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 (Centro Nazionale delle Tecnologie Informatiche e Telematiche) dell'INFN (Istituto Nazionale di Fisica Nucleare) gestisce uno dei più importanti centri di calcolo in Italia, utilizzato da gruppi di ricercatori di fisica delle particelle, astrofisica e altri campi. Questo centro è 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 alcune grandezze, che vengono campionate ogni tre minuti e raccolte in un database insieme ai dati di accounting relativi ai job terminati. Questo studio esplora l'uso di tecniche di Machine Learning e Deep Learning per prevedere il successo o il fallimento dei job, basandosi sull'evoluzione del loro stato nel tempo. In particolare, è stato identificato un sottoinsieme di job che falliscono, denominati zombie. Questi, pur smettendo di effettuare calcoli, non rilasciano l'host fisico, occupando improduttivamente delle risorse fino al loro timeout. L'obiettivo della tesi è stato quello di individuare questi job il più presto possibile, poiché identificarli nelle loro fasi iniziali risulta essere particolarmente vantaggioso in termini di risparmio di risorse derivante dalla loro rimozione. Sono stati proposti e validati due modelli capaci di identificare i job che, con buona probabilità, diventeranno zombie (1 su 2). Le predizioni fornite dal modello possono essere utilizzate per impostare un filtro o un avviso, permettendo così di controllare manualmente i job sospetti o di stabilire una regola per la loro eliminazione.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Caso di studio

Questo elaborato descrive il lavoro di tirocinio e tesi svolto presso il CNAF (Centro Nazionale delle Tecnologie Informatiche e Telematiche) dell'INFN (Istituto Nazionale di Fisica Nucleare), uno dei più importanti centri di calcolo in Italia. Questo centro processa e gestisce decine di petabyte di dati. L'obiettivo è stato quello di analizzare questi dati per identificare un problema che potesse essere risolto tramite l'uso di tecniche di Machine Learning e Deep Learning.

In questo capitolo forniremo una panoramica del centro di calcolo, guardando da dove provengono i dati e come vengono raccolti. Verrà infine presentato il problema che è stato oggetto della nostra analisi, spiegando le motivazioni alla base di questa scelta.

1.1 Il cluster di calcolo del CNAF

Il **Grid computing** è un'architettura di calcolo distribuito che collega centri di calcolo 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 di Fisica delle Alte Energie effettuati con il Large Hadron Collider (LHC) [8].

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 CNAF, che ospita il Tier-1 per tutti e quattro gli esperimenti del LHC. Oltre a questi ultimi, vengono supportati presso il CNAF anche alcuni esperimenti non-LHC di astrofisica delle particelle e fisica dei neutrini [3].

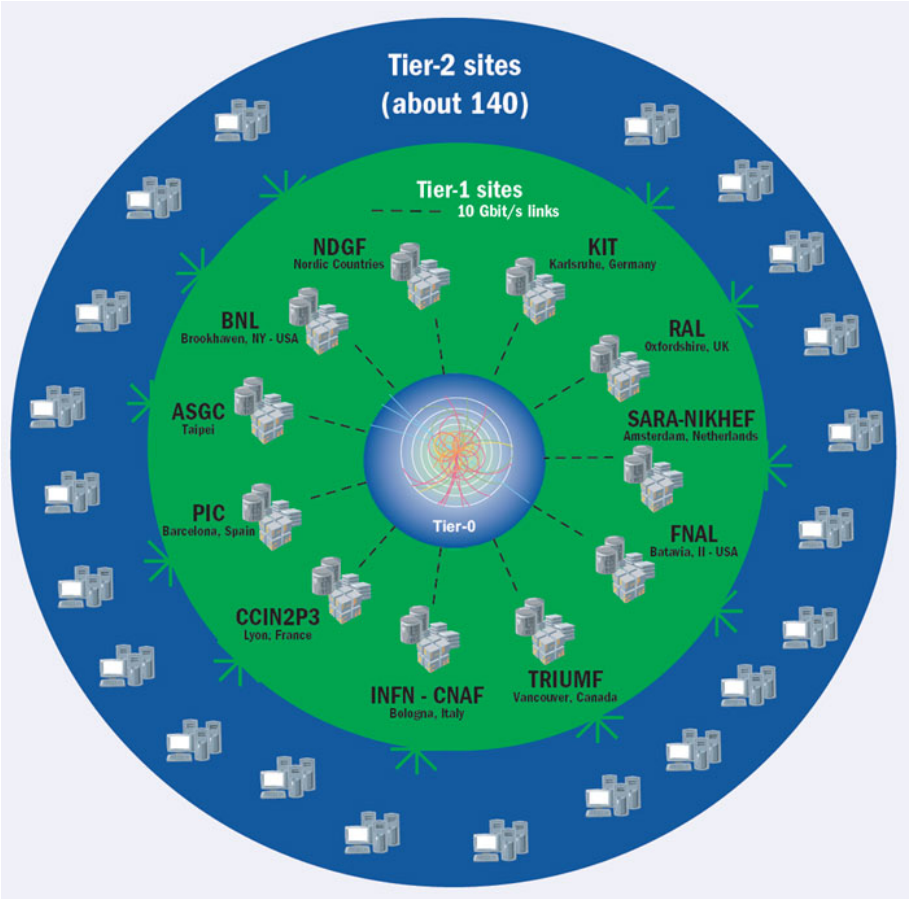

Figura 1.1: Struttura gerarchica del WLCG [11]

Il CNAF dispone di più di 46000 core distribuiti su 960 host fisici per un totale di circa 630 kHS06[1] di potenza di calcolo [31]. L'allocazione di queste risorse segue il paradigma del **High-Throughput Computing** (HTC), dove a differenza dell'High-Performance

[1]Benchmark utilizzato per confrontare le risorse di calcolo in ambito scientifico.

Computing, che mira a ridurre il wall-clock time dei programmi, HTC mira a 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 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 [10].

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 dello stato 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 e che rappresentano lo stato finale dei job.

Successivamente uno script estrae le informazioni rilevanti dai dati di accounting; queste informazioni vengono poi inserite nella tabella `htjob` di un database PostgresSQL. 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.

	htm	
	Righe	**Spazio (GB)**
`hj`	1971830783	343
`htjob`	30799153	14

	htmnew	
	Righe	**Spazio (GB)**
`hj`	1038471316	222
`htjob`	46605815	22

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

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 supererebbe la capacità della memoria RAM a disposizione, è necessario selezionare un sottoinsieme di dati da tali database per effettuare le analisi successive.

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

I sistemi HTC e HPC sono divenuti 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 [37]. Per proseguire l'evoluzione tecnologica nell'era post-legge di Moore [33, 35] 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**
`ts`	Numerico (secondi)	Timestamp UNIX del momento in cui lo stato del job viene misurato
`jobid + idx`	Categorico (Nominale)	ID univoco del job
`queue`	Categorico (Nominale)	Gruppo di appartenenza dell'utente che ha sottomesso il job
`hn (hostname)`	Categorico (Nominale)	Host sul quale il job è in esecuzione
`js`	Categorico (Nominale)	Stato del job: 1 = In coda, 2 = In esecuzione, 3 = Rimosso, 4 = Completato, 5 = Sospeso
`nc`	Numerico (core)	Numero di core CPU impiegati dal job
`hsj`	Numerico (HS06)	Potenza di un core del host
`hsm`	Numerico (HS06)	Potenza totale del host
`cpt (cputime)`	Numerico (secondi)	Tempo di esecuzione sulla CPU del job
`rt (runtime)`	Numerico (secondi)	Tempo totale di esecuzione del job
`owner`	Testo	Utente che ha sottomesso il job (username UNIX)
`rss`	Numerico (KB)	Memoria RAM utilizzata dal job
`swp`	Numerico (KB)	Memoria SWAP utilizzata dal job
`sn (submitnode)`	Categorico (Nominale)	Nodo da cui è stato sottomesso il job
`disk`	Numerico (GB)	Spazio su disco utilizzato dal job

Tabella 1.3: Schema della tabella `htjob` del database

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

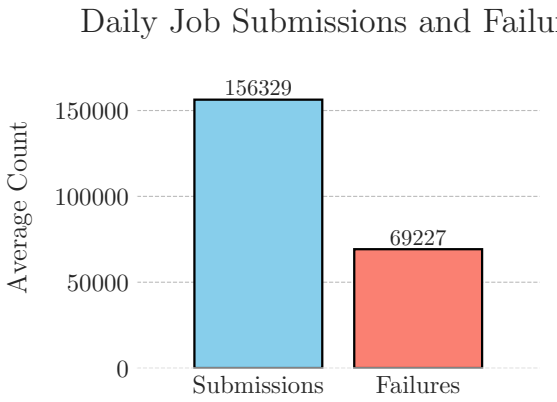

Figura 1.2: Media giornaliera di job sottomessi e falliti nel mese di Marzo 2023

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 circa 150000 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 coda.

Questa tesi si concentra 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, si descrive come l'analisi preliminare abbia evidenziato una categoria di job che falliscono, che risulta essere particolarmente interessante, soprattutto per l'importanza di identificarli e rimuoverli tempestivamente.

Nel capitolo 3, si tratta dell'estrazione e della preparazione dei dati per l'addestramento di modelli predittivi. In seguito, si esplorano i modelli specifici per due modellazioni diverse del problema di identificazione di questa categoria di job.

Nel capitolo 4, si introduce inizialmente una metodologia per valutare correttamente i risultati ottenuti. Successivamente, si discutono i risultati derivanti dall'applicazione dei modelli, illustrando due approcci differenti utilizzati.

Nel capitolo 5, si fornisce una sintesi dei risultati raggiunti e si propongono potenziali direzioni per ulteriori sviluppi e miglioramenti.

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 verosimilmente ridurre lo spreco delle risorse 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 che sono in esecuzione al tempo in cui la grandezza è misurata da HTCondor, ogni tupla può essere considerata come un punto in una serie storica multivariata. In altre parole, a ciascun job corrisponde 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 falli-

