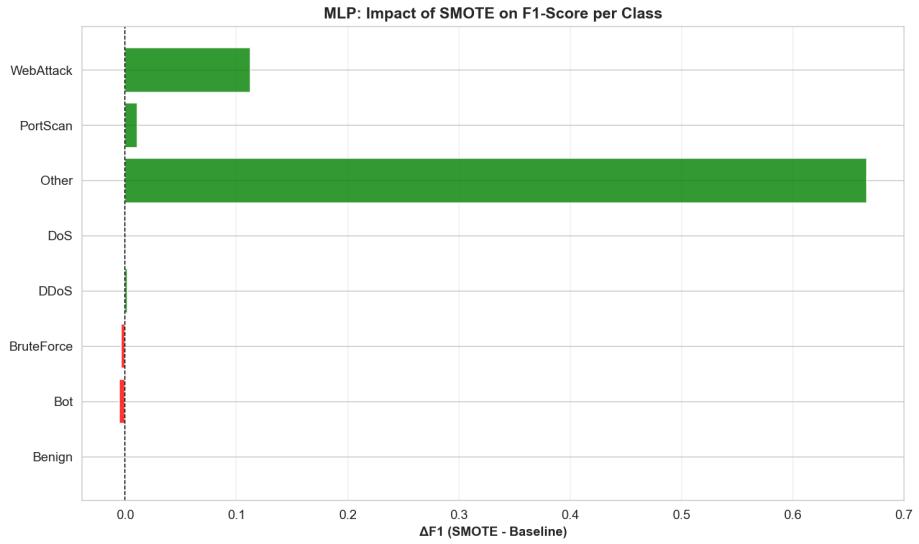


Figura 21: Variazione F1 MLP dopo SMOTE. Maggior guadagno su Other (+67 %) e WebAttack (+11 %), classi grandi invariate.

6.3 Test Finale ed elezione del Best Model

Tutti i 6 modelli candidati vengono valutati sul **test set** (259,395 campioni mai usati durante tutta la fase sperimentale) per calcolare il gap di generalizzazione validation-test. RF + SMOTE (F1-Macro validation: 0.9177) risulta confermato come best model anche su test set.

Tabella 17: Confronto Validation vs Test Set — gap di generalizzazione

Modello	Val Acc	Val F1	Test Acc	Test F1	Gap F1	Gap %
RF + SMOTE	0.9985	0.9177	0.9984	0.9043	0.0134	1.46%
RF Baseline	0.9984	0.9118	0.9984	0.8989	0.0129	1.41%
MLP + SMOTE	0.9946	0.7989	0.9948	0.7691	0.0298	3.73%
MLP Baseline	0.9944	0.7009	0.9944	0.6957	0.0052	0.74%
LGB + SMOTE	0.7768	0.2670	0.7746	0.2665	0.0005	0.19%
LGB Baseline	0.4366	0.2097	0.4385	0.2107	-0.0011	-0.50%

Gap di Generalizzazione: Analizziamo la differenza tra performance su validation e test set: Un gap piccolo (< 2%) indica che il modello generalizza bene su dati nuovi e non ha "imparato a memoria" il validation set. RF+SMOTE mostra gap ottimale: 1.46%.

