

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

Sperimentazione di tecniche di Machine Learning e Deep Learning per la previsione di Job Zombie in sistemi HTC

Relatore:

Prof. Moreno Marzolla

Presentata da:

Alessio Arcara

Correlatore:

Dott. Stefano Dal Pra

Seconda Sessione di Laurea
Anno Accademico 2022 - 2023

*Alla memoria di mio padre e mia madre,
cui dedico ogni mio traguardo.*

Sommario

Il CNAF gestisce un centro di calcolo dotato di oltre 46000 core distribuiti su 960 host fisici. I job vengono accodati e schedulati dal sistema batch (HTCondor) attraverso l'uso di algoritmi di "fairshare". Durante l'esecuzione, vengono monitorate grandezze quali il consumo di memoria e lo spazio su disco, che vengono campionate ogni tre minuti e raccolte in un database insieme ai dati di accounting relativi ai job terminati. Questi job possono variare notevolmente in termini di durata, da pochi minuti a più giorni. Questo studio esplora l'uso di tecniche di Machine Learning per prevedere il successo o il fallimento dei job, basandosi sull'evoluzione del loro stato nel tempo. In particolare, ci si è concentrati sui cosiddetti "job zombie", ossia quei job che, pur terminando, non rilasciano l'host fisico, causando una perdita di risorse fino al loro timeout. Un approccio iniziale, che prendeva in considerazione solo la prima ora di vita del job, ha permesso di identificare con un'accuratezza del 72% la classe meno rappresentata (i job zombie). Per migliorare l'accuratezza, si è preso in considerazione l'intero primo giorno, applicando tecniche di Deep Learning sia supervisionate (CNN, CNN+LSTM, LSTM e Transformer) che non supervisionate (autoencoder e variational autoencoder). Nonostante l'incremento della complessità dei modelli, le reti neurali hanno mostrato una tendenza all'overfitting a causa dell'estremo sbilanciamento dei dati.

Indice

Sommario	i
1 Caso di studio	1
1.1 Il cluster di calcolo del CNAF	1
1.2 La base di dati	3
1.3 Motivazione	4
2 Analisi del database	9
2.1 Analisi esplorativa	9
2.1.1 Caratterizzazione dei job	9
2.1.2 Caratterizzazione dei gruppi	12
2.1.3 Analisi dei nomi dei job	15
2.2 Job Zombie Prediction	16
3 Tecniche di Machine Learning e Deep Learning	23
3.1 Estrazione dei dati	25
3.2 Preparazione dei dati	26
3.2.1 Preparazione delle serie storiche	28
3.2.2 Creazione delle feature	30
3.2.3 Etichettatura dei dati	31
3.2.4 Tecniche di bilanciamento dei dati	32
3.3 Selezioni dei modelli	34
3.3.1 XGBoost (Classificatore)	34
3.3.2 Reti neurali (Classificatore)	37

3.3.3 Autoencoder (Novelty detection)	39
4 Analisi dei risultati	43
4.1 Valutazione delle performance	43
4.1.1 Metriche di valutazione	43
4.1.2 Convalida incrociata	43
4.2 Confronto tra i modelli	43
4.3 Interpretazione dei risultati	43
5 Conclusioni e sviluppi futuri	45
5.1 Sintesi dei risultati	45
5.2 Limitazioni dello studio e proposte per ricerche future	45
Bibliografia	47

Elenco delle figure

1.1	Struttura gerarchica del WLCG [10]	2
1.2	Media giornaliera di job sottomessi e falliti nel mese di Marzo 2023	7
2.1	Rappresentazione di un job come serie storica multivariata	11
2.2	Frequenza di campionamento e aggiornamento dei job in HTCondor	11
2.3	Distribuzione della durata dei job in giorni	12
2.4	A sinistra, durata cumulativa dei job; a destra, numero totale di job e relativi fallimenti per ogni fascia oraria fino a 48 ore, mostrati su scala logaritmica	13
2.5	Distribuzione dei job nella prima ora, suddivisi in intervalli di cinque minuti	13
2.6	Distribuzione della durata dei job per gruppo	14
2.7	Distribuzione del numero totale dei job e dei loro fallimenti per gruppo	15
2.8	Rappresentazione dei job in base alla loro durata	17
2.9	Utilizzo di RAM, SWAP e disco su intervalli di 15 minuti nelle prime 24 ore	19
2.10	Struttura generica di un autoencoder [14]	20
2.11	Visualizzazione job zombie del 2021 tramite t-SNE	21
2.12	Visualizzazione job ATLAS di settembre 2021 tramite t-SNE	21
3.1	Rappresentazione di una pipeline di Machine Learning come un grafo orientato senza cicli	24
3.2	Le prime cinque righe del dataset	26
3.3	Diagramma UML della classe <code>Preprocessor</code> , illustrante l'implementazione del design pattern Template Method	27
3.4	Rappresentazione tridimensionale delle multiple serie storiche multivariate	29
3.5	Visualizzazione delle nuove colonne <code>job type</code> e <code>job work type</code>	31

3.6 Rappresentazioni compresse in un autoencoder tradizionale e un variational autoencoder [25]	33
3.7 Rappresentazione ideale di una serie storica anomala in contrasto con un cluster di serie storiche normali	35
3.8 Struttura semplificata di XGBoost [15]	36
3.9 Struttura di una rete neurale con uno strato nascosto	37
3.10 Struttura di un neurone	38
3.11 Visualizzazione delle connessioni tra i neuroni in uno strato convoluzionale	39
3.12 Estrazione delle feature da parte di una rete neurale convoluzionale [16] . .	39
3.13 Visualizzazione dell'errore di ricostruzione e della soglia per stabilire se gli esempi sono novità	40

Elenco delle tabelle

1.1	Confronto delle dimensioni tra i database <code>htm</code> e <code>htmnew</code>	4
1.2	Schema della tabella <code>hj</code> del database	5
1.3	Schema della tabella <code>htjob</code> del database	6
2.1	Percentuale di job in attesa rimossi senza aver effettuato alcun calcolo su un host fisico	12
2.2	Frequenza dei nomi dei job	16
2.3	Rapporto tra i job zombie e il totale dei job per ciascun gruppo	17

Capitolo 1

Caso di studio

In questo capitolo presenteremo una panoramica del centro di calcolo presso il quale è stato svolto il tirocinio, guardando da dove provengono i dati e come vengono raccolti. Verranno infine presentati gli obiettivi di questo studio e della tesi che ne deriva.

1.1 Il cluster di calcolo del CNAF

Il **grid computing** è un’architettura di calcolo distribuito che collega computer sparsi geograficamente allo scopo di condividere risorse e potenza di calcolo per raggiungere uno scopo condiviso. Attualmente, il più grande sistema grid al mondo è il Worldwide LHC Computing Grid (WLCG), che nasce da una collaborazione internazionale che coinvolge oltre 170 centri di calcolo sparsi in più di 40 nazioni. Lo scopo del WLCG è fornire l’infrastruttura computazionale necessaria per gestire i dati generati dagli esperimenti effettuati con il Large Hadron Collider (LHC) [7].

Come illustrato nella figura 1.1, i centri di calcolo all’interno del WLCG sono strutturati secondo il modello MONARC, che li organizza in un sistema gerarchico di livelli, noti come Tier, ciascuno dei quali ha funzioni e responsabilità ben definite. In questo contesto si colloca il centro nazionale delle tecnologie informatiche e telematiche (CNAF), che ospita il Tier-1 per tutti e quattro gli esperimenti del LHC. Oltre a questi ultimi, vengono supportati presso il CNAF anche gli esperimenti non-LHC di astrofisica delle particelle e fisica dei neutrini [3].

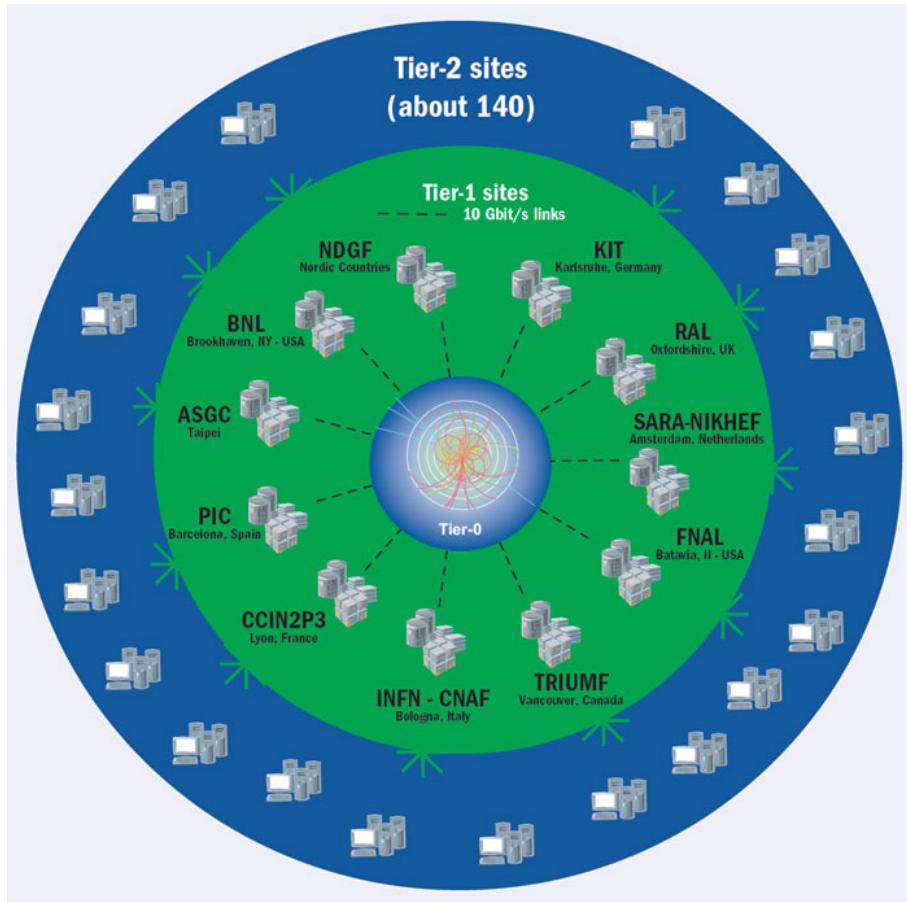


Figura 1.1: Struttura gerarchica del WLCG [10]

Il CNAF offre più di 46000 core distribuiti su 960 host fisici per un totale di circa 630 kHS06¹ di potenza di calcolo [24]. L'allocazione di queste risorse segue il paradigma del **High-Throughput Computing** (HTC), dove a differenza dell'High-Performance Computing, che mira a ridurre il tempo di esecuzione dei programmi, l'obiettivo dell'HTC è massimizzare il throughput, intenso come il numero di job completati per unità di tempo.

In questo sistema, gli utenti sono raggruppati in circa 50 gruppi distinti, ciascuno dei quali corrisponde a un esperimento scientifico specifico. A ogni gruppo è assegnata una quota di risorse che può utilizzare. Quando un utente ha bisogno di utilizzare queste risorse, può sottomettere un **job** al sistema, che rappresenta una o più operazioni

¹metrica per misurare le prestazioni della CPU, sviluppata dal gruppo di lavoro HEPiX. È utilizzata per confrontare le risorse di calcolo in ambito scientifico.

computazionali.

Una volta sottomesso, il job non viene eseguito immediatamente, ma viene messo in una coda gestita da un batch system (HTCondor). Quest'ultimo è responsabile della schedulazione dei job in coda, decidendo quale job eseguire, quando e dove. Per farlo, utilizza algoritmi di “fairshare”, che sono pensati per assicurare una distribuzione equa delle risorse computazionali disponibili, impedendo che un singolo utente o un intero gruppo possa monopolizzare tutte le risorse disponibili.

Se un gruppo non utilizza la quota di risorse assegnata, queste vengono redistribuite tra i gruppi attivi in proporzione alla loro quota. Questo meccanismo assicura che la farm di calcolo lavori quasi sempre alla sua massima capacità, ottimizzando l'uso delle risorse nel lungo termine [9].

1.2 La base di dati

Il caso di studio di questa tesi si basa su informazioni provenienti da due fonti principali: la prima è ottenuta attraverso il monitoraggio dei job in esecuzione, effettuato tramite il comando `condor_q` di HTCondor, eseguito ogni 3 minuti. La seconda proviene dai file *history*, generati automaticamente da HTCondor al termine dell'esecuzione di ciascun job.

Successivamente uno script estrae le informazioni rilevanti dai dati di accounting; queste informazioni vengono poi inserite nella tabella `htjob` di un database PostgreSQL. Analogamente, i dati di monitoraggio vengono raccolti e caricati su una tabella `hj`.

La raccolta dei dati è stata effettuata in due periodi distinti: il primo da settembre a dicembre 2021, e il secondo nel mese di marzo 2023. I dati sono stati immagazzinati in due database separati, identificati come `htm` per i dati del primo periodo e `htmnew` per quelli del secondo.

La tabella 1.1 mostra il numero totale di righe e lo spazio occupato su disco da ciascuna tabella nei database. Dato che la dimensione del dataset supera la capacità della memoria RAM a disposizione, risulta impossibile analizzare l'intero dataset. Pertanto, diventa necessario selezionare un sottoinsieme di dati da tali database per effettuare le analisi successive.

htm			htmnew		
	Righe	Spazio (GB)		Righe	Spazio (GB)
hj	1971830783	343	hj	1038471316	222
htjob	30799153	14	htjob	46605815	22

Tabella 1.1: Confronto delle dimensioni tra i database `htm` e `htmnew`

Le tabelle 1.2 e 1.3 offrono una panoramica sulle colonne presenti, distinguendo tra variabili categoriche e numeriche e fornendo una breve spiegazione su ciascuna colonna.