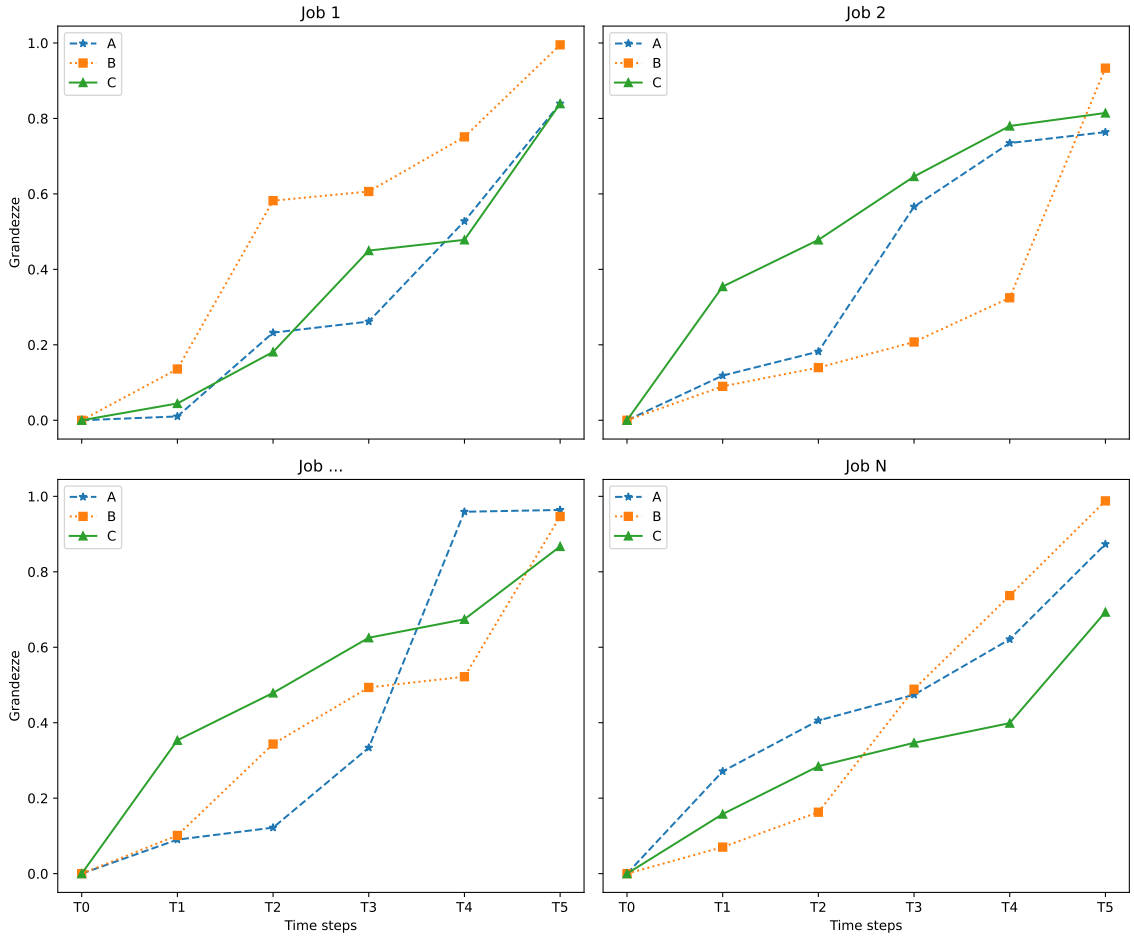

Figura 2.1: Rappresentazione di un job come serie storica multivariata

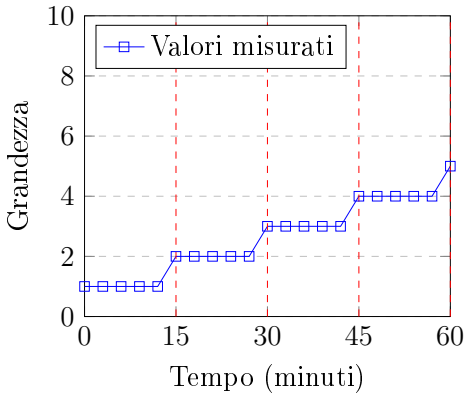

Figura 2.2: Frequenza di campionamento e aggiornamento dei job in HTCondor

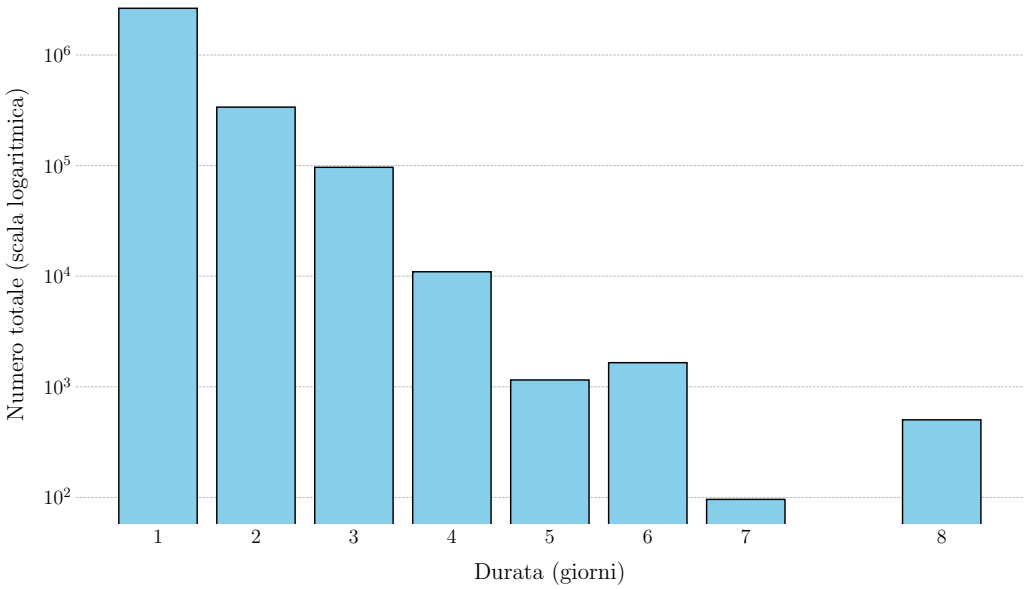

Figura 2.3: Distribuzione della durata dei job in giorni

Tabella 2.1: Percentuale di job in attesa rimossi senza aver effettuato alcun calcolo su un host fisico nel mese di Marzo 2023

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

mento di un job di lunga durata è nettamente più importante rispetto alla previsione del fallimento di un job di breve durata.

2.1.2 Caratterizzazione dei gruppi

Ricordando che gli utenti che sottomettono job al sistema sono raggruppati in base al loro coinvolgimento in specifici esperimenti scientifici, l'analisi relativa ai vari 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.

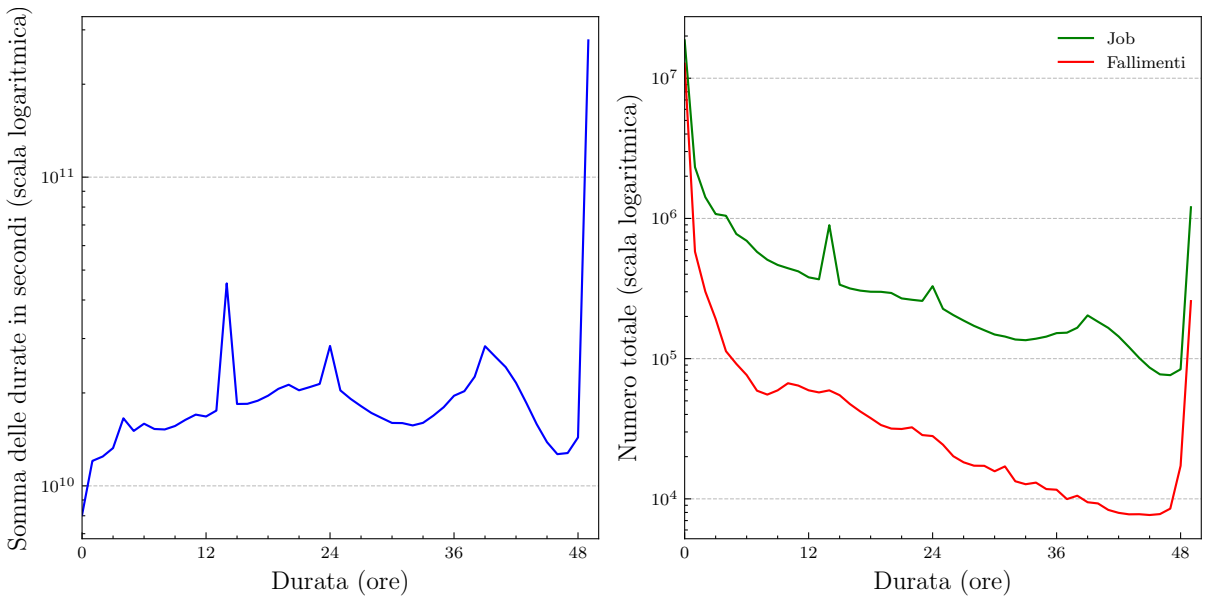

Figura 2.4: A sinistra, durata cumulativa dei job; a destra, numero totale di job e relativi fallimenti per ogni fascia oraria fino a 48 ore, mostrati su scala logaritmica

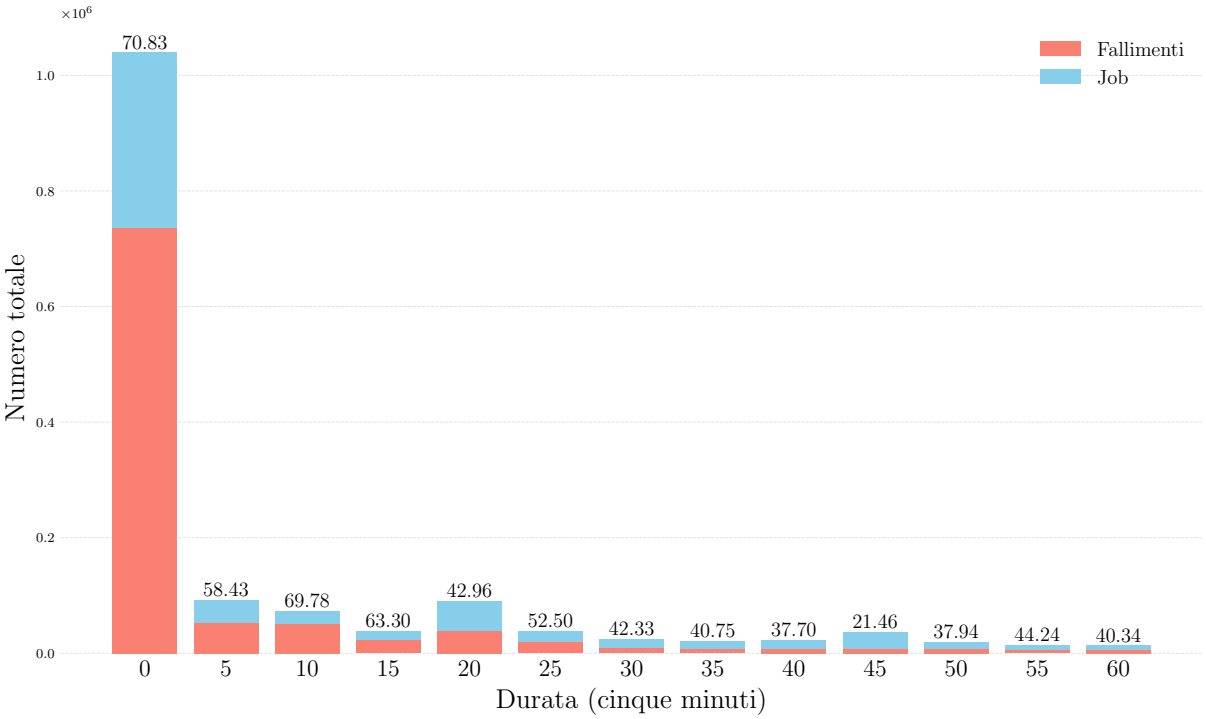

Figura 2.5: Distribuzione dei job nella prima ora, suddivisi in intervalli di cinque minuti

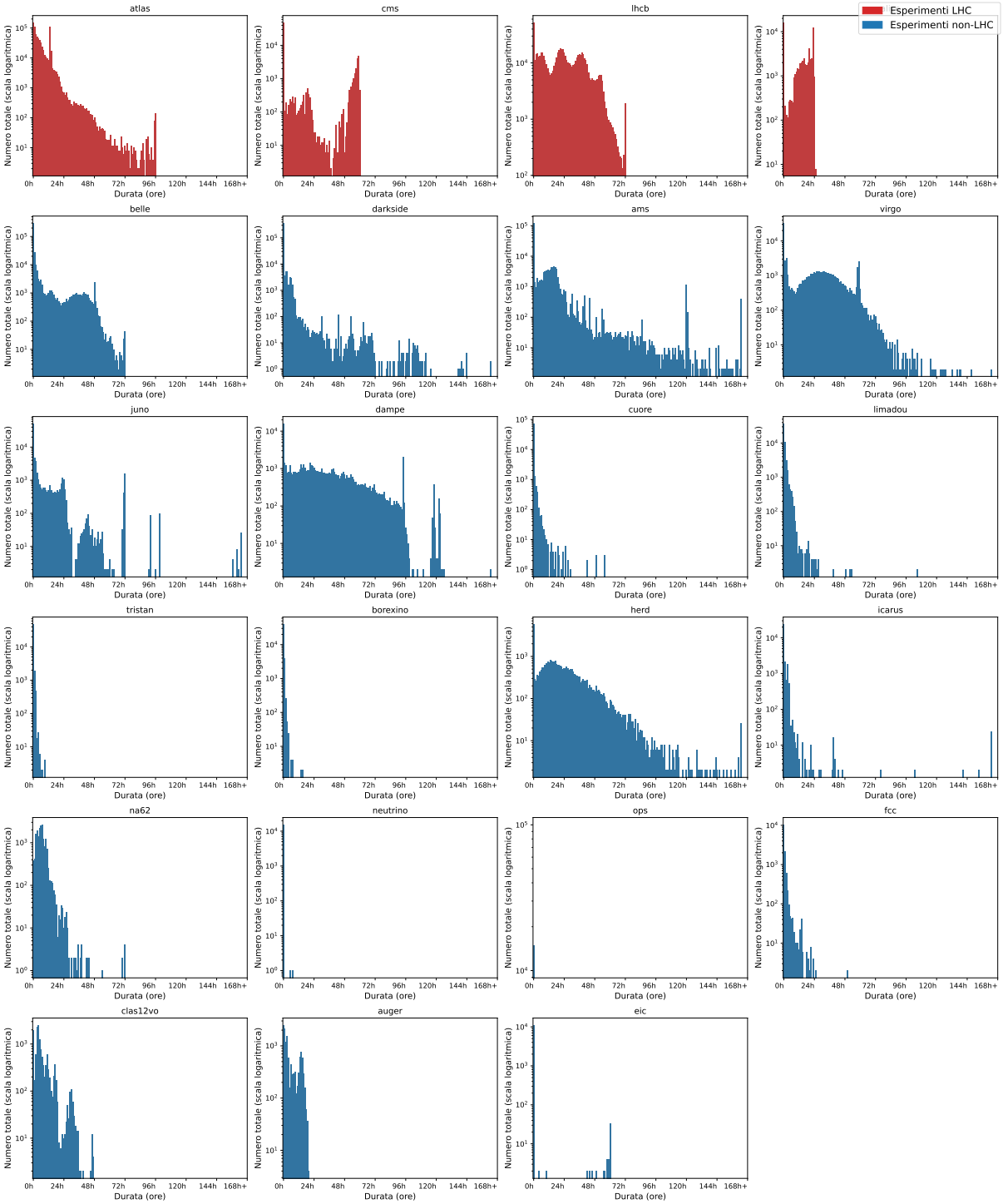

Figura 2.6: Distribuzione della durata dei job per gruppo

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 più 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 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.