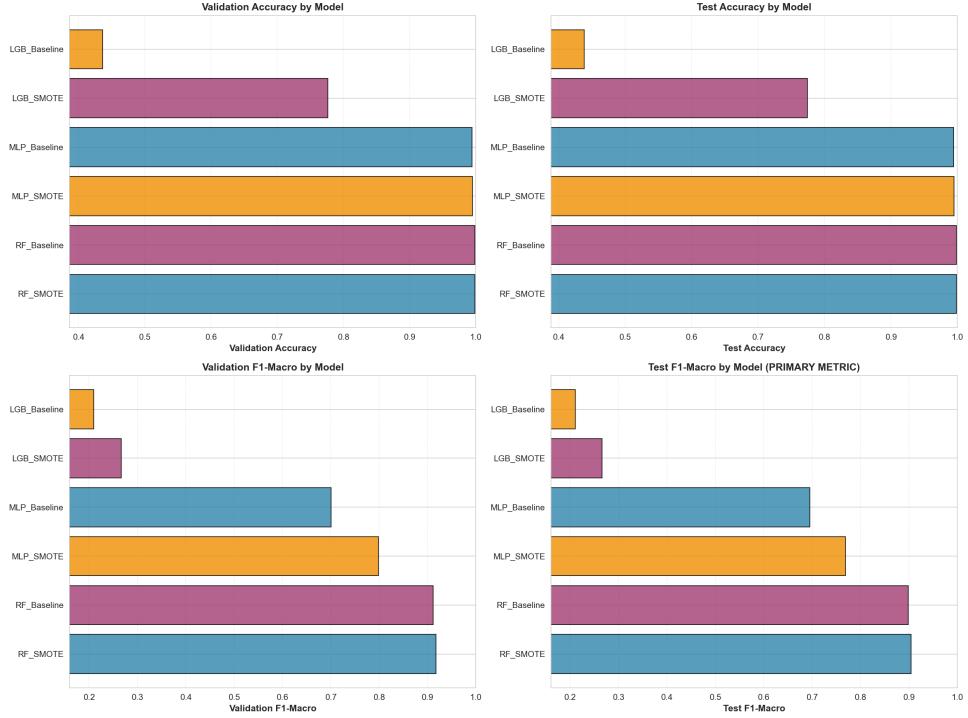


Figura 22: Confronto validation vs test per i 6 modelli. RF+SMOTE mantiene performance consistenti (gap 1.46%), indicando buona capacità di generalizzazione.

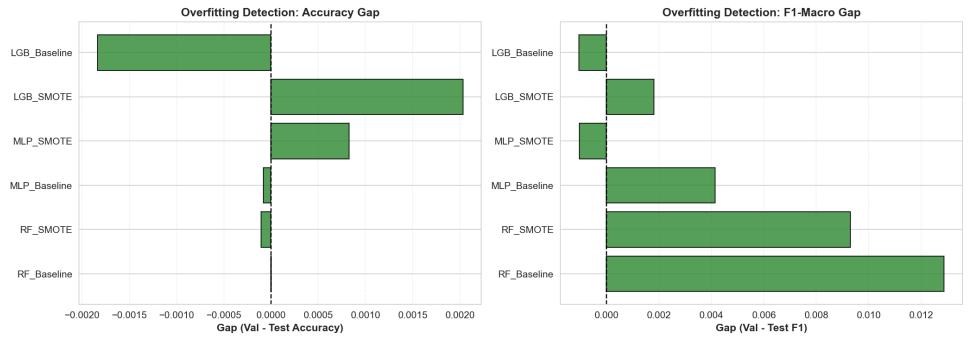


Figura 23: Analisi del gap di generalizzazione. RF e MLP+SMOTE hanno gap controllato (<4%). LGB mostra gap trascurabile, ma su performance inaccettabili ($F1 \sim 0.21-0.27$).

6.3.1 Metriche Per-Class su Test Set

Tabella 18: Metriche per-class — RF+SMOTE su Test Set (259,395 campioni)

Classe	Precision	Recall	F1-Score	Support
Benign	0.9996	0.9985	0.9991	209,773
Bot	0.8408	0.7174	0.7742	184
BruteForce	1.0000	0.9985	0.9992	1,330
DDoS	0.9999	0.9997	0.9998	19,189
DoS	0.9978	0.9992	0.9985	28,389
Other	0.8333	0.7143	0.7692	7
PortScan	0.9907	0.9640	0.9772	222
WebAttack	0.5636	0.9867	0.7174	301
Macro Avg	0.9032	0.9223	0.9043	—
Weighted Avg	0.9985	0.9984	0.9984	259,395

I risultati su test confermano il pattern osservato su validation, con coerenza eccellenza fra validation e test. Questo ci permette di escludere la presenza di overfitting.