Le variabili si suddividono in base al tipo di dati che rappresentano e si suddividono in:

- *categorico nominale*, se contiene valori scelti tra un insieme finito;
- *categorico ordinale*, è simile, ma si definisce una relazione d'ordine tra i valori possibili;
- *numerico*, se è possibile quantificare le differenze tra valori.

1.3 Motivazione

Nel 1965, Gordon Moore, co-fondatore di Intel, pronosticò che il numero di transistor sarebbe raddoppiato ogni 18 mesi [21]. Tuttavia, il trend descritto da Moore arriverà a un termine quando la litografia, il processo usato per stampare i circuiti sui wafer di silicio, raggiungerà la scala atomica. Infatti, a scale atomiche i transistor incontrano fenomeni quantistici che ne disturbano il funzionamento, e le attuali tecniche di produzione diventano proibitive in termine di costi [26, 28].

I sistemi HTC e HPC sono diventati strumenti fondamentali per il progresso della ricerca scientifica. Nonostante ciò, vi sono ancora molteplici problemi importanti in vari settori che non possono essere risolti con le capacità computazionali attuali [29]. Per proseguire l'evoluzione tecnologia nell'era post-legge di Moore è quindi necessario esplorare nuove direzioni. Una di queste riguarda l'incremento del numero di core.

Tabella 1.2: Schema della tabella `hj` del database

Colonna	Tipo	Descrizione
<code>ts</code>	Numerico (secondi)	Timestamp UNIX del momento in cui il job è stato eseguito
<code>jobid + idx</code>	Categorico (Nominale)	ID univoco del job
<code>queue</code>	Categorico (Nominale)	Gruppo di appartenenza dell'utente che ha sottomesso il job
<code>hn (hostname)</code>	Categorico (Nominale)	Host sul quale il job è in esecuzione
<code>js</code>	Categorico (Nominale)	Stato del job: 1 = In coda, 2 = In esecuzione, 3 = Rimosso, 4 = Completato, 5 = Sospeso
<code>nc</code>	Numerico (core)	Numero di core CPU impiegati dal job
<code>hsj</code>	Numerico (HS06)	Potenza di un core del host
<code>hsm</code>	Numerico (HS06)	Potenza totale del host
<code>cpt (cputime)</code>	Numerico (secondi)	Tempo di esecuzione sulla CPU del job
<code>rt (runtime)</code>	Numerico (secondi)	Tempo totale di esecuzione del job
<code>owner</code>	Testo	Utente che ha sottomesso il job (username UNIX)
<code>rss</code>	Numerico (KB)	Memoria RAM utilizzata dal job
<code>swp</code>	Numerico (KB)	Memoria SWAP utilizzata dal job
<code>sn (submitnode)</code>	Categorico (Nominale)	Nodo da cui è stato sottomesso il job
<code>disk</code>	Numerico (GB)	Spazio su disco utilizzato dal job

Tabella 1.3: Schema della tabella htjob del database

Colonna	Tipo	Descrizione
<code>jobid + idx</code>	Categorico (Nominale)	ID univoco del job
<code>username</code>	Testo	Utente che ha sottomesso il job (username UNIX)
<code>queue</code>	Categorico (Nominale)	Gruppo di appartenenza dell'utente che ha sottomesso il job
<code>fromhost</code>	Categorico (Nominale)	Nodo da cui è stato sottomesso il job
<code>jobname</code>	Testo	Nome del job
<code>exechosts</code>	Categorico (Nominale)	Host sul quale il job è in esecuzione
<code>submittimeepoch</code>	Numerico (secondi)	Timestamp UNIX del momento in cui il job è stato sottomesso
<code>starttimeepoch</code>	Numerico (secondi)	Timestamp UNIX del momento in cui il job è stato eseguito
<code>endtimeepoch</code>	Numerico (secondi)	Timestamp UNIX del momento in cui il job è terminato
<code>stime</code>	Numerico (secondi)	Tempo di esecuzione sulla CPU per eseguire le chiamate al sistema per conto del job
<code>utime</code>	Numerico (secondi)	Tempo di esecuzione sulla CPU dedicato alle operazione che il job esegue direttamente
<code>runtime</code>	Numerico (secondi)	Tempo totale di esecuzione del job
<code>maxrmem</code>	Numerico (KB)	Massima memoria RAM utilizzata dal job
<code>maxrswap</code>	Numerico (KB)	Massima memoria SWAP utilizzata dal job
<code>exitstatus</code>	Categorico (Nominale)	= 0 è ok; != 0 è uscito con errore
<code>numprocessors</code>	Numerico (core)	Numero di core CPU impiegati dal job
<code>gpu</code>	Categorico (Nominale)	1 = gpu utilizzata; 0 = gpu non utilizzata
<code>completionepoch</code>	Numerico (secondi)	Timestamp UNIX del momento in cui il job è terminato
<code>jobstatus</code>	Categorico (Nominale)	Stato del job: 1 = In coda, 2 = In esecuzione, 3 = Rimosso, 4 = Completato, 5 = Sospeso

In questa tesi, un *guasto* è definito come un comportamento anomalo a livello software o hardware, che può causare stati illeciti (*errori*) nel sistema o nell'applicazione e che, nel peggio dei casi, può causarne l'interruzione (*fallimenti*).

Sfortunatamente, più core si aggiungono, maggiori sono le probabilità di riscontrare guasti hardware. In parallelo, all'aumentare della complessità hardware, si assiste a una crescente complessità del software, il che lo rende più suscettibile agli errori [6].

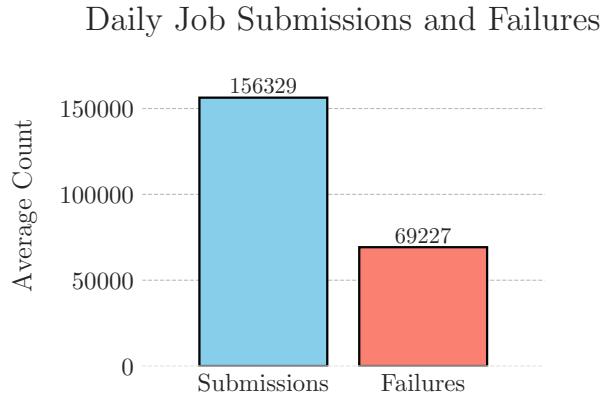


Figura 1.2: Media giornaliera di job sottomessi e falliti nel mese di Marzo 2023

Come evidenziato nella figura 1.2, si osserva che in un centro di calcolo come quello del CNAF, nel mese di Marzo 2023, sono stati sottomessi giornalmente in media 156329 job con un tasso di fallimento che supera il 40%. La frequenza con cui i job falliscono rappresenta una problematica significativa per i centri di calcolo: questa non solo causa uno spreco delle risorse del sistema, ma incide anche negativamente sull'efficienza generale e allunga i tempi d'attesa per i job in attesa di essere eseguiti.

Questa tesi si concentra sui sui fallimenti dei job piuttosto che sui fallimenti a livello di sistema, nonostante la disponibilità di dati degli host fisici del CNAF. Utilizzando tecniche di Machine Learning per identificare pattern nei dati storici, potrebbe essere possibile prevedere la riuscita o il fallimento di un job basandosi sul comportamento di job simili. Questo permetterebbe l'adozione di strategie proattive volte a prevenire i fallimenti prima che accadano, mitigando così i problemi sopracitati. In aggiunta, la capacità di informare l'utente circa il tasso di successo o fallimento di un job che sta per essere sottomesso potrebbe fornire una risorsa informativa utile.

Nel Capitolo 2, vedremo come l'analisi preliminare evidenzierà una categoria di job che falliscono, che risulta essere particolarmente interessante, soprattutto per l'importanza di identifierli e rimuoverli tempestivamente.

Capitolo 2

Analisi del database

In questo capitolo vengono presentate le analisi preliminari effettuate per il caso di studio in questione. Queste analisi hanno permesso di selezionare un sottoinsieme di job e un task di Machine Learning associato che possa apportare benefici al CNAF.

2.1 Analisi esplorativa

2.1.1 Caratterizzazione dei job

In questa tesi considereremo una serie di dati come una sequenza ordinata di punti dati, che esprime la dinamica di un certo fenomeno nel tempo. Quando questi dati sono ordinati in base al tempo, si parla di una **serie storica** (o **temporale**). Indipendentemente dal criterio utilizzato per ordinarli, i punti dati sono registrati ad intervalli di tempo equispaziati. Le serie temporali possono essere di due tipi: **univariate**, che coinvolgono una singola variabile misurata nel tempo, e **multivariate**, dove più variabili sono misurate contemporaneamente.

All'interno della tabella `hj`, che contiene i dati di monitoraggio dei job eseguiti da HTCondor, ogni riga può essere vista come un punto in una serie storica multivariata. In altre parole, ciascun job corrisponde a una serie storica multivariata distinta, dove le variabili rappresentano le diverse grandezze misurate durante il suo ciclo di vita, come illustrato nella figura 2.1. Sebbene queste serie condividano le stesse variabili, la durata di ciascun job, e di conseguenza la lunghezza delle relative serie storiche, cambia.

Come mostrato in figura 2.2, HTCondor campiona lo stato di ciascun job ogni tre minuti, ma aggiorna i valori ogni quindici minuti. Questo significa che ogni serie storica mostra un cambiamento effettivo nei valori solo ogni cinque campionamenti, risultando in una sequenza di cinque valori identici che si ripetono fino al prossimo aggiornamento.

Per quanto riguarda l'aggiornamento dei valori, HTCondor aggiorna un nuovo dato all'interno della serie storica solo quando il valore rilevato supera il precedente massimo. Di conseguenza, ciascuna serie storica può essere vista come una funzione monotona non decrescente, in cui ogni nuovo valore registrato è maggiore o uguale al precedente.

La durata dei job varia considerevolmente, come mostrato dalla distribuzione del numero di job rispetto alla loro durata in giorni, illustrata nella figura 2.3. Vi è una predominanza di job di breve durata, con un calo esponenziale del numero di job al crescere della durata.

Raggruppando i job in base alla loro durata in fasce orarie, fino a un massimo di 48 ore, e aggregando tutti quelli che superano tale durata, è possibile esaminare il numero totale dei job, la frequenza dei loro fallimenti e il cumulativo della loro durata per ciascuna delle prime quarantotto ore, così come per quelli più lunghi. Come evidenziato nella figura 2.4, i job che durano meno di un'ora sono particolarmente numerosi e presentano un elevato tasso di fallimento. Nonostante ciò, il tempo speso sulle risorse di calcolo è pressoché trascurabile se confrontato con il tempo impiegato dai job di durata superiore.

Inoltre, se ci focalizziamo sulla prima ora e suddividiamo i job in intervalli di cinque minuti, possiamo notare che molti di essi hanno una durata inferiore ai cinque minuti, come si può vedere nella figura 2.5. Questo suggerisce che questi job potrebbero essere considerati come semplici tentativi; in altre parole, sono job che, per vari motivi, non trovano le condizioni necessarie per proseguire nella loro esecuzione e quindi falliscono.

In aggiunta, secondo quanto riportato nella tabella 2.1, un significativo 11% dei job viene terminato ancor prima di raggiungere la fase di esecuzione. Questi job, non giungendo alla fase di esecuzione, non effettuano alcun calcolo, il che sottolinea la presenza di un elevato numero di tentativi che risultano irrilevanti in termini di calcolo.

Pertanto, alla luce di queste considerazioni, nasce la seguente idea: predire il fallimento di un job di lunga durata è nettamente più importante rispetto alla previsione del fallimento di un job di breve durata.