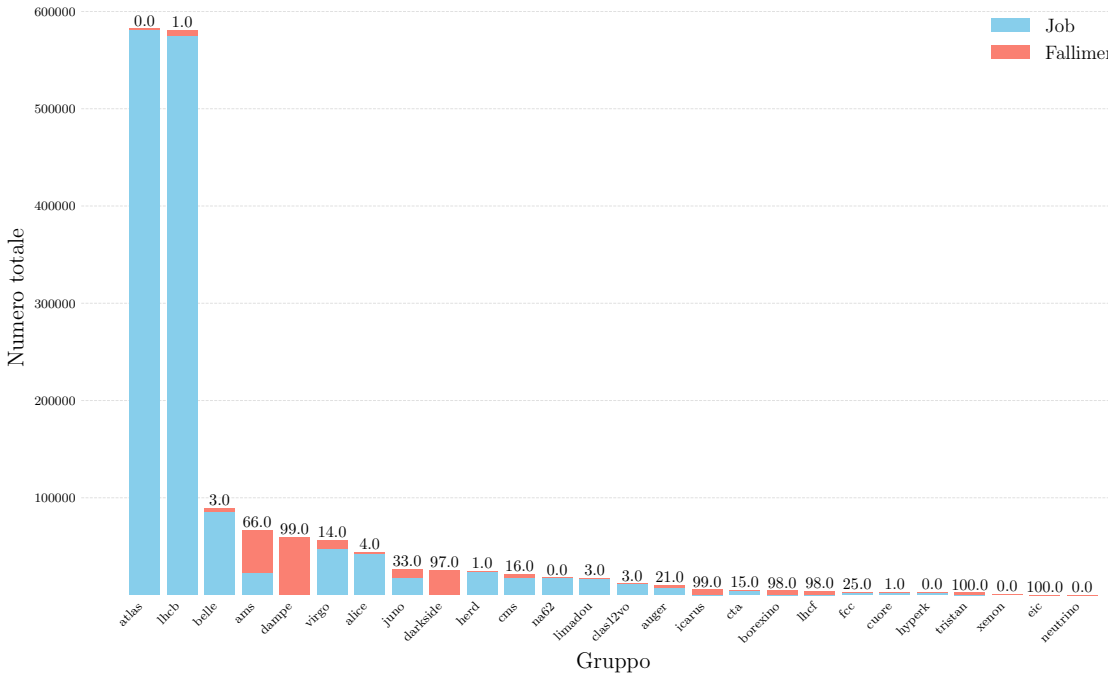

Figura 2.7: Distribuzione del numero totale dei job e dei loro fallimenti per gruppo

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 servono piuttosto 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.

Figura 2.8: Rappresentazione dei job in base alla loro durata

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 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 i nodi "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, potenzialmen-

te 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 permette di rilevare i cosiddetti "job zombie", è essenziale verificare preliminarmente la presenza di cluster ben definiti all'interno del dataset. Idealmente, vorremmo distinguere con precisione questi job anomali dagli altri e per questo scopo possiamo utilizzare l'algoritmo t-Distribuited Stochastic Neighbor Embedding (t-SNE).

t-SNE consente di ridurre la dimensionalità dei dati preservando la vicinanza tra i punti simili e distanziando quelli dissimili [27]. 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 $\mathcal{O}(N^2)$ [28], 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

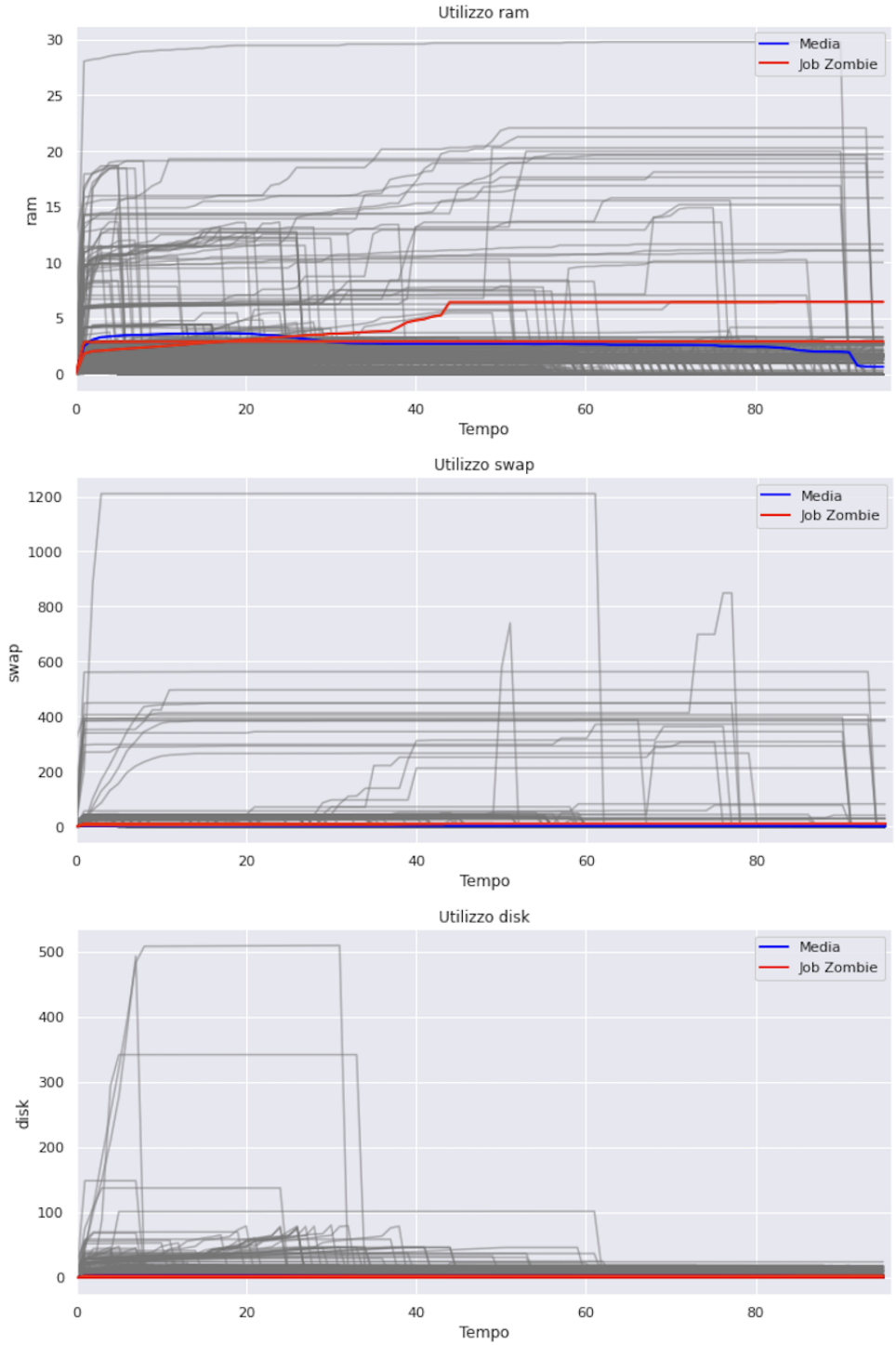

Figura 2.9: Utilizzo di RAM, SWAP e disco su intervalli di 15 minuti nelle prime 24 ore

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 [16].

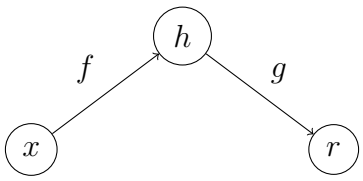

Figura 2.10: Struttura generica di un autoencoder [16]

Se i dati sono caratterizzati da relazioni non lineari, gli autoencoder con funzioni di 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 [16].

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 accumulatisi 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 [38]. 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 approcci 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.

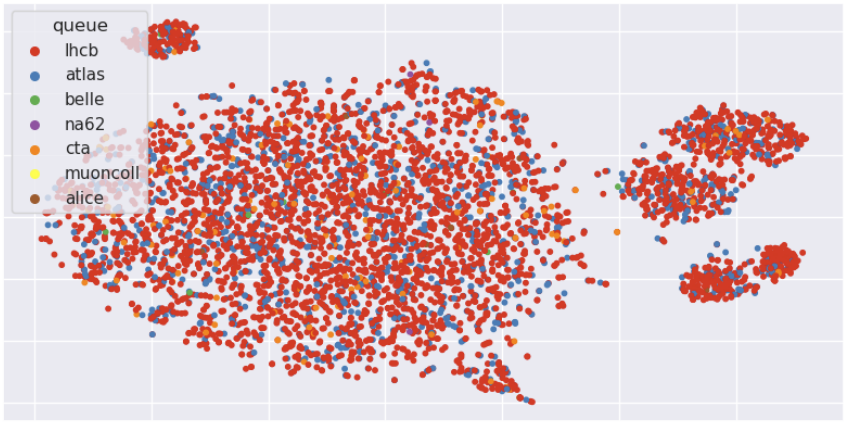

Figura 2.11: Visualizzazione job zombie del 2021 tramite t-SNE

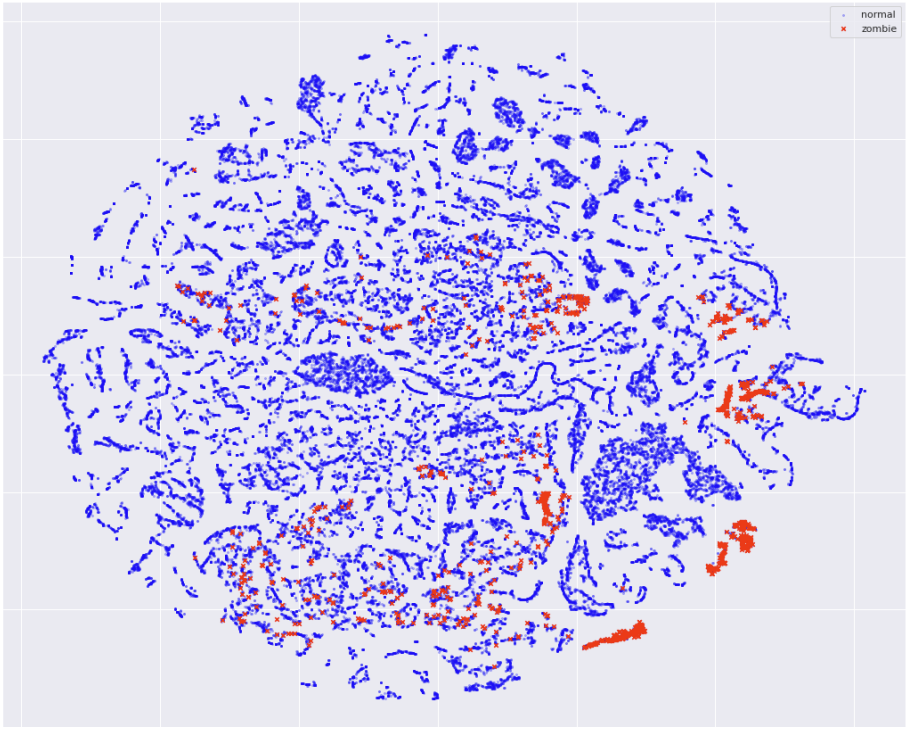

Figura 2.12: Visualizzazione job ATLAS di settembre 2021 tramite t-SNE

Capitolo 3

Tecniche di Machine Learning e Deep Learning

In questo capitolo ci focalizzeremo sulla parte del Machine Learning che si occupa del processo di creazione di modelli in grado di apprendere autonomamente dai dati. Un modello è costituito da un insieme di parametri e una struttura di calcolo che tratta 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 in modo da rendere minima una funzione di perdita[1]. 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 migliorarne 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 con un Grafo Aciclico Diretto (DAG), dove i dati fluiscono in una sola direzione, evitando

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

cicli, e dove ogni nodo in questo grafo rappresenta una fase distinta del processo (vedi figura 3.1).