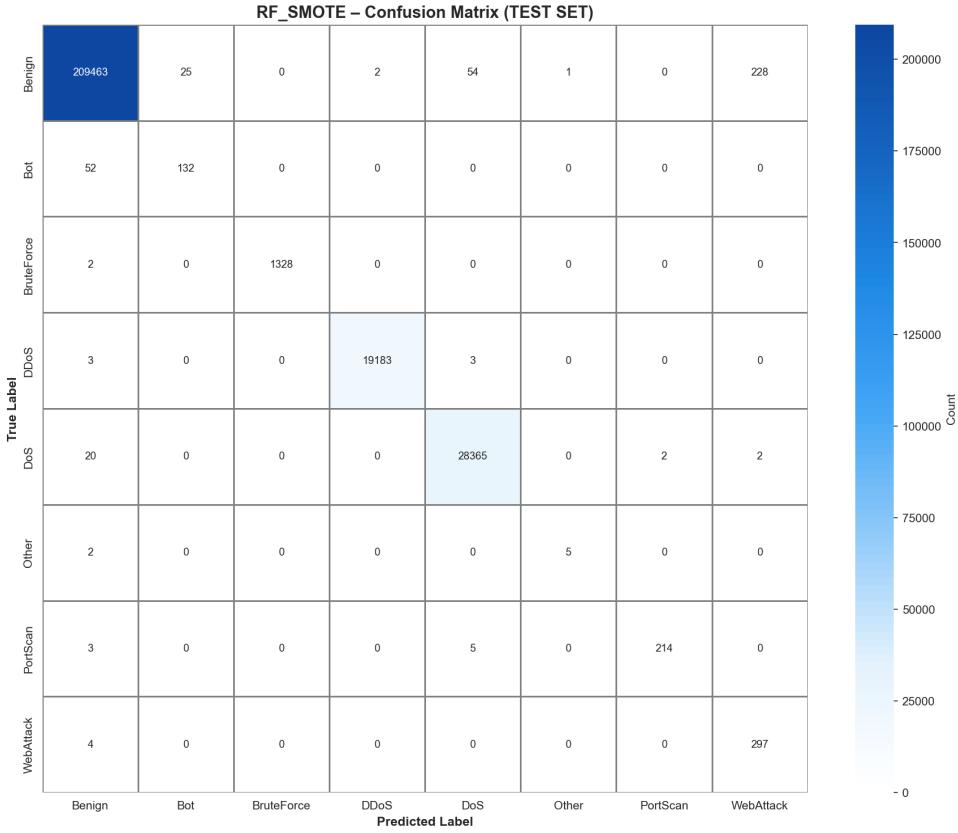


Figura 24: Confusion Matrix RF+SMOTE su Test Set (259,395 samples). La diagonale mostra elevata accuratezza su tutte le classi principali. L'errore più frequente è Benign→WebAttack (~230 campioni, 0.11% del traffico Benign). L'errore più impattante è Bot→Benign (52/184) e Other→Benign (2/7), entrambi ~28% del rispettivo traffico.

6.3.2 Analisi Errori

Tabella 19: Principali pattern di misclassificazione — RF+SMOTE Test Set

Classe Vera	Classe Predetta	Numero Errori
Benign	WebAttack	~230
Bot	Benign	~52
Benign	DoS	~40
DoS	Benign	~24
Benign	Bot	~19
DDoS	Benign	~6
Other	Benign	~2

Metriche aggregate errori (su 259,395 flussi test):

- **False Negatives (attacchi classificati come Benign):** 84 attacchi sfuggiti (principali pattern) su 49,622 attacchi totali = **0.17% FN rate**
- **False Positives (falsi allarmi):** ~310 allarmi falsi su 209,773 flussi Benign = **0.15% FP rate**

Interpretazione pratica: In una rete con 1 milione di flussi al giorno, questo si traduce in:

- ~17 attacchi sfuggiti al giorno (0.17%)
- ~1,500 falsi allarmi al giorno (0.15%)

Entrambi i valori sono considerati accettabili per un sistema IDS production, come discusso nella Sezione 7.4.

6.4 Confronto con Stato dell'Arte

L'approccio scelto, come descritto raggiunge Weighted F1 superiore (0.998 vs 0.97, +2.8 %) utilizzando solo **20 feature** invece di 80 (riduzione del 74.7%). Questo dimostra l'efficacia della consensus feature selection e del feature engineering come applicato nella Sezione 5.2. Inoltre il valore F1-Macro di 0.904 riflette la capacità del modello di rilevare anche le classi minoritarie, metrica non riportata nel baseline originale (che usava solo Accuracy e F1 Weighted).