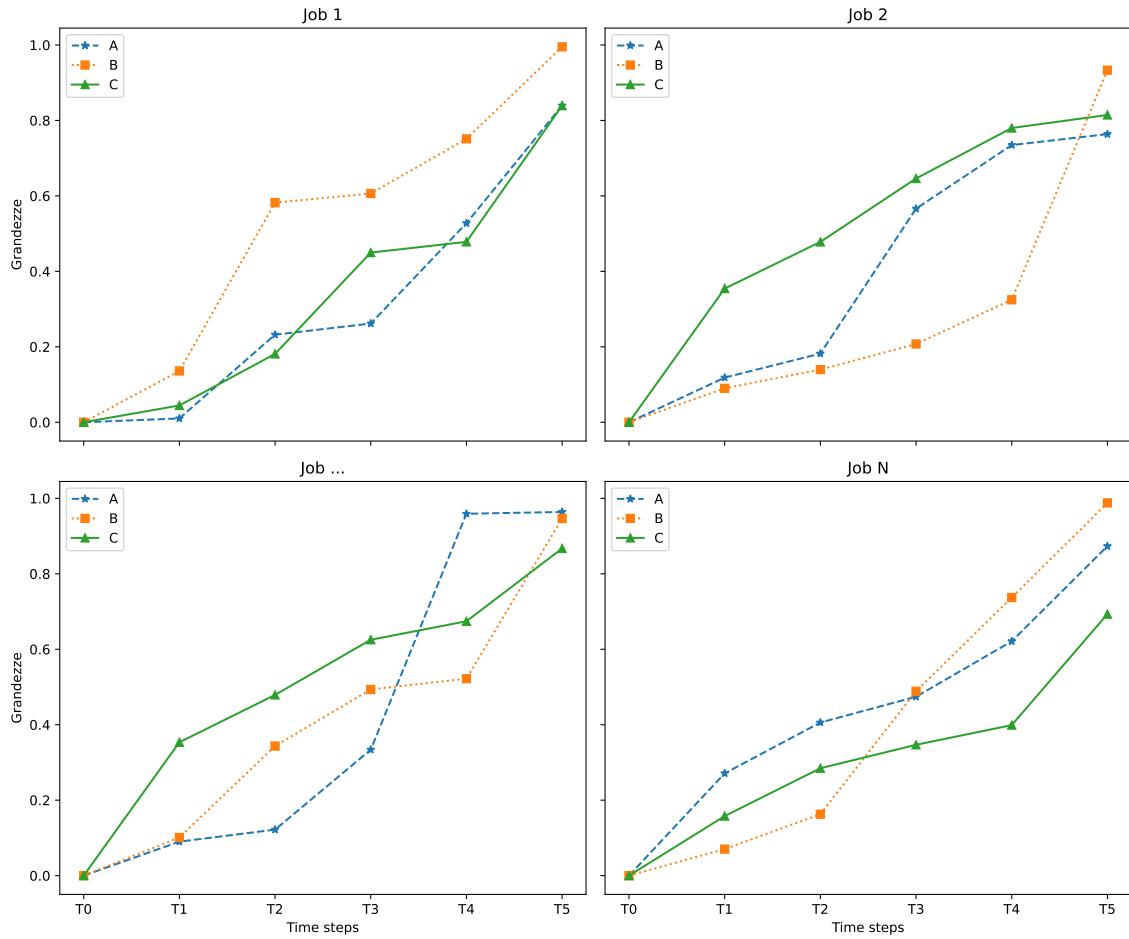


Figura 2.1: Rappresentazione di un job come serie storica multivariata

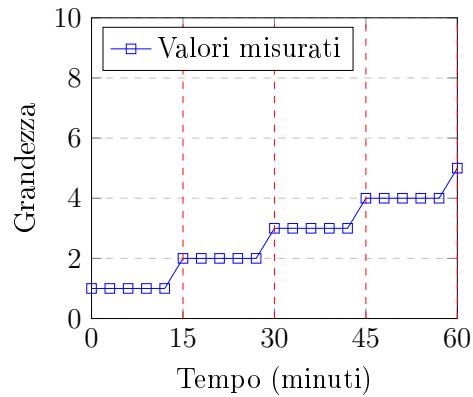


Figura 2.2: Frequenza di campionamento e aggiornamento dei job in HTCondor

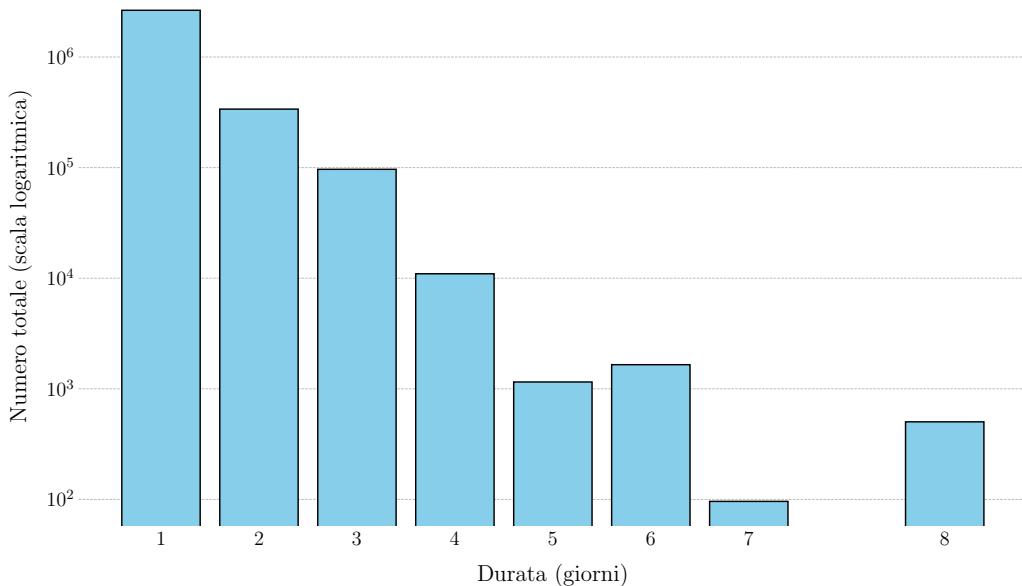


Figura 2.3: Distribuzione della durata dei job in giorni

Tabella 2.1: Percentuale di job in attesa rimossi senza aver effettuato alcun calcolo su un host fisico

Job in attesa rimossi	Job eseguiti totali	Percentuale
402,663	3,589,280	11.22

2.1.2 Caratterizzazione dei gruppi

L'analisi dei gruppi, come mostrato nella figura 2.6, conferma le osservazioni precedentemente fatte. La distribuzione della durata dei job per gruppo riflette l'andamento già notato nella figura 2.3. In particolare, si nota che con l'aumentare dei giorni, il numero di job che rimangono in esecuzione per quella durata diminuisce esponenzialmente.

La figura 2.7 evidenzia come i gruppi associati agli esperimenti LHC sottomettano un numero significativamente maggiore di job rispetto ai gruppi non-LHC. Questo elevato numero di job, tuttavia, non si traduce in un tasso di fallimento proporzionalmente alto, come osservato in precedenza nella figura 2.4. Questa differenza può essere attribuita ai meccanismi interni di controllo presenti nei gruppi LHC, i quali intervengono rimuovendo

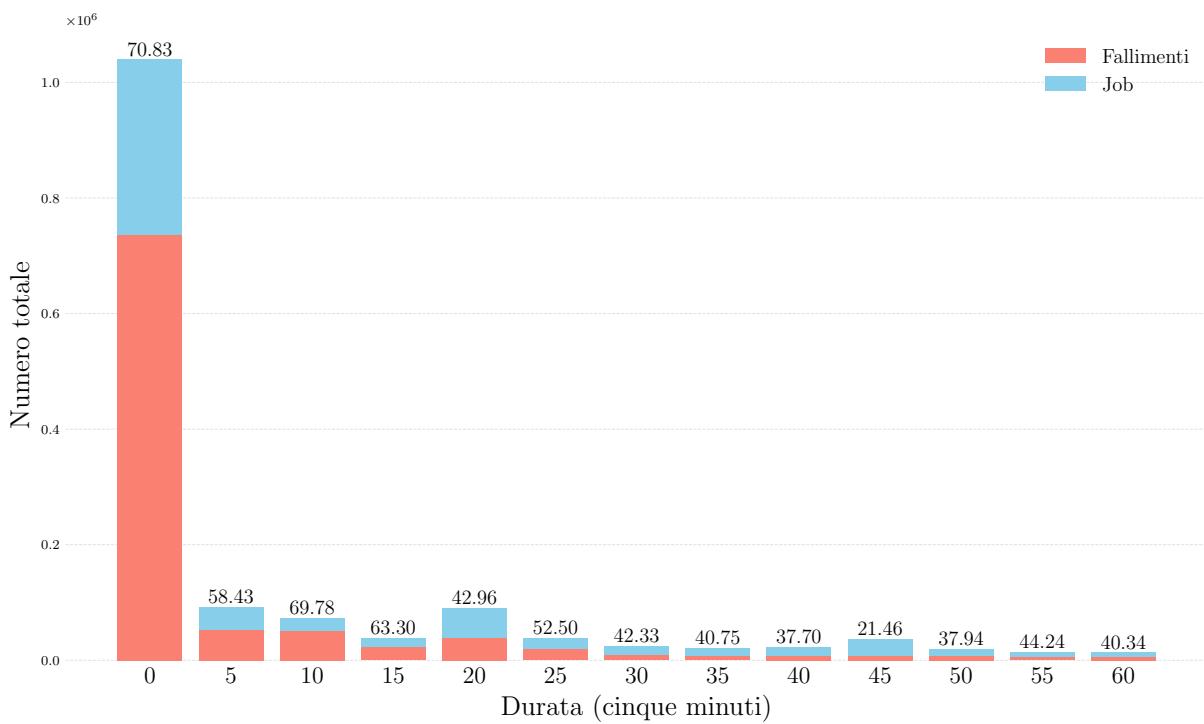
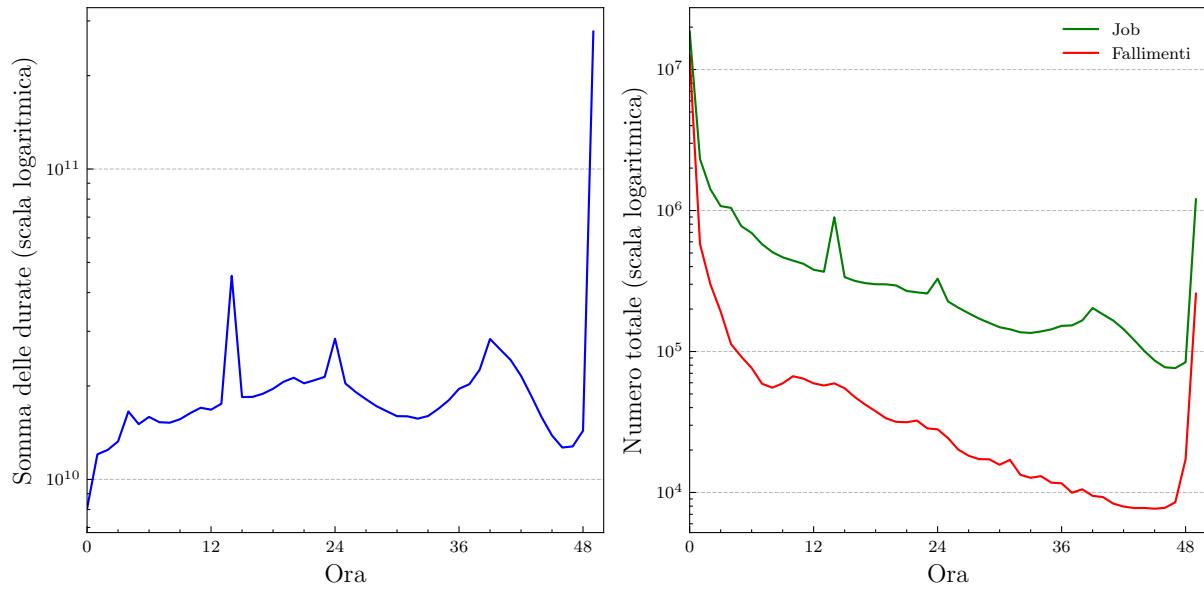


Figura 2.5: Distribuzione dei job nella prima ora, suddivisi in intervalli di cinque minuti

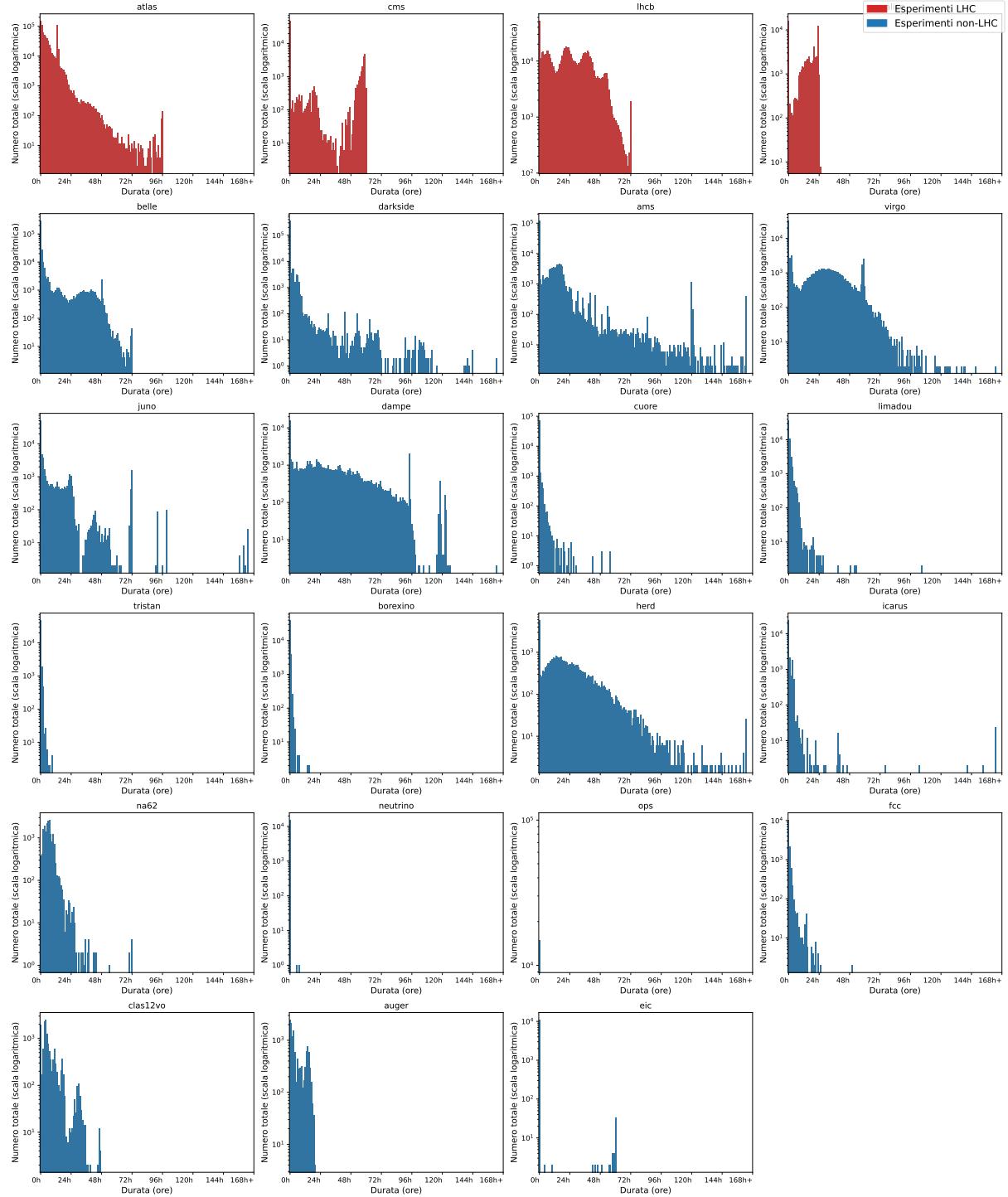


Figura 2.6: Distribuzione della durata dei job per gruppo

autonomamente i job problematici. Al contrario, alcune code non-LHC presentano un tasso di fallimento estremamente alto. Tuttavia, le ragioni specifiche di questi fallimenti non sono note, poiché HTCondor fornisce solo informazioni limitate e generiche sui motivi dei fallimenti.