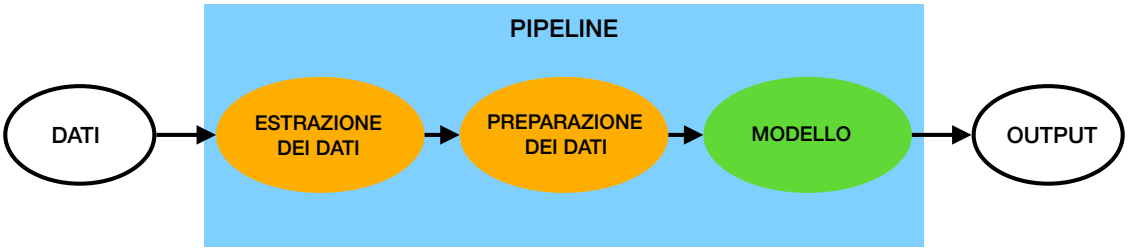

Figura 3.1: Rappresentazione di una pipeline di Machine Learning come un grafo orientato senza cicli

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

Efficacia **Estensione della ricerca degli iperparametri ad altri componenti**: Mentre l'individuazione dei parametri ottimali 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, sebbene questo possa comportare un aumento del tempo di addestramento.

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 sullo stato finale dei job, tra cui l'esito, 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 (*jobid.idx_submitnode*) e il timestamp. Questo timestamp sfrutta l'indice presente nella tabella `hj` per gestire in maniera efficiente le sue grandi dimensioni[2]. 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 array di valori.
- Seleziona 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:

[2]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.

- `job`: identificativo univoco per ogni job.
- `queue`: gruppo di appartenenza dell'utente che ha sottomesso il job.
- `fail`: un valore booleano che indica se il job è fallito.
- `mint` e `maxt`: Timestamp UNIX del momento in cui il job è stato eseguito e del momento in cui è terminato.
- `t`, `ram`, `swap`, `disk`: array di valori che rappresentano le serie storiche di misurazioni.

	job	queue	fail	mint	maxt	t	ram	swap	disk
0	9242718.0_ce04-htc	juno	0	1678695481	1678744082	[80, 80, 80, 80, 261, 261, 261, 261, 440, 440,...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...
1	9242720.0_ce04-htc	juno	0	1678695481	1678721221	[80, 80, 80, 80, 260, 260, 260, 260, 440, 440,...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...
2	9242734.0_ce04-htc	juno	0	1678695481	1678747322	[79, 79, 79, 79, 260, 260, 260, 260, 439, 439,...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...
3	9242733.0_ce04-htc	juno	0	1678695481	1678742461	[79, 79, 79, 79, 260, 260, 260, 260, 439, 439,...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...
4	9242731.0_ce04-htc	juno	0	1678695481	1678710781	[79, 79, 79, 79, 260, 260, 260, 260, 439, 439,...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...	[7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-06, 7e-...

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 stato integrato nella pipeline, in quanto, nell'ambiente operativo reale, si prevede che questa 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 disponibili in formato numerico, 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.

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 [14], 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.

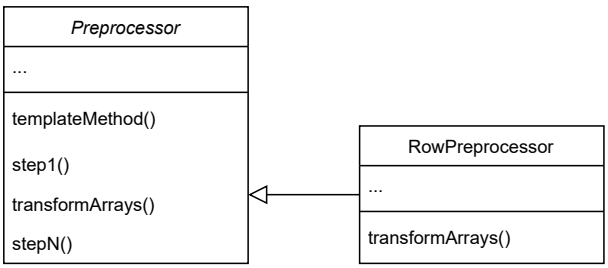

Figura 3.3: Diagramma UML della classe `Preprocessor`, illustrante l'implementazione del design pattern Template Method

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 array 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 convoluzione[3] 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 array, il dataset rappresenta ancora le tre dimensioni delle multiple serie multivariate: job (righe), variabili (colonne) e time step (array), come illustrato nella figura 3.4. Una

[3]La convoluzione è una operazione matematica che consiste nell'applicare un filtro di dimensione finita lungo una 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:

$$(S * K)(i) = \sum_{m=0}^{n} S(m) \cdot K(i - m)$$

dove S è la sequenza originale, K è il filtro e n è la dimensione del filtro.

possibile soluzione potrebbe essere quella di trasformare gli array 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 possano riflettere la tridimensionalità originale dei dati. Tuttavia, prima di procedere, è necessario assicurarsi che tutte le serie storiche siano uniformate alla stessa lunghezza.

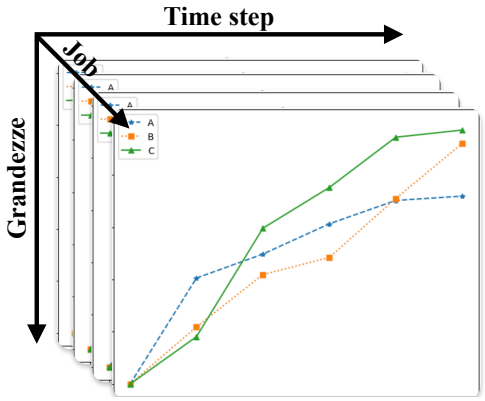

Figura 3.4: Rappresentazione tridimensionale delle multiple serie storiche multivariate

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. Il troncamento, invece, 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 array diventa una colonna (feature) 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 negli array 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` ($\mathcal{O}$(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. Ridimensionare i dati è un passo cruciale per migliorare la convergenza durante l'addestramento dei modelli [25]. Questo passaggio è essenziale per alcuni di questi, come le reti neurali, dove il ridimensionamento dei dati è una precondizione. Al contrario, altri modelli, quali gli alberi decisionali, non richiedono specifiche assunzioni sui dati.

Sono state utilizzate due tecniche di ridimensionamento: la normalizzazione e la standardizzazione. La normalizzazione è il processo che ridimensiona i dati in un intervallo 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 $X' = \frac{X - \mu}{\sigma}$ dove μ è la media e σ la deviazione standard.

La decisione di quale tecnica utilizzare dipende dalla specifica natura del problema affrontato, in quanto diversi problemi possono beneficiare di diverse tecniche di ridimensionamento. In pratica, andremo a testare entrambe le tecniche per determinare quale funziona meglio con il modello in uso.

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 della funzione di perdita associata, 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 vengono rimossi dal batch system dopo aver superato il massimo tempo di esecuzione, cioè 3 giorni per i job grid, e 7 per quelli locali.

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, sovracampionamento 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 [20].

Sovracampionamento. Un'alternativa è il sovracampionamento 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 attraverso tecniche di data augmentation, che creano varianti dei job zombie, offrendo così una gamma più ampia di esempi da cui il modello può apprendere [5]. Va tuttavia considerato che il sovracampionamento 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 [12].

Per questo scopo, si è scelto di usare una variante dell'autoencoder, nota come **variational autoencoder** [23], per generare varianti di job zombie a partire dai job zombie dell'intero 2021. Gli autoencoder tradizionali, come discusso nella sezione 2.2, apprendono due funzioni: un encoder, che mappa gli esempi in punti nello spazio latente, e un decoder, che mappa le 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 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) [32].

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; è invece un falso negativo quando il modello classifica 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 totale[4] derivante dalle predizioni errate, il modello imparerà durante l'addestramento ad attribuire diversa importanza ai vari tipi di errori.

[4] Costo totale $= C_{FN} \cdot FN + C_{FP} \cdot FP$

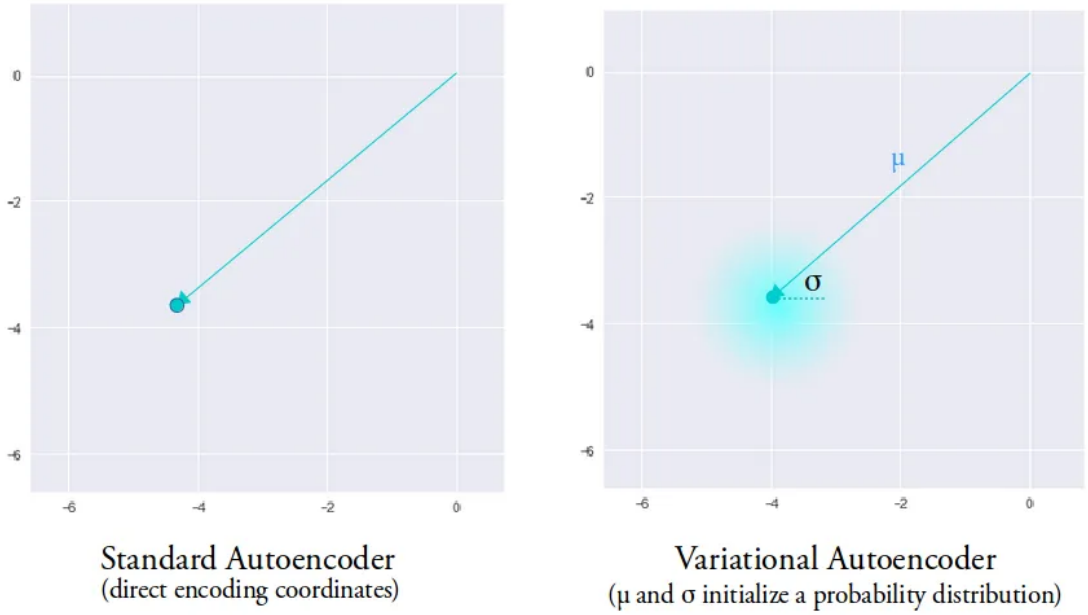

Figura 3.6: Rappresentazioni compresse in un autoencoder tradizionale e un variational autoencoder [32]

3.3 Selezione 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 $f : \mathbb{R}^n \to \{1, \ldots, k\}$ basandosi sulle feature degli esempi in input [16].

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 [15, 29].

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).

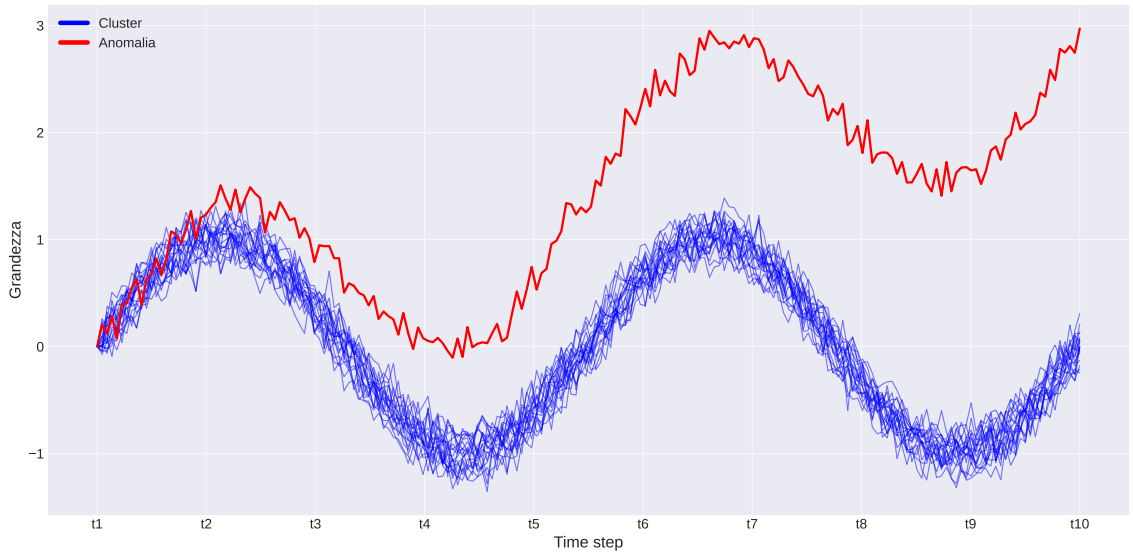

Figura 3.7: Rappresentazione ideale di una serie storica anomala in contrasto con un cluster di serie storiche normali

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-

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, 13].