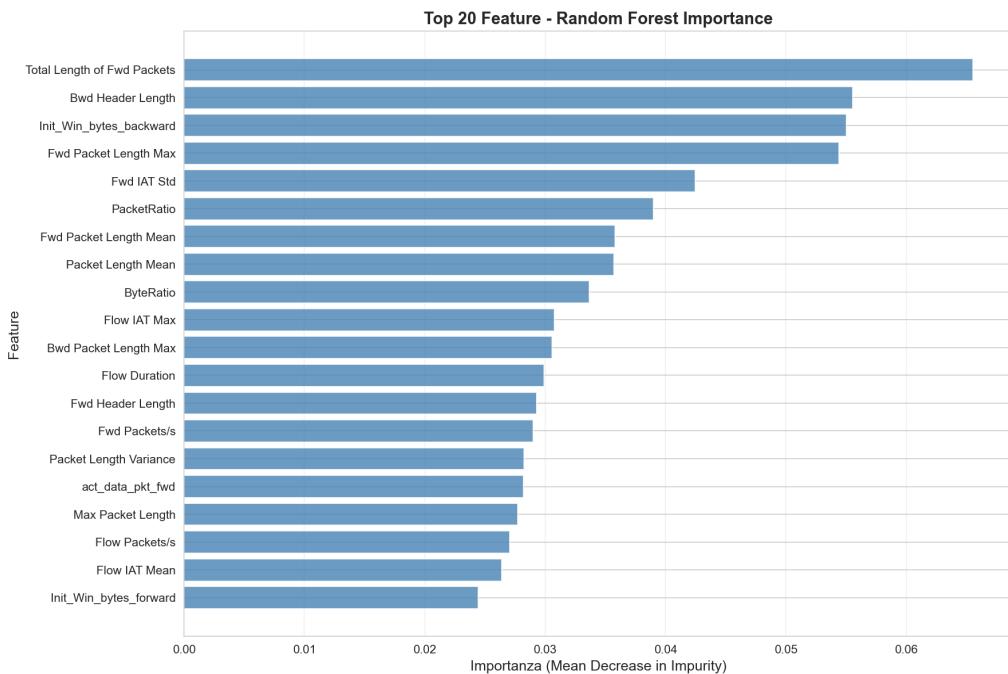


Figura 25: Top-20 feature per importanza nel best model RF+SMOTE. Le due feature ingegnerizzate (PacketRatio, ByteRatio) si posizionano nella top-10, confermando l'efficacia del feature engineering descritto nella Sezione 4.3.

7 Discussione

I risultati presentati nella Sezione 6.3 evidenziano differenze nette tra i tre algoritmi testati. In questa sezione si cerca di capire *perché* questi risultati sono venuti fuori in questo modo: perché Random Forest vince, perché LightGBM fallisce, come si comporta SMOTE a seconda del modello. Si analizzano poi i limiti strutturali che nessun algoritmo può superare senza cambiare approccio, si confronta la metodologia con quella di altri studi, e si discute cosa significherebbe usare questo sistema in un contesto reale.

7.1 Perché la superiorità di Random Forest

7.1.0.1 Decisioni locali invece che globali.

La caratteristica più importante di Random Forest, rispetto agli altri algoritmi testati, è che prende decisioni *locali* in ogni nodo dell'albero. Quando l'algoritmo deve decidere come separare i dati, guarda solo i campioni che sono arrivati fino a quel punto specifico dell'albero, non l'intero dataset.

Questo ha un risvolto importante: anche se la classe Bot rappresenta solo lo 0.07% del training set (860 campioni su oltre 1.2 milioni), un ramo profondo dell'albero può dedicarsi quasi esclusivamente a riconoscere quella classe, perché nei campioni che ci sono arrivati Bot può essere sovrarappresentata. Il parametro `class_weight='balanced'` amplifica ulteriormente questo effetto, assegnando pesi inversamente proporzionali alla frequenza di ciascuna classe durante l'addestramento.

7.1.0.2 Diversità tra gli alberi.

Random Forest costruisce 200 alberi decisionali diversi, ognuno addestrato su un subset casuale dei dati tramite bootstrap. Per effetto della casualità, alcuni alberi vedranno più campioni di Bot, altri più campioni di WebAttack, creando naturalmente una diversità che è utile per le classi rare.

C'è un secondo livello di casualità: ogni split dell'albero considera solo $\sqrt{20} = 4$ feature scelte casualmente (il default di sklearn per la classificazione è `max_features='sqrt'`) invece di tutte le 20. Questo riduce la correlazione tra gli alberi: se tutti gli alberi usassero le stesse feature, tenderebbero a fare gli stessi errori. La predizione finale è un voto a maggioranza tra i 200 alberi.