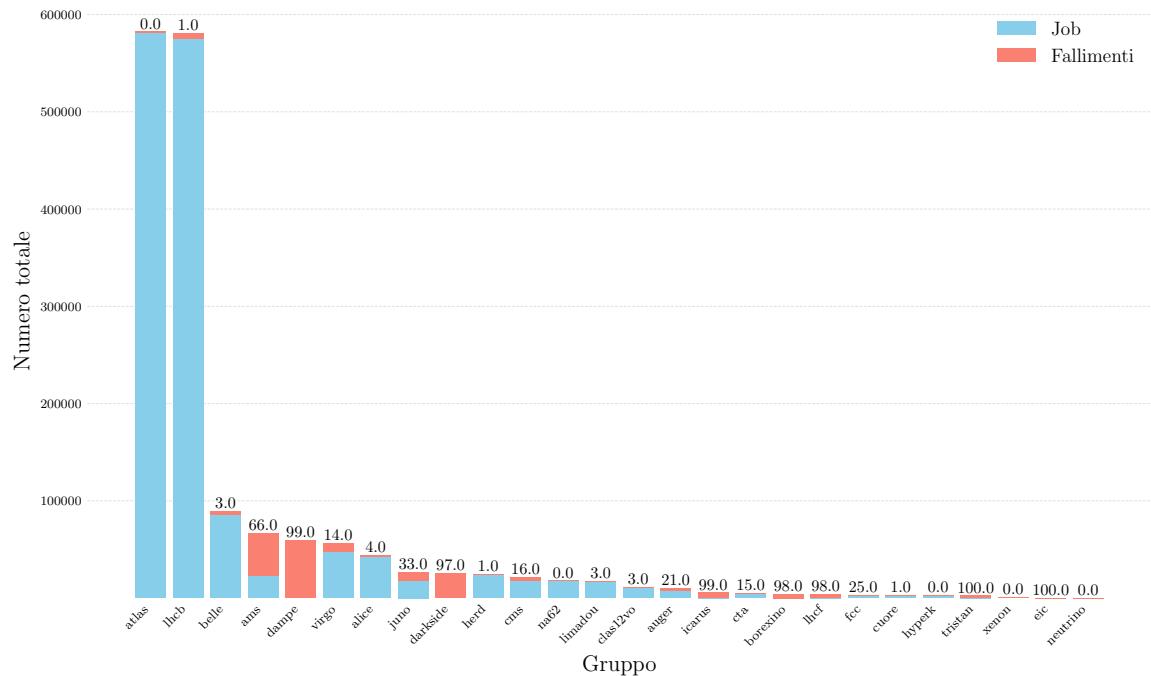


Figura 2.7: Distribuzione del numero totale dei job e dei loro fallimenti per gruppo

2.1.3 Analisi dei nomi dei job

L’analisi semantica dei nomi dei job può arricchire le feature a disposizione di un classificatore e migliorarne le prestazioni [1]. A tal fine, si sono analizzati i nomi dei job, applicando diverse tecniche di pulizia del testo per identificare i termini più frequenti.

La tabella 2.2 mostra i risultati dell’analisi semantica effettuata sui nomi dei job. Questa ha permesso di identificare i cosiddetti **job pilot**, i quali sono progettati per insediarsi in un host fisico e eseguire altri job, detti **payload**. I **job pilot** non sono direttamente coinvolti nei calcoli, ma piuttosto servono a scandagliare le risorse disponibili alla ricerca di un host fisico.

Tabella 2.2: Frequenza dei nomi dei job

Nome	Frequenza
dirac	142,868
pilotwrapper	142,868
script	72,209
eposlhc	70,736
p5600	42,766
p100	27,823
alessr	21,568
bi210	7,544
pileup	2,522
bi214	754

Tuttavia, l’analisi non ha fornito i risultati aspettati. Invece di rilevare diverse tipologie di job, i nomi tendono spesso a riflettere l’esperimento scientifico a cui sono associati, informazione meno utile a comprendere le specifiche funzioni dei job.

2.2 Job Zombie Prediction

Dall’idea delineata nella sezione 2.1.1, siamo interessati a prevedere il fallimento di quei job che garantiscono il maggior payoff. Il payoff è il beneficio ottenuto, in termini di risparmio di risorse, interrompendo tempestivamente un job che è destinato a fallire. Rappresentando il concetto di payoff associato ai job, la figura 2.8 classifica i job in base alla durata, suddividendoli in tre categorie: corti, medi e lunghi. È chiaro, come già detto, che interrompere job lunghi sia più significativo. In questa sezione introduciamo un sottotipo di job lunghi, i **job zombie**, la cui interruzione ci può garantire il massimo payoff.

I job zombie sono job che, sebbene ancora in esecuzione, si sono “bloccati” a un certo punto della loro attività, cessando di svolgere calcoli utili. Questi job entrano in uno stato di “coma”, incapaci di terminare autonomamente la loro esecuzione e continuano



Figura 2.8: Rappresentazione dei job in base alla loro durata

a occupare inutilmente risorse di calcolo finché, una volta raggiunto il limite massimo di tempo di esecuzione impostato dal sistema batch, si attiva un evento di timeout che comporta la loro rimozione dal sistema. In HTCondor, questo limite è attualmente fissato a 3 giorni per i job di tipo `grid` e a 7 giorni per i job di tipo `local`. I job `local`, sottomessi all'interno della stessa “farm” di calcolo tramite il nodo “sn-0x” da utenti che fanno parte dell'organizzazione, beneficiano di maggiore libertà. Al contrario, i job `grid` vengono sottomessi tramite nodi, identificati come “ce0x-htc”, accessibili agli utenti esterni all'organizzazione.

Tabella 2.3: Rapporto tra i job zombie e il totale dei job per ciascun gruppo

Gruppo	Job Zombie	Job totali	Percentuale	Giorni di calcolo persi
LHCb	192	262,251	0.073%	576
JUNO	151	10,137	1.49%	453
ATLAS	45	270,086	0.017%	135
LHCf	8	1,594	0.502%	24
Belle	2	42,087	0.005%	6

La tabella 2.3 mostra un totale di 1,194 giorni di calcolo persi a causa di job zombie, sottolineando l'entità del problema.

Un classificatore progettato per identificare precocemente i job zombie potrebbe liberare risorse occupate inutilmente da tali job, consentendo a nuovi job, potenzialmente produttivi, di iniziare l'esecuzione. Ciò non solo ridurrebbe lo spreco di risorse, ma aumenterebbe il throughput del centro di calcolo.

Purtroppo, poiché HTCondor registra solamente i nuovi massimi nel consumo di risorse e non tiene traccia dei decrementi, un job zombie potrebbe non utilizzare più risorse, ma apparire come se le stesse ancora utilizzando. Questi job possono quindi nascondersi dietro

ad altri job che sono in esecuzione e che stanno utilizzando le risorse, rendendo difficile la loro identificazione. Questa problematica è confermata nella figura 2.9: sebbene si osservi il consumo di RAM, SWAP e disco, la mancata registrazione dei decrementi da parte di HTCondor non permette di distinguere i job effettivamente in esecuzione da quelli zombie.

Un altro problema riguarda la rarità dei job zombie; ad esempio, considerando i dati di marzo 2023, su un totale di 950,558 job, solo 441 sono job zombie, corrispondendo a una percentuale dell'appena dello 0,046%. Questo sbilanciamento può comportare difficoltà nell'addestramento di classificatori robusti e porta anche alla necessità di un'estesa raccolta di dati nel tempo.

Per stabilire se l'applicazione di tecniche di Machine Learning è fattibile nel rilevare i cosiddetti “job zombie”, è essenziale verificare preliminarmente la presenza di cluster ben definiti all'interno del dataset. L'obiettivo sarebbe quello di distinguere con precisione questi job anomali dagli altri.

L'algoritmo t-Distributed Stochastic Neighbor Embedding (t-SNE) consente di ridurre la dimensionalità dei dati preservando la vicinanza tra i punti simili e distanziando quelli dissimili [20]. Passando da uno spazio ad alta dimensionalità a uno spazio bidimensionale o tridimensionale è possibile la visualizzazione dei dati attraverso uno scatterplot. Tuttavia, t-SNE può essere impraticabile con dataset di grandi dimensioni a causa della sua complessità computazionale dell'ordine di $\mathcal{O}(N^2)$ [22], dove N rappresenta il numero delle istanze. Data la rarità dei job zombie, è necessario selezionare un ampio periodo di monitoraggio per raccogliere un campione sufficiente di tali anomalie. Di conseguenza, si accumula un gran numero di istanze, complicando l'uso di t-SNE.

Per ovviare a questo problema, si può ricorrere a un'**autoencoder**, un tipo specifico di rete neurale progettata per comprimere i dati in una rappresentazione a dimensionalità ridotta per poi ricostruire un output il più possibile simile all'input originale. Come illustrato in figura 2.10, un autoencoder è composto da due parti: una funzione di codifica (*encoder*) che trasforma l'input in una rappresentazione compressa ($h = f(x)$) e una funzione di decodifica (*decoder*) che ricostruisce l'input a partire dalla rappresentazione compressa ($r = g(h)$). L'obiettivo è che la rete impari una funzione $g(f(x))$ che non restituisca x ma piuttosto una sua rappresentazione semplificata [14].

Se i dati sono caratterizzati da relazioni non lineari, gli autoencoder con funzioni di

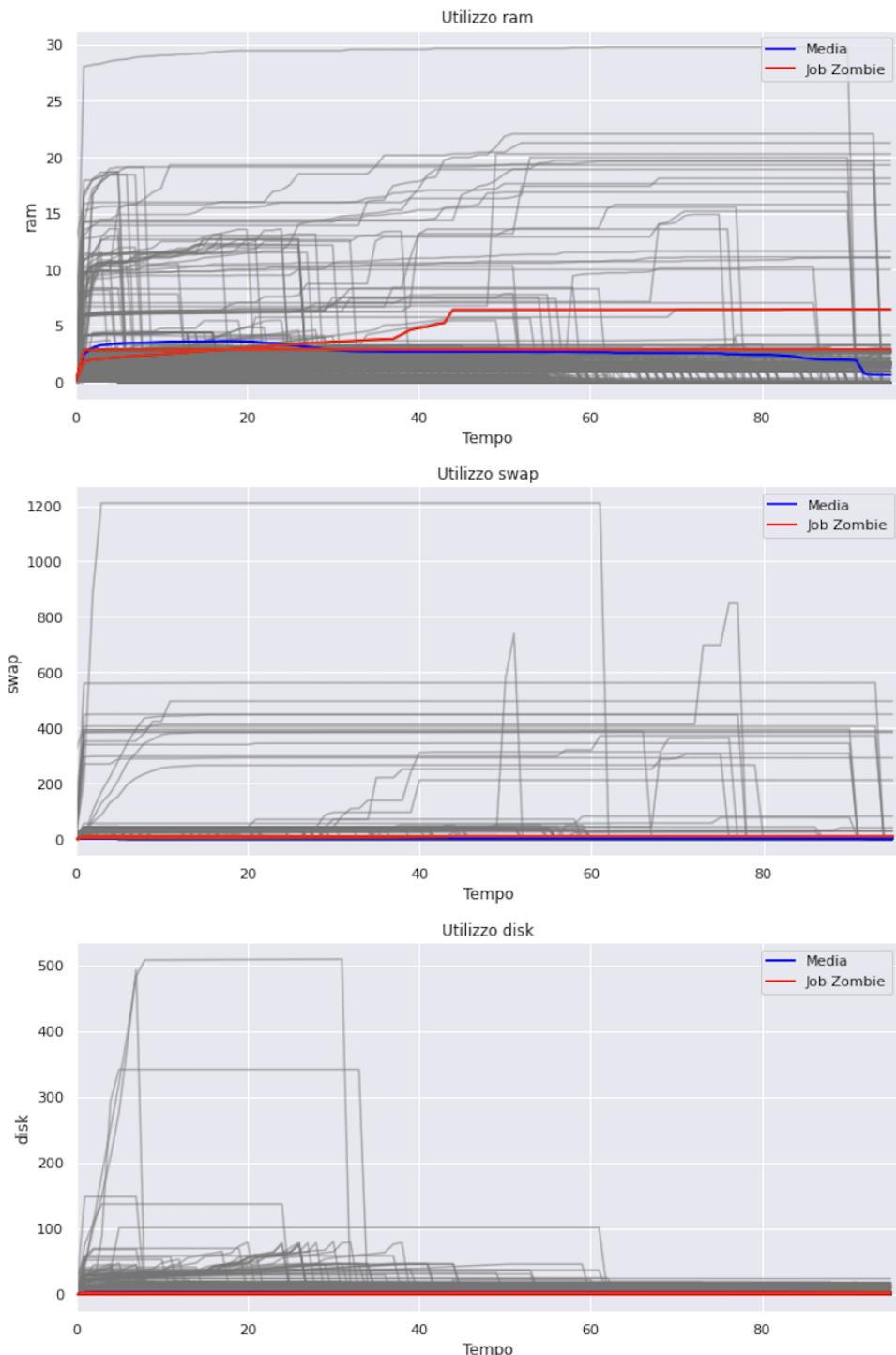


Figura 2.9: Utilizzo di RAM, SWAP e disco su intervalli di 15 minuti nelle prime 24 ore

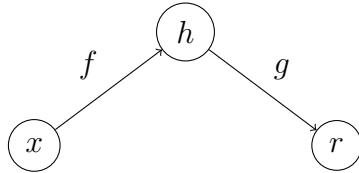


Figura 2.10: Struttura generica di un autoencoder [14]

codifica e decodifica non lineari sono in grado di modellare e preservare tali relazioni anche dopo la compressione dei dati in uno spazio a dimensione ridotta [14].