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 descritta 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 obiettivo[5]:

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

che è la somma delle funzioni di perdita e dei termini di regolarizzazione per i t alberi. Il termine di regolarizzazione, introdotto da XGBoost come miglioramento al Gradient Boosting insieme ad altre ottimizzazioni, contribuisce a ridurre il rischio di overfitting [9].

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.

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

[5] La funzione che andremo a minimizzare durante l'addestramento.

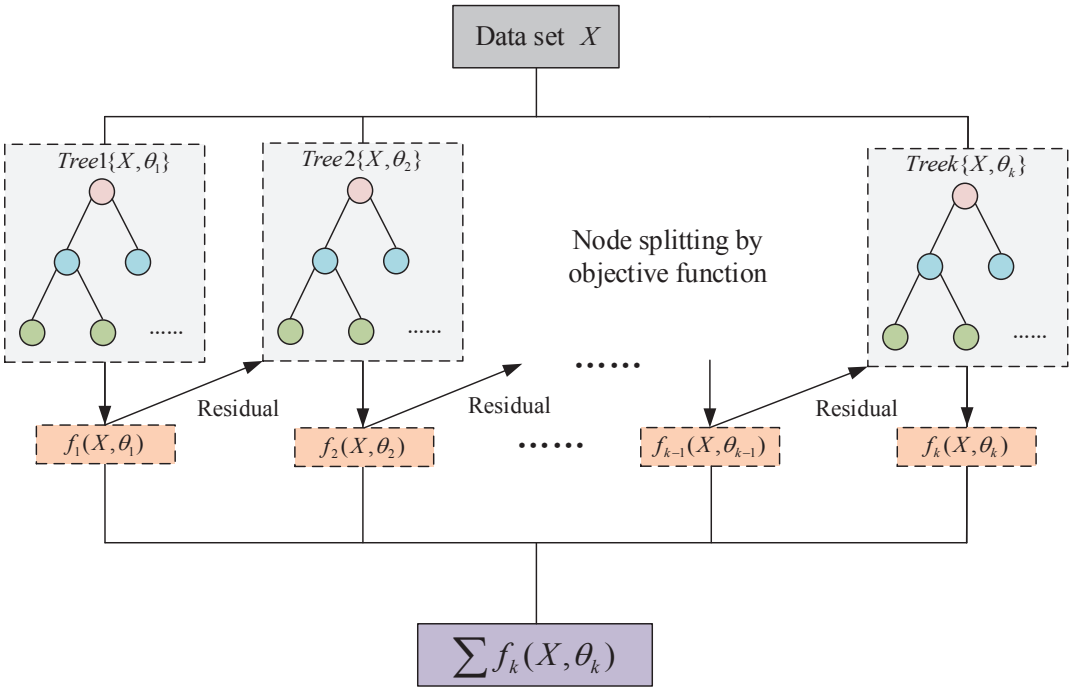

Figura 3.8: Struttura semplificata di XGBoost [17]

comparativi, XGBoost risulta essere un'ottima prima scelta nella trattazione di problemi reali, grazie alla sua efficacia nel trattare dati strutturati [34, 9].

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. Altri 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

collegati a quelli degli strati adiacenti tramite connessioni pesate (vedi figura 3.9).

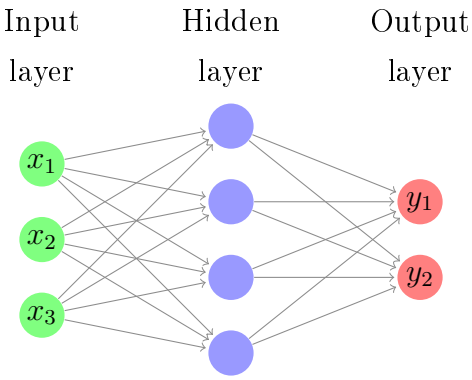

Figura 3.9: Struttura di una rete neurale con uno strato nascosto

In ogni strato, ciascun neurone riceve input da altri neuroni o dall'esterno, elabora questi input per produrre un output, che viene passato 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:

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

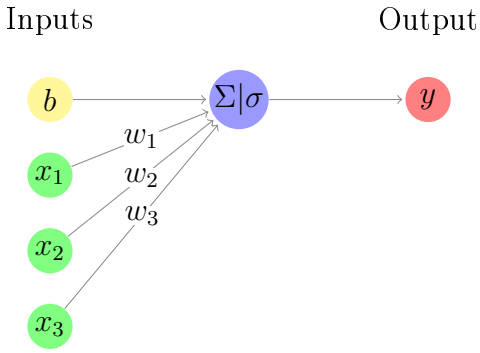

Figura 3.10: Struttura di un neurone

Dove i pesi $w_{j,i}$ per ciascun input x_i e il bias b_j sono i parametri del neurone. Considerando tutti i neuroni, questi sono i parametri complessivi della rete. I pesi vengono

inizializzati in modo semi-casuale e, durante l'addestramento, si procede a regolare i parametri della rete per ottenere l'output 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).

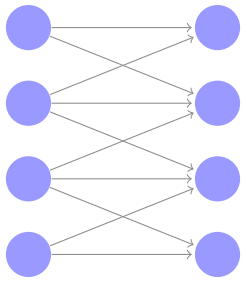

Figura 3.11: Visualizzazione delle connessioni tra i neuroni in uno strato convoluzionale

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 strato convoluzionale è composto da molteplici feature map, ciascuna con un proprio insieme di pesi, in modo che diverse feature possono essere estratte [24].

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).

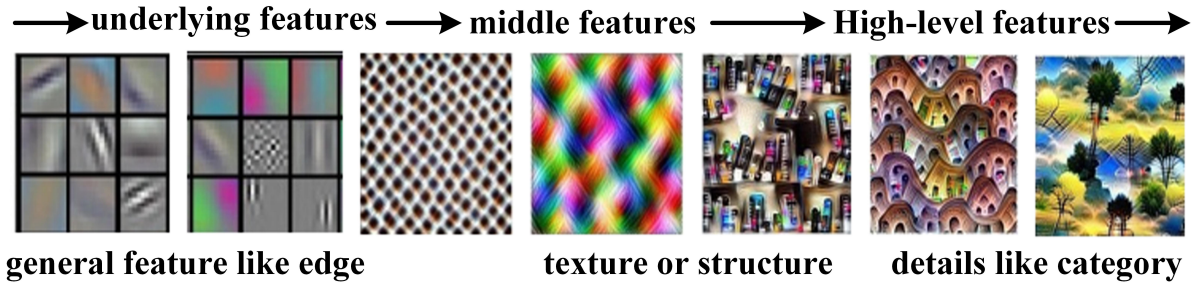

Figura 3.12: Estrazione delle feature da parte di una rete neurale convoluzionale [19]

Reti neurali ricorrenti e Transformers

Le RNN hanno la capacità di "ricordare" la storia degli input processati in uno stato, utilizzandola per elaborare nuovi input e produrre output. Questo le rende particolarmente adatte per le serie temporali, dove eventi passati potrebbero essere importanti per prevedere eventi futuri.

Come illustrato dalla figura 3.13, Una RNN può essere rappresentata come un neurone che riceve un input, genera un output e poi riutilizza questo output come input per il passo successivo. Le RNN lavorano con sequenze di input, come per esempio $(x_{(0)}, x_{(1)}, \ldots)$, dove ciascun elemento corrisponde a un time step. Ad ogni time step t, il neurone riceve l'input $x_{(t)}$ e l'output del time step precedente $y_{(t-1)}$ (se è il primo input, riceve 0). Ogni neurone delle RNN ha due vettori di pesi: W_x per gli input $x_{(t)}$ e W_y per gli output del time step precedente $y_{(t-1)}$. L'output di ogni time step $y_{(t)}$ si calcola come $y_{(t)} = \sigma_h(W_x \cdot x_{(t)} + W_y \cdot y_{(t-1)} + b)$ [15].

Elaborata l'intera sequenza di input, lo stato della rete viene resettato.

Le RNN presentano un problema significativo: la loro capacità di trattenere informazioni negli stati è limitata principalmente al time step successivo. Ciò implica che l'influenza di un valore specifico nella sequenza tende a svanire rapidamente, rendendo difficile catturare dipendenze a lungo termine nelle serie temporali. Per ovviare a questo problema, sono stati sviluppate varianti di RNN, come le Long Short-Term Memory (LSTM) [22] o altri tipi di reti neurali, quali i Transformer [36].

I Transformer, diversamente delle RNN, processano l'intera sequenza in un unico passaggio, apprendendo le relazioni tra i vari elementi della sequenza attraverso i meccanismi

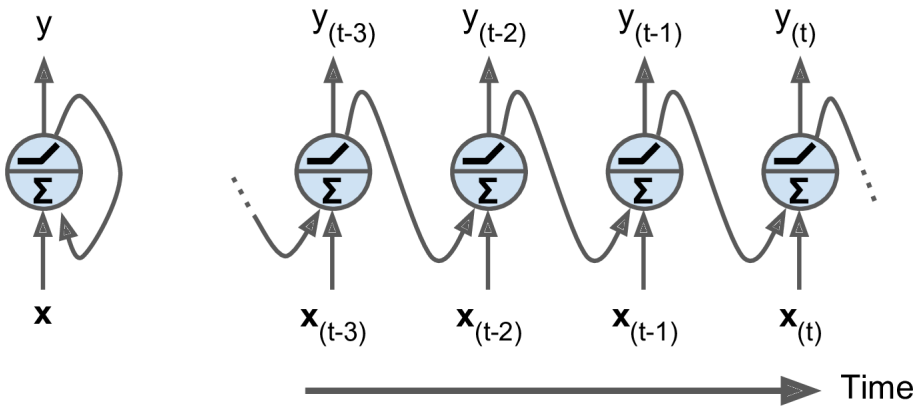

Figura 3.13: Struttura di una semplice rete neurale ricorrente [15]

del multi-head attention e il positional encoding. A differenza delle RNN, che elaborano la sequenza un passo alla volta, i Transformer non conoscono l'ordine dei valori all'interno di essa. Di conseguenza, è necessario incorporare tale ordine attraverso il positional encoding. Il meccanismo di self attention, d'altra parte, pesa l'importanza di ciascun valore all'interno di una sequenza rispetto a tutti gli altri, indipendentemente dalla loro distanza nella sequenza stessa, permettendo così al modello di concentrarsi su specifici valori.

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 all'autoencoder esempi non visti in fase di addestramento, 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. In caso contrario, non riuscirà a costruire correttamente l'input e l'errore di ricostruzione, definito come $\epsilon = \|x - g(f(x))\|$, sarà eccessivo [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 3.14). Minimizziamo il valore soglia per avere un valore il più corretto possibile.