7.1.0.3 Robustezza allo sbilanciamento estremo.

Nonostante un rapporto Benign:Other di 31.577:1, Random Forest riesce comunque a mantenere un F1-Macro di 0.9177 sul validation set. Al contrario, MLP Baseline ottiene solo 0.7009 e fallisce completamente su alcune classi minoritarie ($F1 = 0$ per Other e WebAttack). Questo è perché non esiste una loss function condivisa tra tutti gli alberi che penalizzi di più gli errori sulla classe più numerosa.

7.1.1 Perché LightGBM Fallisce

LightGBM ha ottenuto risultati molto scadenti per una combinazione di fattori che si rinforzano a vicenda.

7.1.1.1 Boosting sequenziale

A differenza di Random Forest che costruisce alberi indipendenti in parallelo, LightGBM costruisce alberi *in sequenza*: ogni nuovo albero cerca di correggere gli errori del precedente. Il problema è che con uno sbilanciamento 31.577:1, gli errori sulla classe Benign dominano completamente il calcolo dell'errore totale.

7.1.1.2 Configurazione inadeguata.

I parametri utilizzati (`max_depth=25, learning_rate=0.1, n_estimators=200`) sono inadatti per questo problema. `max_depth=25` permette alberi troppo profondi che memorizzano pattern specifici della classe maggioritaria causando overfitting. Il `learning_rate` alto (0.1) fa sì che il modello corregga troppo ad ogni passo non raggiungendo mai il punto ottimale. Senza early stopping, il training continua per tutte le 200 iterazioni anche quando le performance sulle classi minoritarie peggiorano.

Gli autori di LightGBM raccomandano: profondità massima intorno a 7, learning rate sotto 0.05, e early stopping. In questo studio invece LightGBM è stato testato con parametri aggressivi, causando una degradazione delle performance. Implementare hyperparameter tuning come ad esempio ottimizzazione bayesiana avrebbe probabilmente prodotto risultati migliori e reso LightGBM più competitivo

7.1.1.3 Instabilità dei pesi con sbilanciamento estremo.

Per compensare lo sbilanciamento, `class_weight='balanced'` assegna pesi inversamente proporzionali alla frequenza. In LightGBM, ogni albero cerca di correggere gli errori dell'albero precedente, ma con questa differenza di pesi così estrema, data dall'imbalance estremo del dataset, gli alberi oscillano: uno corregge troppo verso Other, il successivo sovra-corregge verso Benign.

7.1.2 L'Effetto di SMOTE Varia tra gli Algoritmi

SMOTE è stato applicato seguendo la strategia progressive gap-filling descritta nella sezione 4.4.2.2. L'effetto di SMOTE sui tre modelli è molto diverso:

7.1.2.1 SMOTE su Random Forest:

Nel training set, Bot conta 860 campioni originali che diventano 982 dopo SMOTE, avendo un incremento solo del 1.1%. L'effetto però è misurabile: senza SMOTE, `class_weight='balanced'` rendeva Random Forest iper-sensibile verso Bot, con una recall del 98.9%, ma una precision solo del 61.3%. Dopo SMOTE, la recall scende a 75.5% ma la precision sale a 86.9%. L'F1-Score migliora del 5.1% perché il bilanciamento tra precision e recall diventa più equilibrato.

Su WebAttack F1 peggiora dell'1.3%: Questo accade perché WebAttack e traffico Benign HTTP sono già molto simili a livello di feature statistiche (come discusso nella Sezione 7.2), e l'interpolazione SMOTE non fa altro che creare campioni in quella zona di confine ambigua.

7.1.2.2 SMOTE su MLP:

MLP Baseline mostra $F1=0$ per Other e WebAttack, indicando che il modello non riesce proprio a riconoscerle. Il problema è che durante l'addestramento, la rete impara principalmente dagli errori. Con 978,899 campioni Benign e solo 31 campioni Other, la rete vede circa 31,000 errori su Benign per ogni 1 errore su Other. SMOTE risolve questo portando Other da 31 a 611 campioni ($19.7\times$). Ora la rete vede abbastanza errori su Other da capire che quella classe esiste. Risultato: F1 passa da 0 a 0.67 (+66.7 pp).

MLP con SMOTE viene addestrato per 62 epochhe invece di 65 (baseline). Con una distribuzione più bilanciata, l'early stopping interviene prima: la loss si stabilizza più rapidamente grazie alla maggiore rappresentazione delle classi minoritarie, che riduce le oscillazioni del gradiente.