La figura 2.11 mostra come, l'applicazione di autoencoder per la compressione dei dati, seguita dall'analisi t-SNE, ha permesso l'identificazione di cluster distinti di job zombie accumulatesi nel 2021. È tuttavia importante prestare attenzione all'interpretazione dei risultati ottenuti con t-SNE, infatti le distanze tra i cluster o le loro dimensioni apparenti non sono significative [30]. In aggiunta, la composizione di questi cluster risulta indipendente dal gruppo che ha sottomesso i job. La prevalenza di job provenienti dagli esperimenti LHCb e ATLAS non riflette altro che l'abbondante sottomissione di job da parte degli esperimenti LHC.

Ulteriormente, esaminando i job relativi agli esperimenti LHC, in particolare quelli relativi ad ATLAS, durante la seconda metà di settembre 2021, si è notato che alcuni job zombie si raggruppano in cluster distinti, come illustrato nella figura 2.12. Questo suggerisce che questi job potrebbero essere identificati con maggiore facilità. D'altra parte, vi sono job che si mescolano tra quelli normali, il che potrebbe rendere la loro individuazione più complessa.

In conclusione, l'obiettivo di questa tesi è verificare che i job zombie possano essere identificati attraverso l'applicazione di modelli di Machine Learning.

Nel capitolo 3, esploreremo due diversi modi di modellare il problema con il Machine Learning, applicando tecniche specifiche per ciascuno. Successivamente, nel capitolo 4, valuteremo e confronteremo i risultati ottenuti attraverso i due approcci e le relative tecniche utilizzate.

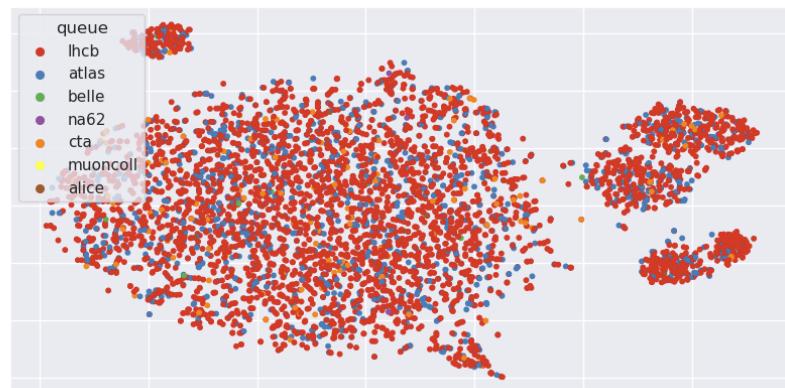


Figura 2.11: Visualizzazione job zombie del 2021 tramite t-SNE

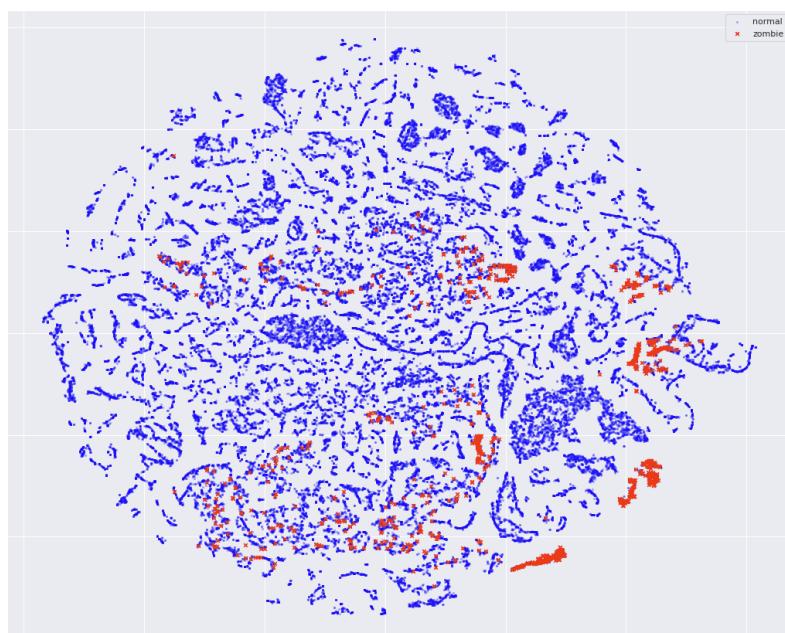


Figura 2.12: Visualizzazione job ATLAS di settembre 2021 tramite t-SNE

Capitolo 3

Tecniche di Machine Learning e Deep Learning

In questa tesi considereremo il Machine Learning come il processo che consente la creazione di modelli in grado di apprendere autonomamente dai dati. Un modello è costituito da un insieme di parametri e una struttura che elabora i dati di input per produrre un output. I parametri vengono appresi durante la fase di addestramento, in cui il modello esamina vari esempi e regola i propri parametri di conseguenza. Gli iperparametri, d'altra parte, sono valori definiti dall'utente prima dell'inizio dell'addestramento che influenzano la struttura del modello e il suo comportamento durante l'addestramento.

Prima di esplorare i modelli specifici, ci concentreremo sul processo di creazione dei modelli di Machine Learning, e vedremo come l'automatizzazione di questo processo, attraverso l'impiego di una **pipeline**, possa ottimizzarne l'efficienza e l'efficacia.

Il processo di creazione di un modello di Machine Learning si compone di vari passaggi: l'estrazione dei dati, la loro preparazione e l'addestramento del modello. Una pipeline collega questi passaggi in sequenza, incapsulandoli in un'entità che, dall'esterno, può essere utilizzata come se fosse il modello stesso. Questa pipeline può essere rappresentata come un Grafo Aciclico Diretto (DAG), dove i dati fluiscono in una sola direzione, evitando cicli, e dove ogni nodo in questo grafo rappresenta una fase distinta del processo (vedi figura 3.1).

L'uso di una pipeline nel Machine Learning ci permette di ottenere i seguenti vantaggi:

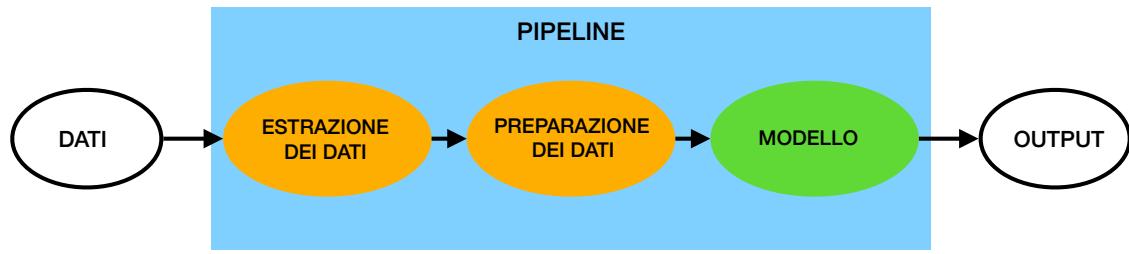


Figura 3.1: Rappresentazione di una pipeline di Machine Learning come un grafo orientato senza cicli

Efficacia Estensione della ricerca degli iperparametri ad altri componenti: Mentre l'individuazione dei parametri migliori per un modello avviene in modo automatico durante l'addestramento, la ricerca degli iperparametri migliori richiede sperimentazioni multiple, testando diversi valori e valutando i risultati del modello secondo metriche prestabilite. Dato che esternamente una pipeline funziona esattamente come un modello, è possibile estendere la ricerca degli iperparametri migliori a componenti non direttamente correlati al modello stesso, come quelle legate all'estrazione e alla preparazione dei dati. Poiché anche la ricerca degli iperparametri è automatizzabile, è possibile testare automaticamente diverse tecniche di preparazione dei dati, semplicemente integrando il componente alla pipeline.

Efficienza Sperimentazione rapida: L'organizzazione di tutti i passaggi in una pipeline può accelerare notevolmente la sperimentazione. Questo è particolarmente utile quando si prevede di testare vari iperparametri o di utilizzare differenti sottoinsiemi di dati. Infatti, incapsulando le operazioni delle diverse componenti in un unico elemento che le esegue sequenzialmente, si evita di ripetere le stesse operazioni più volte, risultando in un risparmio di tempo significativo.

Nelle sezioni seguenti, descriveremo come sono stati affrontati i vari passaggi nella creazione di un modello di Machine Learning per l'identificazione dei job zombie e come si è cercato di integrare ciascun componente alla pipeline.

3.1 Estrazione dei dati

L'estrazione di un dataset viene fatta tramite una query SQL che interroga le tabelle `hj` e `htjob`, selezionando i dati rilevanti. Ricordando che:

- La tabella `hj` contiene lo stato dei job, rappresentato da serie storiche di misurazioni (come `runtime`, `ram`, `swap`, `disk`, ecc.), durante la loro esecuzione.
- La tabella `htjob` fornisce informazioni sull'esito dei job, indicando se sono falliti o meno.

La query esegue le seguenti operazioni:

- Seleziona i job che hanno iniziato e finito la loro esecuzione nel periodo temporale specificato.
- Esegue un JOIN delle tabelle utilizzando l'identificativo univoco di ciascun job (`job.bid.idx_submitnode`) e il timestamp. Questo timestamp sfrutta l'indice presente nella tabella `hj` per gestire in maniera efficiente le grandi dimensioni di questa tabella¹. Poiché la tabella `hj` contiene più record per ogni job, la query li raggruppa per job. In seguito, mediante l'uso dell'operatore `ARRAY_AGG`, le serie storiche vengono trasformate in liste di valori.
- Filtra i job con un tempo di esecuzione superiore a un'ora (`runtime > 3600`), in quanto i job più brevi sono considerati irrilevanti per lo scopo dello studio.

Il risultato di questa query è un dataset come mostrato nella figura 3.2, dove ogni riga rappresenta un job e le colonne includono:

- `job`: identificativo univoco per ogni job.
- `queue`: gruppo di appartenenza dell'utente che ha sottomesso il job.

¹ La logica dietro questo consiste nel selezionare un job da `htjob` e successivamente cercarlo in `hj` limitando la ricerca ai record che rientrano nel periodo in cui `hj.ts` è compreso tra `htjob.starttimeepoch` e `htjob.eventtimeepoch`. Questo permette di restringere notevolmente la ricerca nella tabella `hj` per ogni job selezionato da `htjob` ed evitare di scansionare l'intera tabella.

- **fail**: una variabile booleana che indica se il job è fallito.
 - **mint** e **maxt**: il tempo minimo e massimo di esecuzione del job.
 - **t**, **ram**, **swap**, **disk**: liste di valori che rappresentano le serie storiche di misurazioni.

Figura 3.2: Le prime cinque righe del dataset

Tuttavia, il dataset estratto non è ancora pronto per l'addestramento del modello e necessita di ulteriori trasformazioni da parte del componente successivo. Questo passaggio non è stata integrato nella pipeline, in quanto, nell'ambiente operativo reale, si prevede che questi lavori direttamente con i dati forniti da HTCondor, eliminando così la necessità di estrarre dati da un database SQL.

3.2 Preparazione dei dati

La preparazione dei dati è essenziale nel Machine Learning. Prima di tutto, è necessario convertire i dati in numeri, dato che gli algoritmi di Machine Learning lavorano esclusivamente con dati in tale forma. Inoltre, la qualità e la quantità dei dati sono determinanti per l'efficacia del modello. Se i dati disponibili sono insufficienti o di bassa qualità, i risultati saranno scadenti a prescindere dalla complessità del modello utilizzato. Tipicamente, maggiore è la complessità di un modello, tanto più esso richiederà una grande quantità di dati di alta qualità.

Per realizzare ciò, è stata creata una classe denominata **Preprocessor**, che riceve in input un dataset e restituisce in output un dataset modificato, eseguendo una serie di operazioni intermedie. Queste operazioni possono essere ricondotte a tre categorie: l'aggiunta, la rimozione e la trasformazione di colonne. Le operazioni effettuate sono configurabili attraverso parametri definiti nel costruttore della classe al momento della sua istanziazione. Quando questo componente viene inserito all'interno di una pipeline, tali parametri fungono da iperparametri, permettendo di esplorare diverse configurazioni per identificare quale produca i risultati migliori.

In aggiunta, questa classe è stata implementata seguendo il design pattern Template Method, nel quale i passaggi di un algoritmo vengono divisi in metodi separati, e successivamente invocati da un metodo denominato template (vedi figura 3.3). La superclasse definisce lo scheletro dell'algoritmo, consentendo alle sottoclassi di personalizzare alcuni passaggi sovrascrivendo alcuni metodi. In questo modo, è possibile codificare la parte invariante dell'algoritmo una sola volta nella superclasse, lasciando alle sottoclassi il compito di implementare i comportamenti che possono variare. In pratica, l'uso di questo pattern in questa classe ci consente di aggiungere, modificare e rimuovere passaggi dall'algoritmo in modo semplice ed immediato, senza la necessità di dover intervenire sul codice della classe principale.

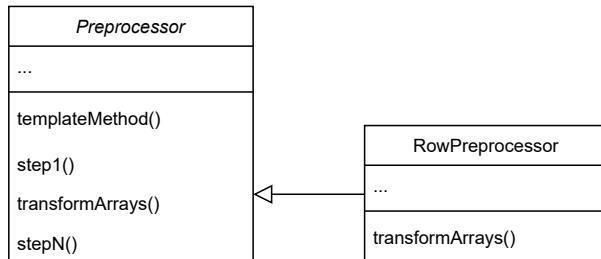


Figura 3.3: Diagramma UML della classe **Preprocessor**, illustrante l'implementazione del design pattern Template Method

Dopo aver delineato la struttura generale della classe **Preprocessor**, descriveremo ora le trasformazioni effettuate dal metodo `preprocess()`, che funge da metodo template, nella preparazione dei dati.