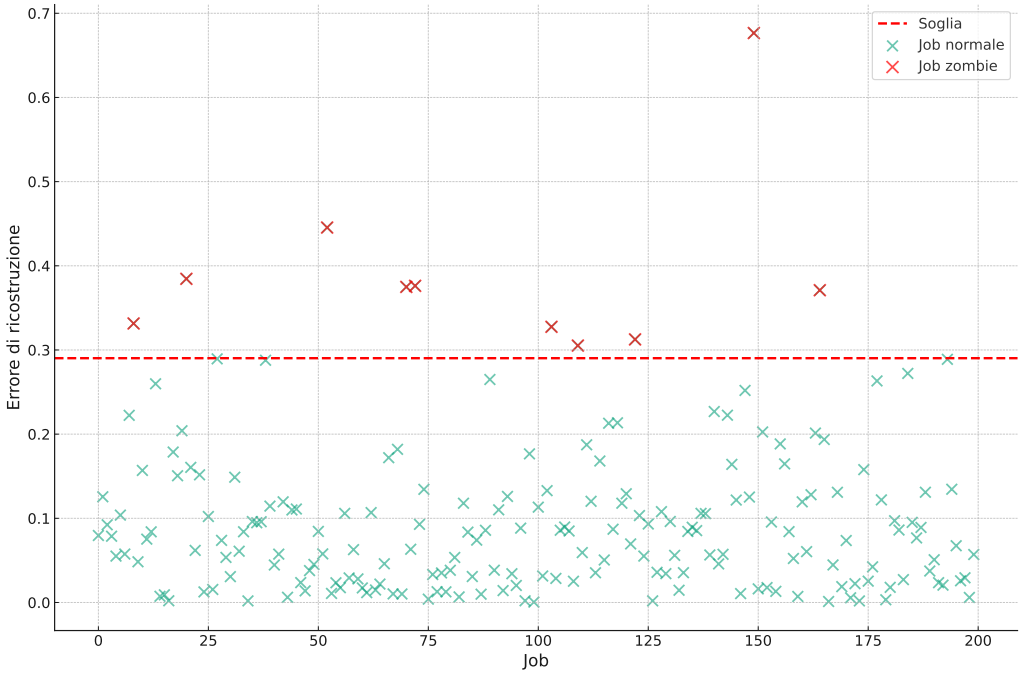

Figura 3.14: Visualizzazione dell'errore di ricostruzione e della soglia per stabilire se gli esempi sono novità

Gli autoencoder, descritti finora in termini di funzioni encoder f e decoder g, sono costituiti da strati e neuroni. L'output dello strato intermedio rappresenta la rappresentazione nello spazio latente. L'obiettivo è quello di formare cluster efficaci in questo spazio latente, al fine di poter associare nuovi job, mai incontrati prima, a gruppi di job esistenti. Tipicamente, per questo scopo, gli autoencoder sono strutturati con una forma a clessidra (vedi figura 3.15), o possono introdurre del rumore negli input, oppure ancora aggiungere un termine di regolarizzazione alla funzione di perdita per "spegnere" alcuni neuroni nello strato intermedio.

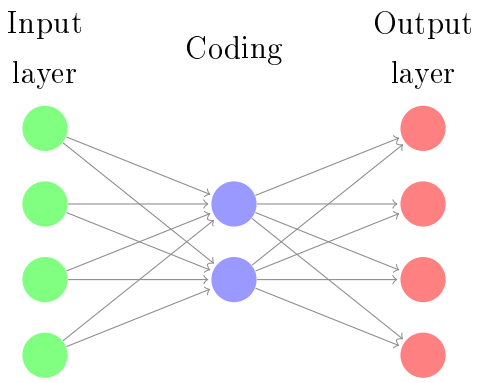

Figura 3.15: Struttura di un autoencoder a clessidra

Capitolo 4

Analisi dei risultati

Ricordando quanto detto nella sezione 2.2, l'obiettivo di questa tesi è verificare se i job zombie possano essere identificati utilizzando modelli di Machine Learning.

Come descritto nella sezione 3.2.4, un classificatore potrebbe avere un errore apparentemente basso semplicemente prevedendo l'etichetta più frequente. Infatti, considerando la proporzione di un job zombie ogni 10,000 job normali, un classificatore che etichetta tutti i job come normali raggiungerebbe un'accuratezza del 99.99%, dove per accuratezza si intende il rapporto tra il numero di predizioni corrette e il numero totale di predizioni effettuate. Un simile modello, che assume tutti i job come normali, equivarrebbe a non avere un modello di Machine Learning. Pertanto, per valutare la fattibilità nell'identificazione dei job zombie, useremo come baseline questo classificatore, che chiameremo "dummy". Dovremo quindi provare che le prestazioni del nostro modello sono statisticamente migliori rispetto ad esso.

In questo capitolo, introdurremo alcune metriche per misurare le vere prestazioni del modello e una procedura per evitare che queste possano portarci a una valutazione distorta dello stesso.

4.1 Valutare le prestazioni del modello

Nell'esempio precedente abbiamo osservato come, in un dataset in cui una classe è predominante rispetto alle altre, l'uso dell'accuratezza come metrica nasconde le prestazioni

del modello. Pertanto, è necessario individuare una metrica che misuri le vere prestazioni del modello permettendo di confrontare diversi modelli nonostante lo sbilanciamento dei dati.

Per valutare le prestazioni del modello, si può utilizzare la matrice di confusione, ovvero una tabella che riassume le predizioni fatte dal modello, dove ogni riga rappresenta l'etichetta reale, mentre ogni colonna l'etichetta predetta dal modello. La matrice distingue gli esempi in quattro categorie: Veri Positivi (VP), Falsi Positivi (FP), Veri Negativi (VN) e Falsi Negativi (FN). Questa suddivisione aiuta a identificare gli eventuali errori del modello, permettendoci di comprendere in che modo esso viene "confuso".

Dai valori ottenuti dalla matrice possiamo definire le metriche precision e recall. La precision indica quanti esempi classificati positivi (negativi) sono realmente positivi (negativi). La recall indica quanti esempi sono classificati positivi (negativi) tra tutti quelli positivi (negativi). Le formule per calcolare queste metriche sono le seguenti:

$$\text{precision}(A) = \frac{VP}{VP + FP} \qquad \text{recall}(A) = \frac{VP}{VP + FN}$$

$$\text{precision}(B) = \frac{VN}{VN + FN} \qquad \text{recall}(B) = \frac{VN}{VN + FP}$$

$$\text{precision} = \frac{\text{precision}(A) + \text{precision}(B)}{2} \qquad \text{recall} = \frac{\text{recall}(A) + \text{recall}(B)}{2}$$

Dove A e B rappresentano le etichette di due classi distinte in un problema di classificazione binaria.

È utile avere un'unica metrica che combini precision e recall per confrontare due classificatori. Questa metrica, denominata F_β, è la media armonica tra precision e recall. La formula è la seguente:

$$F_\beta = \frac{(1 + \beta^2) \cdot \text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}}$$

Il parametro β determina il peso relativo tra recall e precision: $\beta > 1$ favorisce la recall, mentre $\beta < 1$ favorisce la precision.

Le tabelle 4.1 e 4.2, ci forniscono una valutazione più realistica delle prestazioni del classificatore "dummy".

	Normale (previsto)	Zombie (previsto)
Normale (reale)	9999 (VP)	0 (FN)
Zombie (reale)	1 (FP)	0 (TN)

Tabella 4.1: Matrice di confusione del classificatore "Dummy"

	Normale	Zombie	All
Precision	0.99	N.D	0.5
Recall	1	0	0.5
F_1			0.5

Tabella 4.2: Precisione, Recall e F_1 del classificatore "Dummy"

Nested-cross-validation. Nell'apprendimento supervisionato, può essere ingannevole valutare le prestazioni di un modello sui dati utilizzati per l'addestramento, poiché il modello si adatta specificatamente a questi esempi. Il vero indicatore delle prestazioni di un modello è la sua capacità di generalizzare su nuovi dati, non visti durante l'addestramento. È necessario pertanto valutare le prestazioni del modello su un set di dati separato, non utilizzato durante l'addestramento.

In modo analogo, se tentiamo di migliorare le prestazioni di un modello ottimizzando i suoi iperparametri basandoci su questo nuovo set di dati, incorriamo nel rischio di overfitting. In altre parole, anche se il modello potrebbe apparire molto efficace su questo set specifico, in realtà lo stiamo solo adattando per massimizzare le prestazioni su tali dati. Di conseguenza, i risultati ottenuti su questo nuovo set potrebbero nuovamente risultare fuorvianti.

Per evitare ciò, si utilizza il metodo di holdout, che prevede la divisione dei dati in in tre insiemi separati: uno per addestrare il modello (training set) un secondo per ottimizzare

gli iperparametri (validation set) e un terzo per valutare le prestazioni del modello (test set). Questo ci garantisce che le prestazioni del modello valutate sul test set non siano influenzate da un eccessivo adattamento ai dati.

Tuttavia, dato che tutti i dati provengono da una distribuzione sconosciuta, sorge una domanda: quando selezioniamo un sottoinsieme di questi dati per dividerli in seguito in tre insiemi distinti, come possiamo essere certi che le prestazioni del modello non siano soltanto il risultato della natura stocastica dei dati, e che, di conseguenza, le prestazioni sul set di test non siano frutto del caso? Pertanto, il metodo di holdout, che divide i dati in set di addestramento, validazione e test, è ancora sensibile alla selezione del set di test.

Un metodo più sofisticato per ottenere una buona stima delle prestazioni a regime di un modello è la nested-cross-validation [7]. Questa si basa sulla procedura iterativa della k-fold cross-validation, in cui il dataset viene diviso in k parti uguali, chiamate "folds". Per ogni iterazione del processo, un fold diverso viene utilizzato come set di test, mentre i restanti $k-1$ folds fungono da set di addestramento. Questo procedimento viene ripetuto k volte, con ogni fold utilizzato una volta come set di test.

La nested cross-validation consiste in due cicli di k-fold cross-validation annidati: il ciclo interno si occupa dell'ottimizzazione degli iperparametri e della selezione del modello migliore, mentre quello esterno valuta le prestazioni del modello selezionato (vedi figura 4.1). Calcolando la media delle ripetute misurazioni delle prestazioni del modello, si riduce la varianza nelle stime, rendendo così il risultato finale più affidabile e indicativo delle reali capacità del modello.

Uno svantaggio della nested-cross-validation è l'aumento significativo del numero di valutazioni del modello richieste. Pertanto, se il modello utilizzato all'interno di questa procedura ha tempi di addestramento prolungati o un gran numero di iperparametri da ottimizzare, questo metodo può diventare impraticabile.

4.2 Risultati ottenuti

Come menzionato nella sezione 2.2, rimuovere i job zombie può garantire il massimo payoff, a patto però che questi vengano identificati e rimossi poco dopo il loro inizio nel sistema. Se, per esempio, un job zombie viene individuato solo quando è vicino al

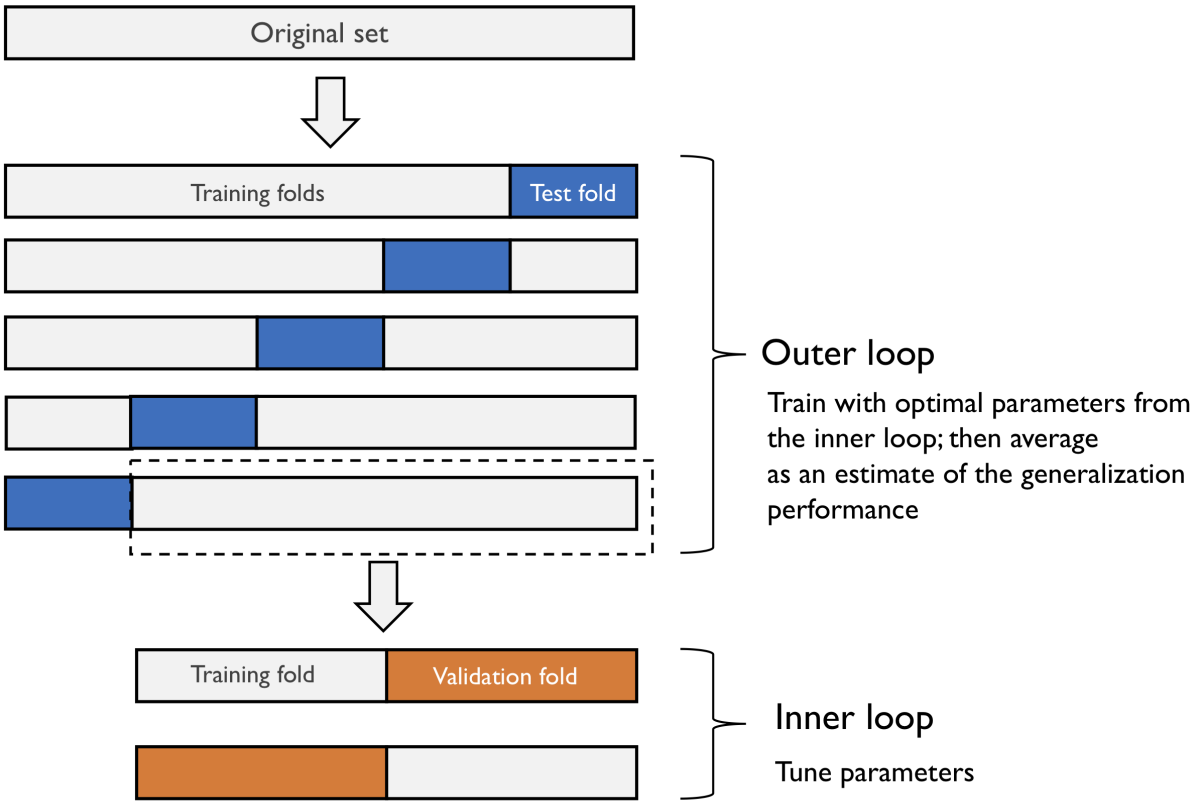

Figura 4.1: Nested cross-validation [30]

suo timeout, avrà già occupato risorse per un lungo periodo. Di conseguenza, il payoff ottenuto alla sua rimozione è notevolmente ridotto.