7.1.2.3 SMOTE su LightGBM:

Dopo SMOTE, LightGBM migliora sulle classi maggioritarie: Benign ($F1: 0.58\rightarrow0.88$) e DoS ($F1: 0.54\rightarrow0.77$), ma Bot, Other e WebAttack peggiorano fino a $F1=0$. Anche con SMOTE, il gradient boosting alloca i 200 alberi preferenzialmente alle classi più numerose, questo perché Bot ha solo 982 campioni, ancora troppo pochi per avere un effetto sull'addestramento.

7.2 I Limiti dell'Approccio Basato su Flussi di Rete

Analizzando gli errori del modello migliore (sezione 6.3) emergono anche limiti che non dipendono dall'algoritmo usato, ma dall'approccio stesso.

7.2.1 WebAttack: Difficoltà a distinguere con Benign

Il problema più evidente è WebAttack: 237 falsi positivi (traffico normale classificato come attacco) con precision del 56%. Questo non è un limite del modello ma un limite intrinseco delle feature del dataset.

7.2.1.1 Il problema fondamentale.

Una richiesta HTTP POST legittima e una contenente SQL injection sono statisticamente quasi identiche a livello di flusso di rete: stessa grandezza di pacchetti, stessa dimensione del payload, stesso tempo di risposta. CICFlowMeter estrae statistiche aggregate (media, deviazione standard, massimo) senza accedere al contenuto dei pacchetti.

7.2.1.2 Cosa servirebbe.

Per distinguere i due casi bisognerebbe analizzare il payload HTTP, cercando pattern come SQL keywords (`SELECT`, `DROP TABLE`) o XSS payloads (`<script>` eseguibili nelle richieste). Questa tecnica ha però limiti pratici importanti: è computazionalmente costosa e solleva problemi di privacy legati al GDPR.

7.2.2 Bot: Difficoltà ad individuare la classe

La classe Bot ha una recall del 72%, questo perché i bot moderni sono progettati per essere difficili da rilevare.

7.2.2.1 Mimetizzazione

Il bot utilizzato nel dataset genera traffico progettato per sembrare normale ed evita pattern fissi, rendendolo quasi indistinguibile da una normale sessione web.

Il pattern più caratteristico di una botnet è il *beaconing*: il bot contatta periodicamente il C&C server. Il problema è che CICFlowMeter aggrega statistiche su singoli flussi TCP di durata 1–10 secondi, mentre il pattern di beaconing emerge solo analizzando decine di minuti o ore di traffico: una connessione ogni due minuti su un singolo flusso sembra normalissima, ma su una finestra di 24 ore diventa un pattern statisticamente anomalo.

7.2.3 Efficacia IDS per classe:

Tabella 20: Efficacia della detection flow-based per categoria di attacco (RF+SMOTE, test set)

Categoria	F1 Test	Rilevabilità	Perché
DDoS/DoS	> 0.99	Ottima	Volume anomalo
BruteForce	> 0.99	Ottima	Molte connessioni fallite
PortScan	0.98	Buona	Molte porte contattate
Bot C&C	0.77	Media	Si mimetizza
WebAttack	0.72	Media	Richiede analisi del payload (DPI)
Other (Infiltration+Heartbleed)	0.77	Limitata	Attacchi multi-fase con footprint distribuito

L'approccio flow-based funziona bene su attacchi volumetrici e ripetitivi (DDoS, BruteForce, PortScan) ma va complementato con misure come signature-based intrusion detection per WebAttack e behavioral analytics per Bot.

7.3 Confronto con Altri Studi

Le scelte metodologiche adottate in questo studio sono state in larga parte motivate da problemi documentati in letteratura sul dataset CIC-IDS-2017 e sull'applicazione di ML agli IDS in generale.

7.3.1 Feature Topologiche

Catillo et al. [2] dimostrano che modelli addestrati su CIC-IDS-2017 con feature topologiche (IP sorgente, IP destinazione, porta di destinazione) ottengono performance eccellenti sul dataset originale ma degradano significativamente su reti diverse, perché il modello impara a riconoscere gli indirizzi specifici del testbed invece delle caratteristiche statistiche del traffico malevolo. In questo studio la feature Destination Port è stata rimossa a priori, seguendo questa indicazione.