3.2.1 Preparazione delle serie storiche

Nella sezione 2.1.1, abbiamo identificato alcuni problemi nelle misurazioni registrate da HTCondor relative allo stato dei job. Un problema è la presenza di valori ripetuti all'interno delle serie storiche. Queste serie sono attualmente rappresentate come liste di valori, che non corrispondono al formato numerico richiesto dai modelli di Machine Learning. Pertanto, è necessario non solo rimuovere le ripetizioni, ma anche convertire queste serie storiche in un formato di dati strutturato.

Riduzione della frequenza di campionamento. Applicando un'operazione di convoluzione² con un passo (*stride*) di 5 e un filtro di 5 elementi con valore $\frac{1}{5}$, possiamo effettuare una decimazione della sequenza originale. Ciò comporta di ottenere una nuova sequenza, la cui lunghezza è pari a un quinto della lunghezza della serie storica originale. In pratica, il filtro calcola la media di ogni gruppo di cinque valori consecutivi. Se questi cinque valori sono identici, il risultato sarà il valore stesso, che sostituisce la sequenza dei cinque valori, eliminando le ripetizioni nella nuova sequenza.

Trasformazione delle multiple serie storiche multivariate. Sebbene durante il processo di estrazione dei dati le serie storiche multivariate siano state rappresentate come liste, il dataset rappresenta ancora le tre dimensioni delle multiple serie multivariate: job (righe), variabili (colonne) e time step (liste), come illustrato nella figura 3.4. Una possibile soluzione potrebbe essere quella di trasformare le liste in una singola statistica, come la media o il massimo, ma ciò cancellerebbe qualsiasi indicazione sull'evoluzione temporale di ciascun job. Il nostro obiettivo è quello di introdurre nuove feature che

²Operazione matematica che consiste nell'applicare un filtro di dimensione finita lungo la sequenza di valori. Il filtro, di solito di piccole dimensioni, viene fatto scorrere su tutta la sequenza, e in ogni posizione si calcola una somma pesata tra i valori della sequenza e quelli del filtro. Questo processo trasforma la sequenza originale in una nuova attraverso la formula:

$$(S * K)(i) = \sum_{m=-d}^d S(m) \cdot K(i - m)$$

dove S è la sequenza originale, K è il filtro e d è l'intero inferiore della metà della lunghezza del filtro.

possano riflettere la tridimensionalità originale dei dati. Tuttavia, prima di procedere, è necessario assicurarsi che tutte le serie storiche siano uniformate alla stessa lunghezza.

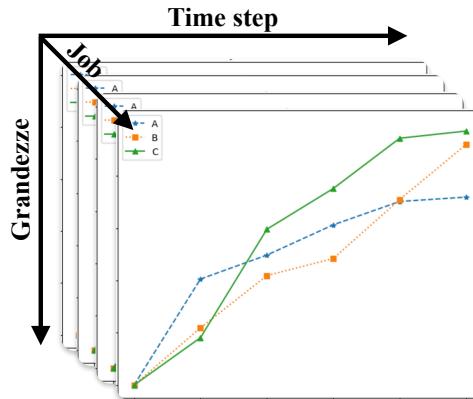


Figura 3.4: Rappresentazione tridimensionale delle multiple serie storiche multivariante

Per gestire serie di dati con lunghezze diverse, possiamo utilizzare due strategie: lo zero-padding e il troncamento. Lo zero-padding si applica alle sequenze più corte, aggiungendo zeri fino a raggiungere una lunghezza prefissata. Invece, il troncamento si usa per ridurre le sequenze più lunghe, tagliandole fino a che non raggiungono la stessa lunghezza prestabilita. La classe `Preprocessor` implementa entrambe le tecniche: stabilisce una lunghezza fissa per tutte le sequenze, troncando quelle che eccedono questa lunghezza e applicando lo zero-padding a quelle che non la raggiungono.

Una volta ottenute serie storiche di uguale lunghezza, vengono applicate le seguenti trasformazioni, in base al modello utilizzato:

- **Trasformazione in colonne:** Ogni elemento di ogni lista diventa una colonna separata nel dataset. Ad esempio, con un sottocampionamento a 15 minuti per un giorno, avremo 96 time step, corrispondenti a 96 colonne per ciascuna serie temporale.
- **Trasformazione in righe:** Gli elementi nelle stesse posizioni nelle liste formano righe distinte. Quindi, con un sottocampionamento a 15 minuti per un giorno, otteniamo 96 righe per ogni job.

3.2.2 Creazione delle feature

Dopo aver convertito le serie storiche in formato numerico, rimangono alcune colonne, come `job` e `queue`, che sono di tipo categorico nominale e che necessitano anch'esse di essere convertite in formato numerico. Inoltre, è importante assicurarsi che le colonne numeriche siano sulla stessa scala, poiché le differenze di scala possono portare a prestazioni subottimali nei modelli. Quindi, un passo importante nella preparazione dei dati è il ridimensionamento di queste colonne.

One-hot encoding. Una possibile soluzione per gestire le colonne di tipo categorico potrebbe essere quella di assegnare un valore intero a ciascun valore categorico. Tuttavia, questa strategia può indurre il modello a interpretare i valori numerici vicini come simili e quelli distanti come dissimili, il che non è appropriato per le colonne di tipo categorico nominale.

Per risolvere questo problema, si può utilizzare la tecnica dell'one-hot encoding, che crea una colonna binaria per ogni valore categorico: la colonna sarà impostata a 1 per la categoria corrispondente e a 0 per tutte le altre. Sfortunatamente, il one-hot encoding può generare un eccessivo numero di colonne in presenza di colonne con alta cardinalità, come `job` ($\mathcal{O}(\text{numero righe})$) o `queue` ($\mathcal{O}(50)$). In questi casi, si rischia di avere troppe feature irrilevanti, compromettendo l'efficacia del modello di Machine Learning.

Per garantire che il modello impari efficacemente dai dati, è fondamentale selezionare feature rilevanti ed eliminare quelle irrilevanti. È altresì importante che il modello interpreti i dati in modo simile a come li percepiamo noi, evidenziando le caratteristiche salienti dei dati e la struttura del problema.

Per far ciò, sono state create due nuove colonne, `job type` e `job work type` (vedi figura 3.5), seguite dall'applicazione dell'one-hot encoding. La prima colonna raggruppa i gruppi di utenti, in LHC e non-LHC, che, come osservato nell'analisi preliminare, hanno meccanismi interni diversi e potrebbero comportarsi in maniera differente. Allo stesso modo, viene estratto il `submit_node` dalla colonna `job`, dove ogni ID è composto da `jobid.idx_submitnode`, e classificato i job in base al fatto che il `submit_node` sia “sn0x” o “ce0x”, distinguendo così i job sottomessi dagli utenti interni del CNAF da quelli degli utenti esterni.



Figura 3.5: Visualizzazione delle nuove colonne `job type` e `job work type`

Ridimensionamento delle feature. La normalizzazione è il processo che ridimensiona i dati in un range tra 0 e 1. La formula è $X' = \frac{(X - X_{min})}{X_{max} - X_{min}}$, dove X_{min} e X_{max} sono rispettivamente i valori minimi e massimi della feature. La standardizzazione trasforma i dati in modo che abbiano media 0 e varianza di 1. La formula è $X' = \frac{X - \mu}{\sigma}$.

Ridimensionare i dati è

Ridimensionare i dati è importante perché aiuta i modelli a convergere meglio ai parametri migliori

3.2.3 Etichettatura dei dati

In base alla presenza di etichette nei dati, possiamo distinguere tra apprendimento supervisionato e non supervisionato. Nell'apprendimento supervisionato, i modelli vengono addestrati con un dataset, dove ogni esempio è associato a un'etichetta, rappresentante un valore categorico o numerico. Durante l'addestramento, il modello tenta di prevedere le etichette per esempi che non ha mai visto prima. Le predizioni sono poi confrontate con le etichette per calcolare l'errore, che indica quanto le predizioni del modello si discostano dai valori reali. L'obiettivo è migliorare la precisione del modello minimizzando l'errore, tipicamente attraverso la discesa del gradiente.

Attualmente, il dataset non include etichette che identifichino quali job siano zombie, il che ci impedisce di impostare un apprendimento supervisionato. Per generare le etichette, calcoliamo il tempo di esecuzione di ogni job come `int((maxt - mint) / 86400)` per ottenere il numero di giorni di esecuzione. Poi, con la colonna `job_type`, etichettiamo come job zombie quelli che soddisfano la condizione:

```
se giorni > (3 se job_type == 'grid', altrimenti 7) e fail == 1
```

3.2.4 Tecniche di bilanciamento dei dati

Nella sezione 2.1.1, abbiamo osservato che i job zombie sono estremamente più rari dei job normali, con un rapporto di 1 a 10,000. A causa di questo forte sbilanciamento, il classificatore potrebbe semplicemente apprendere a identificare tutti gli esempi come job normali (etichettati come 0), ottenendo così un errore apparentemente molto basso, ma in realtà ignorando completamente i job zombie, che sono esattamente quelli che desideriamo identificare.

Per tentare di ridurre il bias del modello verso la classe maggioritaria sono state adottate tre tecniche in combinazione: sottocampionamento, sovraccampionamento e Cost-sensitive learning. Queste tecniche mirano a bilanciare la distribuzione delle classi nel dataset e a migliorare la capacità del modello di distinguere tra job normali e job zombie.

Sottocampionamento casuale. Il sottocampionamento casuale è una strategia molto semplice in cui vengono cancellati casualmente degli esempi di job normali dal dataset. Tuttavia, ciò può comportare la perdita di informazioni preziose per il modello, causando una perdita della sua precisione [17].

Sovraccampionamento. Un’alternativa è il sovraccampionamento dei job zombie. Il metodo più semplice consiste nell’aggiungere duplicati di esempi già presenti nel dataset; tuttavia, ciò non fornisce nuove informazioni utili al modello. Un approccio più efficace è la generazione di nuovi esempi artificiali, che sono varianti dei job zombie [5].

Pertanto, si è scelto di usare una variante dell’autoencoder, nota come **variational autoencoder** [18], per generare varianti di job zombie a partire dai job zombie dell’intero 2021. Gli autoencoder tradizionali apprendono funzioni per mappare esempi in punti nello spazio latente e viceversa, per mappare le loro rappresentazioni compresse dallo spazio latente allo spazio originale. Tuttavia, se si prende una variante di un esempio nello spazio latente, il decoder $g(h)$ genererà un output privo di senso, in quanto non è in grado di gestire regioni dello spazio mai esplorate. Il variational autoencoder risolve questo problema facendo sì che l’encoder anziché restituire una rappresentazione compressa h , fornisca una media μ e una deviazione standard σ . Intuitivamente, la rappresentazione compressa viene campionata da una distribuzione gaussiana caratterizzata dalla media μ

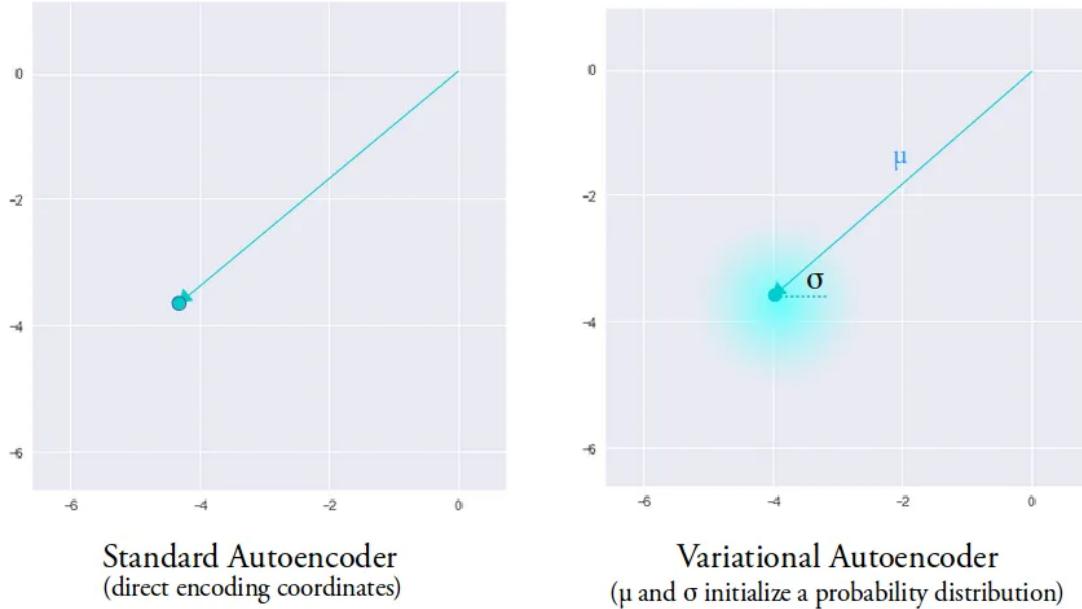


Figura 3.6: Rappresentazioni compresse in un autoencoder tradizionale e un variational autoencoder [25]

e dalla deviazione standard σ . In questo modo, il decoder impara a mappare non solo un singolo punto nello spazio latente, ma anche tutti i punti nelle sue vicinanze (vedi figura 3.6) [25].

Tuttavia, il sovraccampionamento aumenta il rischio di overfitting, cioè la possibilità che il modello si adatti eccessivamente ai dati di addestramento, perdendo così la capacità di generalizzare su dati nuovi [11].