D'altra parte, se da un lato desideriamo rimuovere dei job zombie il prima possibile, dall'altro non vogliamo eliminare erroneamente quei job che stanno effettuando calcoli utili. All'inizio dell'esecuzione, disponiamo solo di pochi campionamenti da parte di HTCondor, il che limita la quantità di informazioni disponibili per il modello per fare delle predizioni. Idealmente, concedendo più tempo ai job prima di classificarli come zombie e accumulando un maggior numero di campionamenti, potremmo aumentare potenzialmente la precisione di un modello.

Le prossime due sezioni esploreranno queste due idee: la prima che mira a massimizzare il payoff, mentre la seconda estende il periodo di osservazione per incrementare la precisione del classificatore.

4.2.1 Prima ora

Nel primo approccio, si è optato per focalizzare l'applicazione di un modello di Machine Learning ai job monitorati nel mese di settembre 2021 appartenenti al gruppo di utenti ATLAS. Questa decisione è stata guidata sia dall'elevata presenza di job zombie in ATLAS rispetto ad altri gruppi (vedi figura 2.11), sia dai risultati dell'analisi visuale rappresentata nella figura 2.12, in cui erano stati distinti alcuni cluster di job zombie ben distinti.

Verrano considerati i dati relativi allo stato dei job nella loro prima ora di esecuzione, in modo da creare un modello che, dopo la prima ora, sia capace di classificare i job come zombie o normali.

La preparazione dei dati, con alcune specifiche modifiche rispetto alla procedura descritta nella sezione 3.2, è stata condotta come segue:

- Si è ridotta la frequenza di campionamento da 3 a 15 minuti per eliminare i valori ripetuti, portando così a array che contengono 4 valori anziché 20.
- I valori contenuti negli array sono stati trasformati in colonne. Questo passaggio elimina la cognizione temporale associata ai valori delle serie storiche.
- È stata introdotta una trasformazione polinomiale per le quattro colonne create da ogni serie storica. Per esempio, elevando al quadrato due colonne, si ottiene $(a, b)^2 = a^2 + 2ab + b^2$. Questo serve per generare nuove feature che esplicitano le interazioni tra le feature esistenti.
- Sono state aggiunte le feature `job work type` e `job type` nonostante non fossero necessarie nell'approccio in questione.
- Si è applicato l'one-hot encoding per le variabili categoriche, e la scelta del metodo di ridimensionamento delle feature è stata inclusa negli iperparametri da ottimizzare.
- Non è stato effettuato alcun bilanciamento dei dati.

Nella prima metà di settembre, è stata eseguita una nested cross-validation per valutare le prestazioni a regime e ricercare i migliori iperparametri, dettagliati nella tabella 4.3, della pipeline composta dal `Preprocessor` e dal modello XGBoost. Con i parametri

così definiti, il modello è stato riaddestrato su tutto il dataset compreso tra il 01 e il 15 settembre. Successivamente, il modello ottimizzato è stato testato sul dataset dal 16 al 30 settembre, con i risultati riportati nella tabella 4.4 e nella tabella 4.5.

Iperparametro	**Valore**
`preprocessor__num__scaler`	`StandardScaler()`
`preprocessor__num__poly__degree`	2
`xgb__min_child_weight`	6
`xgb__subsample`	0.65
`xgb__colsample_bytree`	0.8
`xgb__gamma`	0.2
`xgb__n_estimators`	300
`xgb__max_depth`	4
`xgb__learning_rate`	0.02
`xgb__reg_alpha`	0.4

Tabella 4.3: Iperparametri migliori del modello XGBoost sul dataset del periodo 01-15 settembre 2021

	Normale (previsto)	Zombie (previsto)
Normale (reale)	86719	272
Zombie (reale)	612	704

Tabella 4.4: Matrice di confusione del modello XGBoost sul dataset del periodo 16-30 settembre 2021

I risultati mostrano una buona capacità del modello nel distinguere tra job zombie e job normali, raggiungendo un F_1 score di 0.80 e una precisione del 72% per la classe meno rappresentata.

Si è poi utilizzato i risultati ottenuti da XGBoost e dal classificatore "Dummy", entrambi valutati con nested-cross-validation nello stesso periodo (vedi figura 4.6), per applicare

	Normale	Zombie	All
Precision	0.99	0.72	0.85
Recall	0.99	0.53	0.76
F_1			0.80

Tabella 4.5: Precisione, Recall e F_1 del modello XGBoost sul dataset del periodo 16-30 settembre 2021

il t-test[1] con una confidenza del 90%. Il p-value risultante è significativamente inferiore alla soglia del 10%, permettendoci di rigettare l'ipotesi nulla. Di conseguenza, possiamo affermare che le differenze osservate sono statisticamente significative, dimostrando che è possibile identificare i job zombie del gruppo ATLAS.

"Dummy"	**XGBoost**
F_1	F_1
0.494	0.792
0.494	0.802
0.494	0.786
0.494	0.772
0.494	0.769
t-value	**p-value**
-47.05	1.22e-06

Tabella 4.6: Confronto degli F_1 score del classificatore "dummy" e di XGBoost sui 5 fold esterni della nested-cross-validation sul dataset del periodo 01-15 settembre 2021

Infine, è possibile esaminare la conoscenza appresa dal modello, ottenendo una visione complessiva e sintetica del suo comportamento e permettendoci di capire quali feature sono importanti nelle decisioni che prende. La figura 4.2 mostra un grafico che rappre-

[1] Un test statistico che calcola un intervallo di confidenza per la differenza tra le medie di due gruppi. Se l'intervallo non include lo zero, questo indica una differenza significativa tra le medie.

senta la rilevanza di ciascuna feature nella costruzione di ogni albero decisionale. Tali rilevanze sono calcolate come la media delle importanze determinate da ciascun albero nell'ensemble.

Dall'analisi del grafico emergono alcuni spunti interessanti:

- Il primo campionamento della RAM e, come era previsto, le feature `job work type` e `job type` non hanno rilevanza.
- Il modello sembra beneficiare dall'interazione tra le diverse variabili.
- Vi è un'apparente gerarchia temporale nell'importanza delle feature, pur avendo rimosso ogni significato temporale nella fase di preparazione dei dati. Per esempio, il quarto campionamento della RAM appare più significativo del terzo, e così via, indicando una crescente rilevanza delle feature nel tempo.
- Le feature relative al disco sono più influenti, suggerendo che i job inizialmente potrebbero essere impegnati più in operazioni di download di file piuttosto che in calcoli.

Data la minore importanza delle misurazioni iniziali rispetto a quelle successive, si deduce che le informazioni iniziali sono meno informative. Questo ci suggerisce di considerare serie storiche di maggior lunghezza nel passo successivo, per arricchire le informazioni a disposizione del modello e incrementarne la precisione.

4.2.2 Prime 24 ore

Il passo successivo è stato estendere l'applicazione del modello di Machine Learning a tutti i gruppi di utenti, non solo ATLAS. Si è inoltre deciso di considerare le prime 24 ore per ottenere un maggior numero di campionamenti sullo stato dei job prima di effettuare una predizione. In aggiunta, si è optato per l'uso di dati più recenti, specificatamente quelli di marzo 2023, al fine di assicurarci che le nostre valutazioni effettuate riflettano le condizioni attuali del centro di calcolo.

Invece di trasformare i valori degli array in colonne, perdendo così la cognizione temporale, si è optato per mantenere l'aspetto temporale delle serie storiche. A tale scopo,

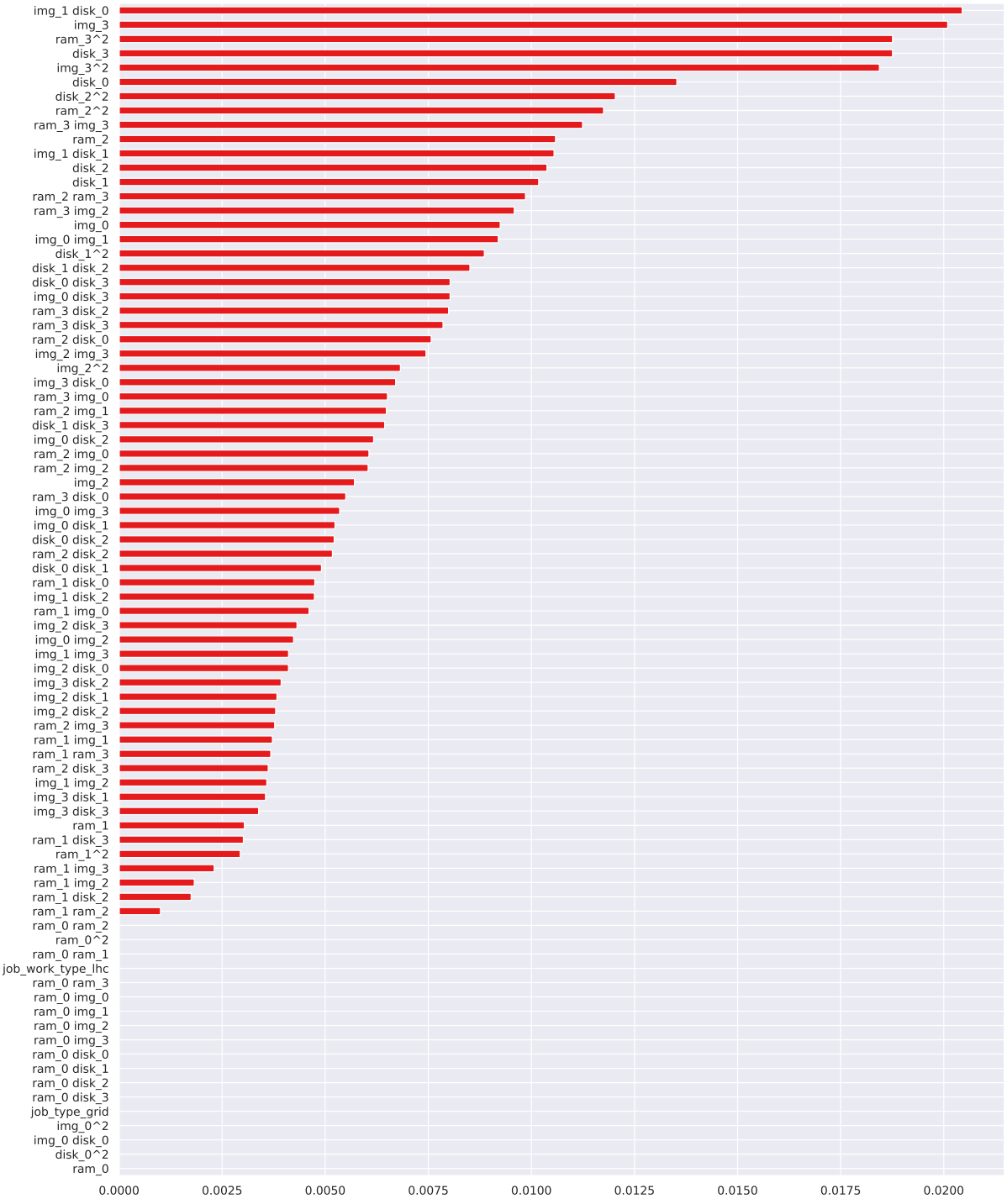

Figura 4.2: Rilevanza delle feature secondo il modello XGBoost sul dataset del periodo 01-15 settembre 2021

si è scelto di utilizzare reti neurali capaci di lavorare con tensori. In queste reti, l'input può essere un tensore 3D, strutturato in tuple del tipo (samples, time steps, features), che si allinea perfettamente alle dimensioni delle nostre multiple serie storiche multivariate. Questo approccio consente alle reti di catturare e apprendere anche le informazioni temporali inerenti a ciascun job.

Rispetto alla preparazione dei dati del primo approccio sono state apportate le seguenti modifiche:

- Per gestire serie storiche di lunghezza variabile, gli array sono stati uniformati tramite zero-padding e troncamento, assicurando che ogni array abbia esattamente 96 elementi.
- I valori negli array di 96 elementi sono stati prima convertiti in righe di una matrice. Successivamente, questa matrice è stata trasformata in un tensore 3D con dimensioni (job, time step, feature).
- Per bilanciare i dati, è stato rimosso il 90% dei job normali. In aggiunta, sono state introdotte varianti di job zombie per raggiungere una proporzione di 1:10. Questa proporzione è stata poi utilizzata come costi da assegnare ai falsi positivi e ai falsi negativi nel modello.