7.3.2 Scelta delle Metriche di Valutazione

Sharafaldin et al. [8], autori del dataset CIC-IDS-2017, valutano il loro classificatore Random Forest usando F1-Weighted e Accuracy globale, ottenendo F1-Weighted = 0.97 su 80 feature. Come discusso nella Sezione 3, queste metriche sono dominate dalla classe Benign. Lo possiamo vedere dai risultati LightGBM Baseline: Accuracy = 97.5% ma F1-Macro = 62.1%,

Per questo motivo F1-Macro è stata adottata come metrica primaria: assegna peso uguale a tutte le 8 classi indipendentemente dal loro supporto, ed è l'unica metrica che non nasconde fallimenti su attacchi rari. I risultati sono accompagnati da tabelle per-class complete (Sezione 6.2).

Con sole 20 feature, RF+SMOTE raggiunge F1-Weighted = 0.998, superiore al baseline di Sharafaldin et al. [8] che è del 97%.

7.3.3 Applicazione Corretta di SMOTE

Chawla et al. [3] nella formulazione originale di SMOTE, si mostra come questo genera campioni sintetici interpolando tra i k-nearest neighbors di campioni reali. Se i campioni del test set partecipano a questa interpolazione, la separazione tra training e test set viene invalidata.

Per evitarlo, SMOTE è applicato esclusivamente dopo lo split, solo ai training fold durante la cross-validation, mai al validation o al test set (Sezione 4.4.2.2).

7.3.4 Split Train/Validation/Test

Engelen et al. [4] evidenziano come molte delle anomalie nel dataset CIC-IDS-2017 come connessioni TCP troppo brevi, duplicati, feature con valori infiniti se ignorate portano a stime di performance troppo ottimistiche.

In questo studio oltre a gestire le anomalie del dataset, viene effettuato uno split 70/15/15 con un validation set che viene usato per l'elezione del modello migliore tra i 6 candidati. Il test set viene utilizzato esclusivamente dopo la model selection per valutare tutti i 6 modelli e calcolare il gap di generalizzazione, senza mai influenzare le decisioni di training o selezione. Il gap validation-test di 1.46% per RF+SMOTE (Sezione 6.3) conferma che il modello non ha overfittato il validation set durante la selezione.

7.4 Applicabilità in Ambiente Reale

7.4.1 Costo degli errori

Il modello migliore commette ~ 84 falsi negativi principali (attacchi classificati come Benign) e ~ 310 falsi positivi (falsi allarmi) sul test set.

7.4.1.1 Falsi negativi: ~ 84 attacchi sfuggiti (0.17%).

Dei 49,622 attacchi nel test set, circa 84 vengono classificati come Benign nei principali pattern di errore. Il caso più preoccupante sono i 52 Bot non rilevati, in quanto un bot compromesso può esfiltrare dati senza che il sistema se ne accorga. Gli altri 24 sono attacchi DoS, facilmente rilevabili e gestibili con timeout sul WAF, e 2 sono Other (Infiltration/Heartbleed), siccome ci sono 7 campioni totali, è una categoria difficilmente gestibile.

7.4.1.2 Falsi positivi: ~ 310 falsi allarmi (0.15%).

Su 209,773 flussi normali, circa 310 vengono classificati erroneamente come attacchi. Su una rete con 1 milione di flussi al giorno otteniamo $\sim 1,500$ falsi allarmi al giorno, circa 63 all'ora.

7.4.2 Architettura a Più Livelli

Il classificatore flow-based da solo non basta. Basandosi sui limiti discussi nella Sezione 7.2, funziona meglio come primo filtro in un'architettura a 3 livelli dove ogni livello compensa i limiti del precedente.

7.4.2.1 Livello 1: Flow-based IDS

Un IDS che Analizza il 100% del traffico in tempo reale. È efficace su attacchi volumetrici (DDoS, DoS, BruteForce, PortScan) dove raggiunge $F1 > 0.97$. Il vantaggio principale è la velocità e il peso del RF, che è facilmente caricabile interamente in RAM su qualsiasi hardware moderno senza overhead di I/O.

7.4.2.2 Livello 2: Signature-based IDS

Analizza solo i flussi marcati come sospetti dal Livello 1. Ad esempio, per WebAttack, guardando il contenuto del payload HTTP, può cercare pattern malevoli esatti (tipo `DROP TABLE`, `<script>`) raggiungendo precision superiore al 56% del Flow-Based

7.4.2.3 Livello 3: Behavioral Analytics

Aggrega statistiche su finestre lunghe (1–24h) per rilevare pattern temporali come il beaconing dei Bot. Questo approccio richiede un database che traccia lo storico dei flussi per IP/dominio, richiede un'implementazione più complessa e costosa, ma è necessario per attacchi che si mimetizzano nel traffico normale.