Cost-sensitive learning. Possiamo distinguere gli errori commessi dal modello in falsi positivi e falsi negativi in base a due criteri: un errore è falso positivo quando il modello identifica erroneamente un job come zombie nonostante non lo sia; è invece un falso negativo quando il modello etichetta come normale un job che in realtà è zombie.

Assegnando un valore diverso, che definiamo “costo”, ai falsi positivi e ai falsi negativi, e minimizzando il costo totale³ derivante dalle predizioni errate, il modello imparerà durante l’addestramento ad attribuire diversa importanza ai vari tipi di errori.

³Costo totale = $C_{FN} \cdot FN + C_{FP} \cdot FP$

3.3 Selezioni dei modelli

Nel Machine Learning, il problema di identificare i job zombie può essere risolto modellando il problema in diversi modi e creando modelli specifici per risolverlo. In questa tesi, si è scelto di approcciare il problema modellandolo in due modi: la classificazione e la novelty detection.

Nel primo caso, al modello viene richiesto di specificare, come output, a quale dei k valori categorici un esempio appartiene. Per risolvere questo problema, il modello deve produrre una funzione del tipo $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$ basandosi sulle feature degli esempi in input [14].

Nel caso della novelty detection, invece, si chiede al modello di modellare i dati e di riconoscere le “novità”, ovvero ciò che non ha mai visto dai dati. Consideriamo una novità come un punto dei dati che non appare consistente con ciò che è stato osservato nei dati di addestramento. A differenza della classificazione, i modelli non modellano esplicitamente un'anomalia. Essi imparano solo a riconoscere ciò che è normale. Per funzionare efficacemente, l'addestramento richiede un dataset “pulito”, cioè privo di job zombie [13, 23].

L'ipotesi è che, modellando sufficienti dati, possiamo ottenere un modello che ha già visto abbastanza per riconoscere i job zombie come novità rispetto a ciò che ha appreso come normale (vedi figura 3.7).

Procederemo con la presentazione dei modelli impiegati e illustreremo le ragioni della loro scelta.

3.3.1 XGBoost (Classificatore)

XGBoost (eXtreme Gradient Boosting) è un modello basato su un insieme, o “ensemble”, di alberi decisionali (vedi figura 3.8). Ogni albero in questo ensemble è sua volta un modello, con nodi intermedi dell'albero che pongono domande binarie sulle feature dei dati, dividendoli in sottoinsiemi. Questo processo si ripete ricorsivamente in ciascun nodo fino a raggiungere i nodi foglia, che rappresentano le classificazioni finali per le istanze.

XGBoost utilizza il metodo del Gradient Boosting per costruire l'ensemble di alberi decisionali, mirando a combinare modelli “deboli” per formare un modello “forte”. In que-

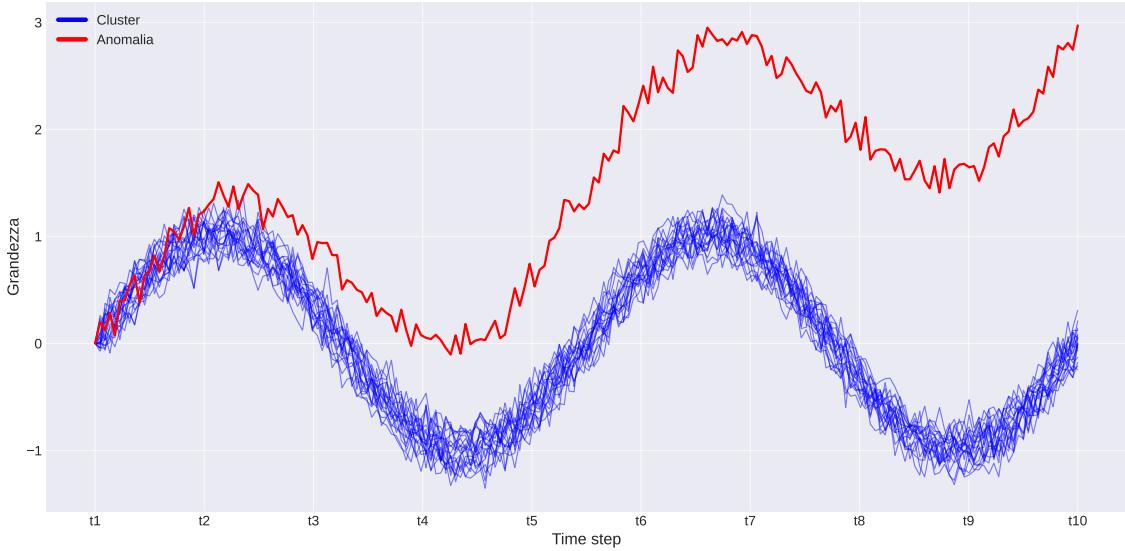


Figura 3.7: Rappresentazione ideale di una serie storica anomala in contrasto con un cluster di serie storiche normali

sto metodo, nuovi alberi decisionali vengono aggiunti iterativamente, ciascuno addestrato per correggere gli errori residui del precedente. Idealmente, aggiungendo sufficienti alberi, i residui si distribuiranno casualmente attorno allo zero, rendendo impossibile ulteriori distinzioni [4, 12].

In pratica, il processo inizia con un modello che fa previsioni costanti (per esempio, predice sempre 0) e, ad ogni iterazione k , viene aggiunto un nuovo modello $\hat{y}_i^{(k)} = f_k(x_i)$ al precedente fino al raggiungimento del numero massimo di alberi definito da un iperparametro. La struttura di questo processo può essere descritto dalle seguenti equazioni:

$$\begin{aligned}\hat{y}^{(0)} &= 0 \\ \hat{y}^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\ &\vdots \\ \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k = \hat{y}_i^{(t-1)} + f_t(x_i)\end{aligned}$$

L'ensemble può essere rappresentato come la somma di t funzioni, $(\sum_{k=1}^t f_k(x_i))$, dove ogni funzione corrisponde a un albero. Durante l'addestramento, XGBoost ottimizza

queste t funzioni con la seguente funzione obiettivo⁴:

$$\text{obj}^{(t)} = \sum_{i=1}^n \ell(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^t \Omega(f_k)$$

che è la somma delle funzioni di perdita⁵ e dei termini di regolarizzazione per i t alberi. Il termine di regolarizzazione, introdotto da XGBoost come ottimizzazione al Gradient Boosting insieme ad altre ottimizzazioni, contribuisce a ridurre il rischio di overfitting [8].

Una volta che il modello è addestrato, le predizioni vengono effettuate calcolando le predizione di ogni albero e sommando insieme i risultati di tutti gli alberi per ottenere la previsione finale.

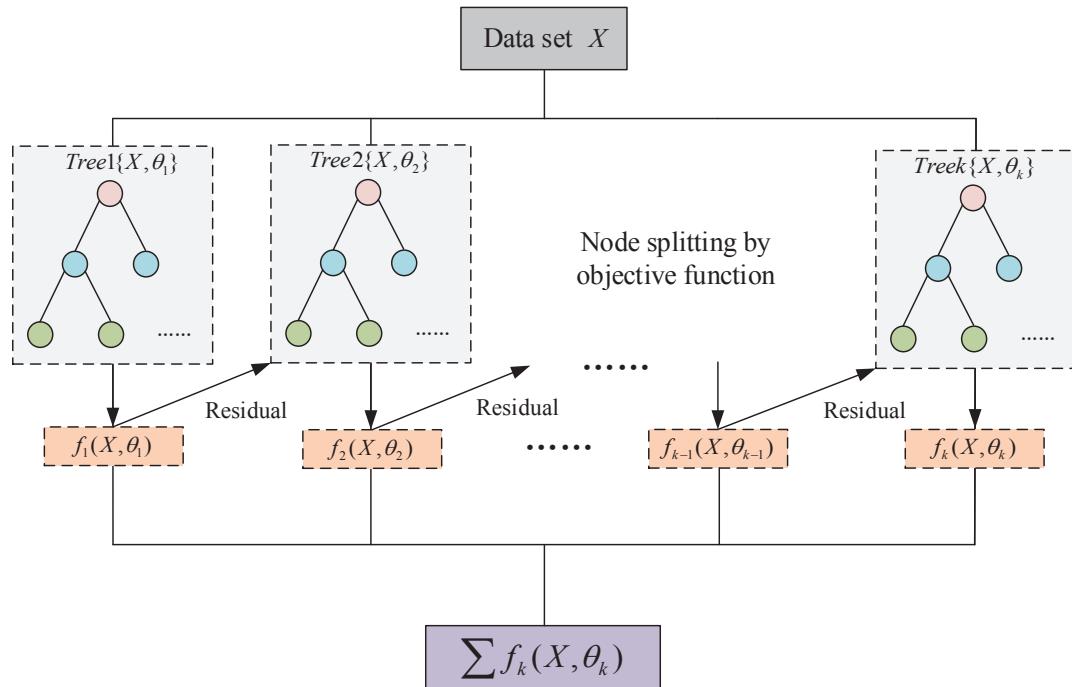


Figura 3.8: Struttura semplificata di XGBoost [15]

Nonostante il teorema “no free lunch” affermi che non esista un modello che a priori sia superiore ad altri e che la scelta del modello possa essere determinata solo attraverso test

⁴La funzione che andremo a minimizzare durante l’addestramento.

⁵La funzione di perdita misura la discrepanza tra i valori predetti dal modello e i valori reali dei dati.

comparativi, XGBoost si tratta di un'ottima prima scelta nella trattazione di problemi reali, grazie alla sua efficacia nel trattare dati strutturati [27, 8].

3.3.2 Reti neurali (Classificatore)

Come discusso nel paragrafo 3.2.2, la creazione di un modello che apprenda efficacemente dai dati richiede un'attenta selezione di feature rilevanti, un processo che non è semplice. Un'alternativa è l'uso di reti neurali profonde per la classificazione. Le reti neurali convoluzionali (CNN), ad esempio, offrono il vantaggio di estrarre automaticamente le feature più rilevanti durante l'addestramento. Altre tipi di reti, come le reti neurali ricorrenti (RNN) e i Transformer, sono in grado di utilizzare l'intera storia di una serie temporale e catturare le dipendenze temporali per fare delle previsioni.

Una rete neurale è composta da unità chiamate neuroni, organizzati in strati: uno strato di input, che riceve i dati iniziali; uno o più strati nascosti che elaborano i dati; e infine uno strato di output, che produce il risultato finale. I neuroni di ogni strato sono connessi a quelli degli strati adiacenti tramite connessioni pesate (vedi figura 3.9).

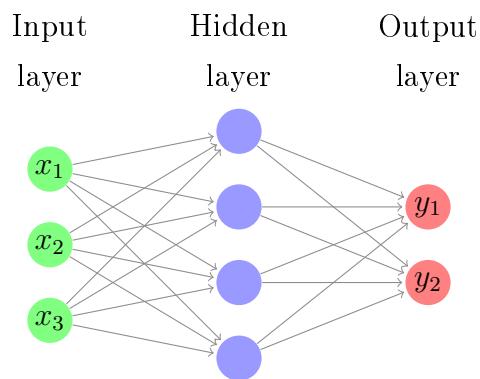


Figura 3.9: Struttura di una rete neurale con uno strato nascosto

In ogni strato, ciascun neurone riceve input da altri neuroni o dall'esterno, elabora questi input per produrre un output, che viene restituito ad altri neuroni o all'esterno. Come mostrato nella figura 3.10, il neurone calcola l'output come la somma pesata dei suoi input, a cui si aggiunge un bias e successivamente applica una funzione di attivazione, indicata con σ . Matematicamente, per il j -esimo neurone, l'output y_j può essere espresso come:

$$y_j = \sigma\left(\sum_i w_{j,i} \cdot x_i + b_j\right)$$

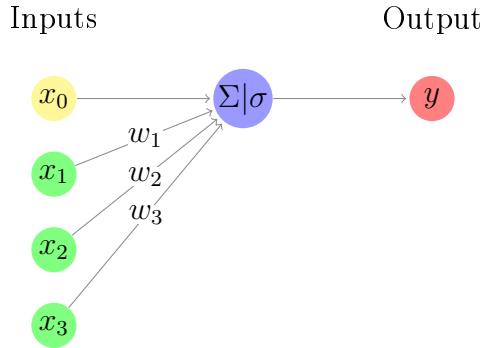


Figura 3.10: Struttura di un neurone

I parametri $w_{j,i}$ per ciascun input x_i e il bias b_j sono i parametri del neurone; i parametri di tutti i neuroni sono i parametri della rete. L'output di una rete è determinata dai parametri di ciascun neurone e dalle loro connessioni. I pesi vengono inizializzati semi-casuale. Durante l'addestramento della rete con esempi, si procede a regolare i parametri della rete per ottenere il comportamento desiderato.

Reti neurali convoluzionali

Nelle reti neurali, il modo in cui i neuroni sono connessi permette di eseguire diverse operazioni. Quando ogni neurone di uno strato è collegato a tutti quelli degli strati adiacenti, si parla di strati densi. Nelle reti convoluzionali, invece, i neuroni di uno strato sono connessi solamente a un sottoinsieme di quelli dello strato precedente, generalmente vicini (vedi figura 3.11).

Quest'operazione è equivalente a una convoluzione, dove un filtro (o feature detector) scorre attraverso l'input, con i pesi del filtro che vengono appresi durante l'addestramento. Ciò significa che, se i neuroni in uno strato convoluzionale condividono lo stesso insieme di pesi, ciò equivale ad applicare lo stesso filtro su tutti gli input. Gli output dei neuroni che condividono gli stessi pesi formano una "feature map". Il filtro utilizzato nella convoluzione corrisponde all'insieme dei pesi condivisi dai neuroni all'interno di una feature map. Uno

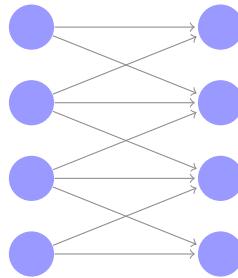


Figura 3.11: Visualizzazione delle connessioni tra i neuroni in uno strato convoluzionale

strato convoluzionale è composto da molteplici feature map, ciascuna con un proprio insieme di pesi, in modo che diverse feature possono essere estratte [19].