8 Conclusioni e Sviluppi Futuri

8.1 Riepilogo dei risultati

Questa relazione ha presentato una pipeline per classificare il traffico di rete sul dataset CIC-IDS-2017, confrontando tre algoritmi: Random Forest, LightGBM, MLP in versione baseline e con SMOTE, affrontando i problemi metodologici documentati in letteratura.

Il modello migliore, RF+SMOTE, raggiunge sul test set F1-Macro 0.904 e F1-Weighted 0.998 con un gap validation-test del 1.46%, segno che il modello generalizza senza overfitting.

Il lavoro presenta quindi un miglioramento delle prestazioni rispetto al baseline di Sharafaldin et al. [8] usando solo 20 feature invece di 80. Tuttavia c'è da dire che nel paper originale la classificazione avveniva su 15 classi mentre in questo lavoro le classi di attacco sono state aggrate in 8 macro-categorie.

Gli obiettivi della Sezione 1 sono stati raggiunti: l'analisi esplorativa ha identificato e rimosso gli artefatti documentati da Engelen et al. [4], riducendo il dataset da 2,8 milioni a 1,7 milioni di flussi. La feature selection ha ridotto le dimensioni da 79 a 20 feature (riduzione del 74.7%) senza perdere capacità discriminativa. Il confronto tra 6 configurazioni ha mostrato che Random Forest è robustamente superiore, che SMOTE ha effetti molto diversi a seconda dell'algoritmo, e che LightGBM senza hyperparameter tuning non è competitivo su dati fortemente sbilanciati.

8.2 Limiti del Lavoro

I risultati vanno interpretati tenendo conto di alcune limitazioni.

8.2.0.1 Dataset datato.

CIC-IDS-2017 è del 2017, generato in laboratorio. Il traffico reale di una rete aziendale ad oggi è molto diverso: molto più HTTPS, traffico cloud e da dispositivi IoT. Dunque non è garantito che un modello addestrato su questi dati funzioni altrettanto bene su traffico reale contemporaneo.

8.2.0.2 LightGBM non ottimizzato.

Come discusso nella Sezione 7.1, LightGBM è stato valutato con una configurazione aggressiva non adatta a dataset fortemente sbilanciati. I risultati scadenti sono in parte dovuti a questa scelta, in quanto con hyperparameter tuning LightGBM avrebbe potuto essere maggiormente competitivo.

8.2.0.3 Limiti strutturali del flow-based.

come discusso nella Sezione 7.2, i problemi su WebAttack e Bot richiedono approcci complementari per essere risolti.

8.2.0.4 Nessun test su rete reale.

Tutti i risultati sono misurati su un test set estratto dallo stesso dataset di training. Non è stato testato su traffico reale, che sarebbe il vero caso d'uso del modello.

8.3 Sviluppi Futuri

Dall'analisi sulle limitazioni possiamo individuare alcune direzioni concrete per gli sviluppi futuri:

La prima è l'ottimizzazione degli iperparametri di LightGBM con ricerca bayesiana. LightGBM con i parametri giusti potrebbe essere competitivo, soprattutto per la sua efficienza, specialmente in termine di prestazioni, su dataset massivi.

La seconda direzione è la validazione su dataset più recenti come CIC-IDS-2018 e NF-UQ-NIDS (2023) che hanno caratteristiche diverse e verificherebbero quanto i pattern appresi siano generalizzabili.

Infine, tecniche di explainability come SHAP renderebbero le predizioni comprensibili agli operatori SOC.

Riferimenti bibliografici

- [1] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.
- [2] Mirko Catillo, Antonio Del Vecchio, Antonio Pecchia, and Umberto Villano. Transferability of machine learning models learned from public intrusion detection datasets: the cicids2017 case study. *Software Quality Journal*, 30:955–981, 2022.
- [3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
- [4] Gints Engelen, Vera Rimmer, and Wouter Joosen. Troubleshooting an intrusion detection dataset: the cicids2017 case study. *IEEE Security & Privacy*, 19(6):7–11, 2021.
- [5] Mohamed Amine Ferrag, Leandros Maglaras, Sotiris Moschoyiannis, and Helge Janicke. Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. *Journal of Information Security and Applications*, 50:102419, 2020.
- [6] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*, volume 30, 2017.
- [7] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167, 2019.
- [8] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, pages 108–116, 2018.