Questa struttura permette ai neuroni degli strati convoluzionali inferiori di estrarre feature elementari, mentre gli strati successivi combinano queste feature per ottenere rappresentazioni più complesse (vedi figura 3.12).

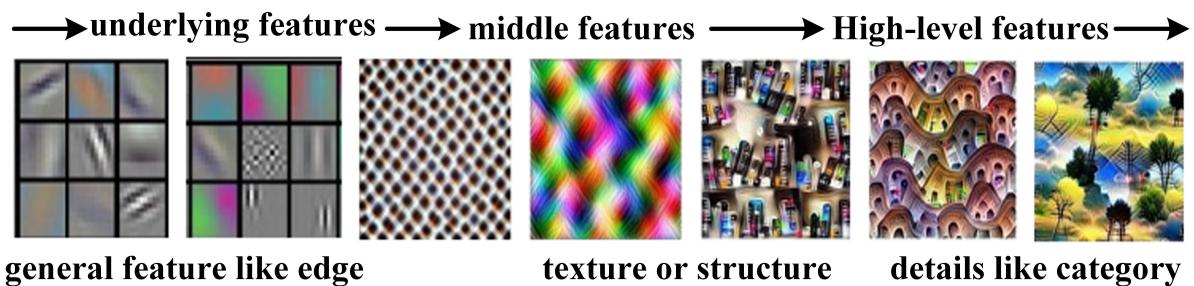


Figura 3.12: Estrazione delle feature da parte di una rete neurale convoluzionale [16]

Reti neurali ricorrenti

3.3.3 Autoencoder (Novelty detection)

Un'autoencoder è anch'esso un modello che viene addestrato per minimizzare l'errore di ricostruzione di un input dopo averlo compresso in uno spazio latente. Durante l'addestramento con un dataset “pulito”, l'encoder impara a comprimere gli esempi nello spazio latente, mentre il decoder impara a ricostruirli nella loro forma originale. Come visto nel paragrafo 3.2.4, quando si fornisce al autoencoder esempi non visti in fase di addestramen-

to, se questi vengono compressi in qualcosa di simile a ciò che ha già compresso nello spazio latente, allora riesce a ricostruire l'input con buona fedeltà. Altrimenti, non riuscirà a costruire correttamente l'input e l'errore di ricostruzione, definito come $\epsilon = \|x - g(f(x))\|$, sarà alto [2].

Per stabilire se un esempio è una novità, viene considerato l'errore di ricostruzione insieme alle etichette. Utilizzando un secondo dataset che comprende i job zombie, si imposta un valore soglia in base all'errore di ricostruzione: quelli che sono sopra la soglia sono job zombie (1) e quelli che sono al di sotto sono job normali (0) (vedi figura 3.13). Minimizziamo il valore soglia per avere un valore il più corretto possibile.

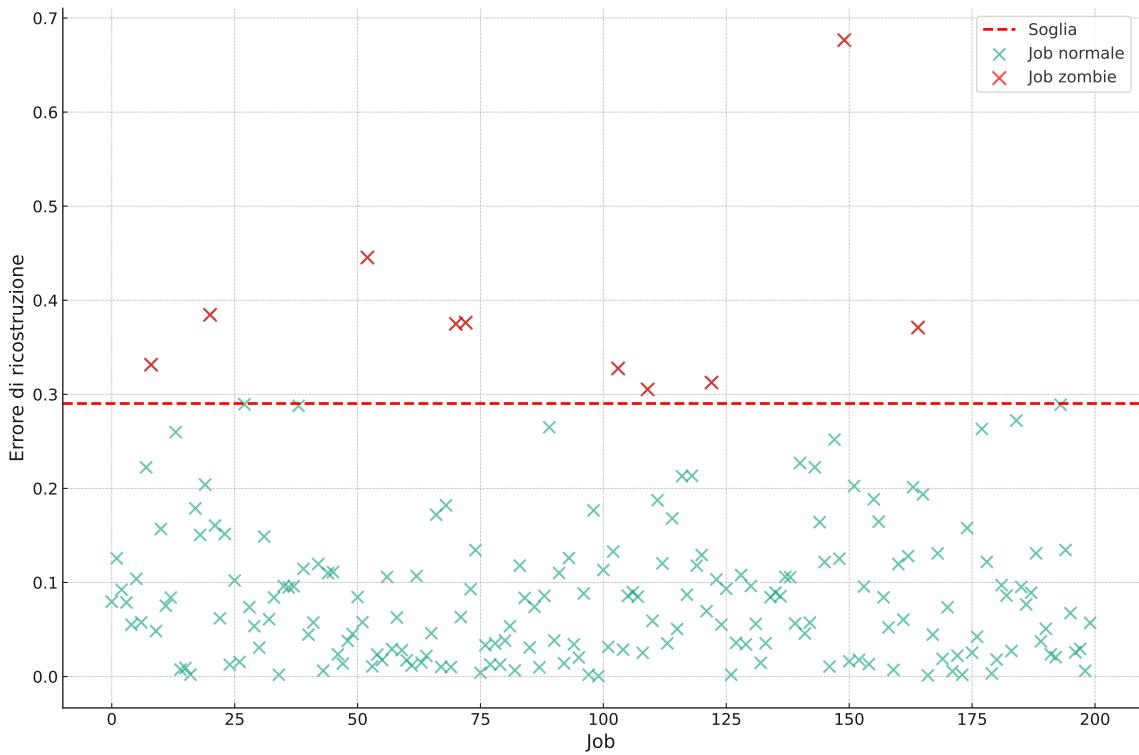


Figura 3.13: Visualizzazione dell'errore di ricostruzione e della soglia per stabilire se gli esempi sono novità

L'autoencoder essendo una rete neurale, composta da un encoder e un decoder, queste due parti descritte fino ad ora come funzioni, sono a loro volta delle strutture composte da strati e neuroni, che possono essere profondi quanto si vuole e con le architetture

precedentemente citate in modo da imparare la rappresentazione ideale dei dati, questo è importante, poiché siamo interessati a m

autoencoder are not limited to dense networks; you can build also convolutional , recurrent

ma cosa ben più importante è che siamo interessati a vedere dei dati nello spazio latente non nello spazio dei dati, ciò vuol dire che è importante per un autoencoder imparare in una rappresentazione h che impara le feature importanti.

- Undercomplete
- Denoise
- Sparse

Capitolo 4

Analisi dei risultati

4.1 Valutazione delle performance

4.1.1 Metriche di valutazione

4.1.2 Convalida incrociata

4.2 Confronto tra i modelli

4.3 Interpretazione dei risultati

Capitolo 5

Conclusioni e sviluppi futuri

5.1 Sintesi dei risultati

5.2 Limitazioni dello studio e proposte per ricerche future

Bibliografia

- [1] Anupong Banjongkan et al. «A Study of Job Failure Prediction at Job Submit-State and Job Start-State in High-Performance Computing System: Using Decision Tree Algorithms». In: *Journal of Advances in Information Technology* 12 (2021), pp. 84–92. URL: <https://api.semanticscholar.org/CorpusID:234326555>.
- [2] Andrea Borghesi et al. «Anomaly Detection Using Autoencoders in High Performance Computing Systems». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (lug. 2019), pp. 9428–9433. ISSN: 2159-5399. DOI: [10.1609/aaai.v33i01.33019428](https://doi.org/10.1609/aaai.v33i01.33019428). URL: <http://dx.doi.org/10.1609/aaai.v33i01.33019428>.
- [3] G Bortolotti et al. «The INFN Tier-1». In: *Journal of Physics: Conference Series* 396.4 (dic. 2012), p. 042016. DOI: [10.1088/1742-6596/396/4/042016](https://doi.org/10.1088/1742-6596/396/4/042016). URL: <https://dx.doi.org/10.1088/1742-6596/396/4/042016>.
- [4] L. Breiman. «Arcing the edge». In: 1997. URL: <https://api.semanticscholar.org/CorpusID:14849468>.
- [5] Jason Brownlee. *SMOTE for Imbalanced Classification with Python*. Machine Learning Mastery. Mar. 2021. URL: <https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/> (visitato il 15/11/2023).
- [6] Franck Cappello et al. «Toward Exascale Resilience: 2014 update». In: *Supercomputing Frontiers and Innovations* 1.1 (giu. 2014), pp. 5–28. DOI: [10.14529/jsfi140101](https://doi.org/10.14529/jsfi140101). URL: <https://superfri.org/index.php/superfri/article/view/14>.
- [7] CERN. *Worldwide LHC Computing Grid*. 2023. URL: <https://wlcg.web.cern.ch> (visitato il 28/10/2023).

- [8] Tianqi Chen e Carlos Guestrin. «XGBoost: A Scalable Tree Boosting System». In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. ACM, ago. 2016. DOI: [10.1145/2939672.2939785](https://doi.org/10.1145/2939672.2939785). URL: <http://dx.doi.org/10.1145/2939672.2939785>.
- [9] CNAF. *WLCG Tier-1 data center - Calcolo*. URL: <https://www.cnaaf.infn.it/calcolo/> (visitato il 28/10/2023).
- [10] Stefano Dal Pra et al. «Evolution of monitoring, accounting and alerting services at INFN-CNAF Tier-1». In: *EPJ Web of Conferences* 214 (gen. 2019), p. 08033. DOI: [10.1051/epjconf/201921408033](https://doi.org/10.1051/epjconf/201921408033).
- [11] Alberto Fernández et al. *Learning from Imbalanced Data Sets*. Gen. 2018. ISBN: 978-3-319-98073-7. DOI: [10.1007/978-3-319-98074-4](https://doi.org/10.1007/978-3-319-98074-4).
- [12] Jerome H. Friedman. «Greedy function approximation: A gradient boosting machine.» In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. DOI: [10.1214/aos/1013203451](https://doi.org/10.1214/aos/1013203451). URL: <https://doi.org/10.1214/aos/1013203451>.
- [13] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2^a ed. O'Reilly Media, 2019, pp. 266–267.
- [14] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [15] Rui Guo et al. «Degradation state recognition of piston pump based on ICEEM-DAN and XGBoost». In: *Applied Sciences* 10 (set. 2020), p. 6593. DOI: [10.3390/app10186593](https://doi.org/10.3390/app10186593).
- [16] Chu He et al. «Nonlinear Manifold Learning Integrated with Fully Convolutional Networks for PolSAR Image Classification». In: *Remote Sensing* 12 (feb. 2020), p. 655. DOI: [10.3390/rs12040655](https://doi.org/10.3390/rs12040655).
- [17] H. He e Y. Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*. Wiley, 2013. ISBN: 9781118646335. URL: <https://books.google.it/books?id=CVHx-Gp9jzUC>.
- [18] Diederik P Kingma e Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: [1312.6114 \[stat.ML\]](https://arxiv.org/abs/1312.6114).

- [19] Y. Lecun et al. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791).
- [20] Laurens van der Maaten e Geoffrey Hinton. «Visualizing Data using t-SNE». In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: <https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf>.
- [21] Gordon E. Moore. «Cramming more components onto integrated circuits». In: *Electronics* 38.8 (1965), pp. 114–117.
- [22] Nicola Pezzotti et al. «Approximated and User Steerable tSNE for Progressive Visual Analytics». In: *IEEE Transactions on Visualization and Computer Graphics* 23.7 (2017), pp. 1739–1752. DOI: [10.1109/TVCG.2016.2570755](https://doi.org/10.1109/TVCG.2016.2570755).
- [23] Marco A.F. Pimentel et al. «A review of novelty detection». In: *Signal Processing* 99 (2014), pp. 215–249. ISSN: 0165-1684. DOI: <https://doi.org/10.1016/j.sigpro.2013.12.026>. URL: <https://www.sciencedirect.com/science/article/pii/S016516841300515X>.
- [24] Andrea Rendina. *INFN-T1 site report*. https://indico.cern.ch/event/1200682/contributions/5087586/attachments/2538178/4368754/20221031_Infnt1_site_report.pdf. Accessed: 2023-10-28. 2022.
- [25] Irhum Shafkat. *Intuitively Understanding Variational Autoencoders*. Towards Data Science. Feb. 2018. URL: <https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf> (visitato il 15/11/2023).
- [26] J. M. Shalf e R. Leland. «Computing Beyond Moore’s Law». In: *Computer* 48.12 (dic. 2015), pp. 14–23. ISSN: 1558-0814. DOI: [10.1109/MC.2015.374](https://doi.org/10.1109/MC.2015.374).
- [27] Ravid Shwartz-Ziv e Amitai Armon. «Tabular Data: Deep Learning is Not All You Need». In: *CoRR* abs/2106.03253 (2021). arXiv: [2106.03253](https://arxiv.org/abs/2106.03253). URL: <https://arxiv.org/abs/2106.03253>.
- [28] Thomas N. Theis e H.-S. Philip Wong. «The End of Moore’s Law: A New Beginning for Information Technology». In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50. DOI: [10.1109/MCSE.2017.29](https://doi.org/10.1109/MCSE.2017.29).

- [29] Oreste Villa et al. «Scaling the Power Wall: A Path to Exascale». In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 830–841. DOI: [10.1109/SC.2014.73](https://doi.org/10.1109/SC.2014.73).
- [30] Martin Wattenberg, Fernanda Viégas e Ian Johnson. «How to Use t-SNE Effectively». In: *Distill* (2016). DOI: [10.23915/distill.00002](https://doi.org/10.23915/distill.00002). URL: <http://distill.pub/2016/misread-tsne>.