Le reti neurali delle tipologie descritte nella sezione 3.3.2 sono state implementate (per i dettagli implementativi, si rimanda il lettore al repository[2]) e sperimentate:

- convoluzionali: FCN e ResNet [26, 21]
- ricorrenti: LSTM [22]
- transformer: Transformer encoder [36]

Poiché i tempi di addestramento con le reti neurali sono significativamente maggiori, è stato necessario utilizzare il metodo holdout, rispetto alla nested-cross-validation, per valutare le prestazioni del modello. Del dataset, si è destinato un terzo dei dati al test set e, dai dati rimanenti, un quinto è stato selezionato come validation set.

[2] https://github.com/alessioarcara/JobFailurePrediction-CNAF

Le LSTM e l'encoder del Transformer si sono subito distinti per prestazioni superiori rispetto alle controparti convoluzionali. Dato che i tempi di addestramento delle LSTM risultavano lunghi, si è preferito concentrarsi sull'encoder del transformer, che grazie alla sua parallelizzazione intrinseca, ha garantito tempi di addestramento minori e permesso di esplorare diverse configurazioni con più facilità.

Va notato inoltre che il sovracampionamento, utilizzato per ridurre lo sbilanciamento, si è rivelato controproducente. Infatti, i modelli si adattavano a predire dei job zombie del 2021, che non erano più rappresentativi dei job zombie presenti nel sistema nel 2023. Pertanto, si è rimosso questo componente.

Le tabelle 4.7 e 4.8 presentano i risultati ottenuti configurando l'encoder del Transformer con gli iperparametri elencati nella tabella 4.9. Questa configurazione ha portato a un F_1 score di 0.73, con una precisione del 42% e una recall del 55%. Questo indica che, sebbene il modello sia in grado di identificare i job zombie, questo è poco preciso.

	Normale (previsto)	Zombie (previsto)
Normale (reale)	20860	92
Zombie (reale)	53	67

Tabella 4.7: Matrice di confusione del modello Transformer sul set di test del dataset del periodo 13-31 marzo 2023

	Normale	Zombie	All
Precision	0.99	0.42	0.70
Recall	0.99	0.55	0.77
F_1			0.73

Tabella 4.8: Precisione, Recall e F_1 del modello Transformer sul set di test del dataset del periodo 13-31 marzo 2023

In seguito, si sono confrontate le prestazioni dell'encoder del Transformer con quelle di XGBoost, utilizzando gli stessi dati. La preparazione è rimasta invariata, ad eccezione

Iperparametro	Valore
preprocessor__num__scaler	MinMaxScaler()
head_size	32
num_heads	4
ff_dim	64
num_transformer_blocks	1
mlp_units	96
mlp_dropout	0.5
dropout	0.5
loss	binary_crossentropy
optimizer	Adam(learning_rate=1e-4)

Tabella 4.9: Iperparametri del modello Transformer sul set di addestramento del dataset periodo 13-31 marzo 2023

della trasformazione delle serie storiche in colonne. Le tabelle 4.10 e 4.11 confermano i risultati precedentemente osservati per campionamenti fino a un'ora, evidenziando una maggiore precisione di XGBoost, del 57%, rispetto alla controparte, sebbene a discapito della recall. Vale la pena notare, però, che esiste la possibilità di aumentare la precision di un classificatore sacrificando un po' della recall. Questo si ottiene impostando un valore soglia sulle probabilità di previsione del classificatore. Aumentando questa soglia, il classificatore diventerà più sicuro nelle sue previsioni, riducendo così la frequenza dei falsi positivi [15].

	Normale (previsto)	Zombie (previsto)
Normale (reale)	20860	38
Zombie (reale)	69	51

Tabella 4.10: Matrice di confusione del modello XGBoost sul set di test del dataset del periodo 13-31 marzo 2023

	Normale	Zombie	All
Precision	0.99	0.57	0.78
Recall	0.99	0.42	0.71
F_1			0.74

Tabella 4.11: Precisione, Recall e F_1 del modello XGBoost sul set di test del dataset del periodo 13-31 marzo 2023

Nel caso di XGBoost, è possibile però impiegare la nested-cross-validation per confrontare i suoi risultati con quelli di un classificatore "Dummy". Applicando nuovamente il t-test con una confidenza del 90% e guardando i risultati illustrati nella tabella 4.12, possiamo respingere l'ipotesi nulla a causa del p-value, notevolmente inferiore alla soglia del 10%, permettendoci di affermare che la differenza tra i due modelli è significativa. Pertanto, possiamo concludere che è fattibile identificare i job zombie.

"Dummy"	**XGBoost**
F_1	F_1
0.494	0.713
0.494	0.743
0.494	0.739
0.494	0.737
0.494	0.754
t-value	**p-value**
-36.16	3.49e-06

Tabella 4.12: Confronto degli F_1 score del classificatore "dummy" e di XGBoost sui 5 fold esterni della nested-cross-validation sul set di addestramento e validation del dataset del periodo 13-31 marzo 2023

In conclusione, nonostante l'incremento nel numero di campionamenti, non si è ottenuto il miglioramento desiderato di precisione nei modelli. Questo calo di precisione

potrebbe però essere dovuto all'aver considerato non più un singolo gruppo di utenti "ideale", come ATLAS, ma più gruppi di utenti. Questi ultimi conducono esperimenti diversi e potrebbero avere job con caratteristiche differenti. Di conseguenza, i risultati ottenuti in questo passo sono più generalizzabili rispetto a quelli del primo passo.

Prima di passare alle conclusioni, presenteremo un approccio che è stato sperimentato, ma che non ha prodotto i risultati attesi, rivelando tuttavia un limite nel nostro caso di studio.

Non stazionarietà. Come discusso nella sezione 3.3, in questa modellazione del problema, i job zombie sono considerati come delle "novità", cioè come qualcosa di nuovo e non simile a ciò che il modello ha visto in precedenza.

Si è proceduto con il gruppo di utenti ATLAS, nel periodo di settembre 2021, per valutare l'efficacia di questo approccio, iniziando prima con un caso più semplice. La preparazione dei dati è rimasta invariata a quella descritta precedentemente, a settembre 2021. I job zombie sono stati rimossi dal set di addestramento per permettere all'autoencoder di apprendere le caratteristiche dei job normali e di formare potenzialmente dei cluster nello spazio latente di job "già visti".

In seguito, è stato addestrato un autoencoder applicando i diversi vincoli, descritti nella sezione 3.3.3, e utilizzando diverse configurazioni di iperparametri, quali il numero di neuroni, il numero di strati e la tipologia degli strati. Come si può osservare nella figura 4.3, nessuna di queste configurazioni ha portato a una rappresentazione soddisfacente nel periodo di test del 16-30 settembre, sufficiente per separare i job zombie dai job normali tramite un valore soglia.

Il motivo di ciò potrebbe essere attribuito a molteplici fattori, alcuni dei quali sono già stati discussi nella sezione 2.2. In più, in questa modellazione, vi è l'ipotesi implicita che le caratteristiche dei job "già visti" rimangono costanti nel tempo. Tuttavia, se il sistema è non stazionario a causa dell'arrivo di nuovi esperimenti da parte dei gruppi di utenti, i quali mutano le caratteristiche dei job, ciò potrebbe portare il modello a modellare come novità qualcosa che, in realtà, non lo è.

Su questa base, si sono volute valutare le prestazioni del modello XGBoost addestrato nel periodo dal 13 al 31 marzo, per osservare come esso si comporta sui dati del mese suc-

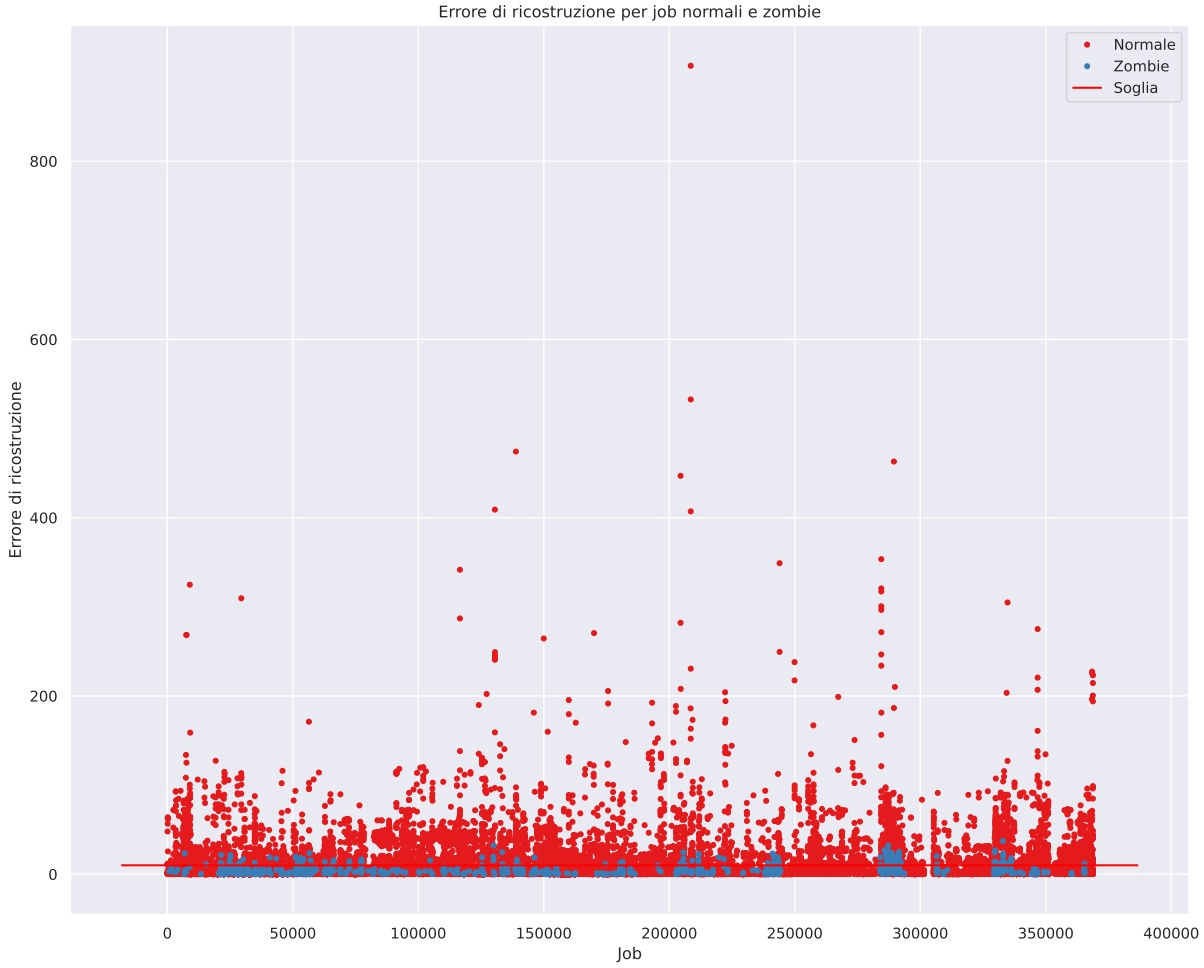

Figura 4.3: Errore di ricostruzione sul dataset del periodo 16-30 settembre 2021

cessivo. Nella tabella 4.13 si nota un crollo delle prestazioni del modello. Questo conferma che le caratteristiche dei job cambiano nel tempo e che la conoscenza del modello degrada, rendendola meno rappresentativa della realtà attuale. Tuttavia, questo non rappresenta un problema, in quanto, se i dati cambiano, anche i modelli possono cambiare. Questo si può ottenere riaddestrando periodicamente il modello, il che può essere automatizzato all'interno della pipeline. Integrando un componente che monitora le performance del modello e scatena un evento di riaddestramento quando le prestazioni scendono sotto una certa soglia arbitraria (come illustrato in figura 4.4), si possono mantenere le prestazioni del modello nel tempo.

	Normale (previsto)	Zombie (previsto)
Normale (reale)	253687	295
Zombie (reale)	31	9

Tabella 4.13: Matrice di confusione del modello XGBoost sul dataset del periodo 21-28 aprile 2023

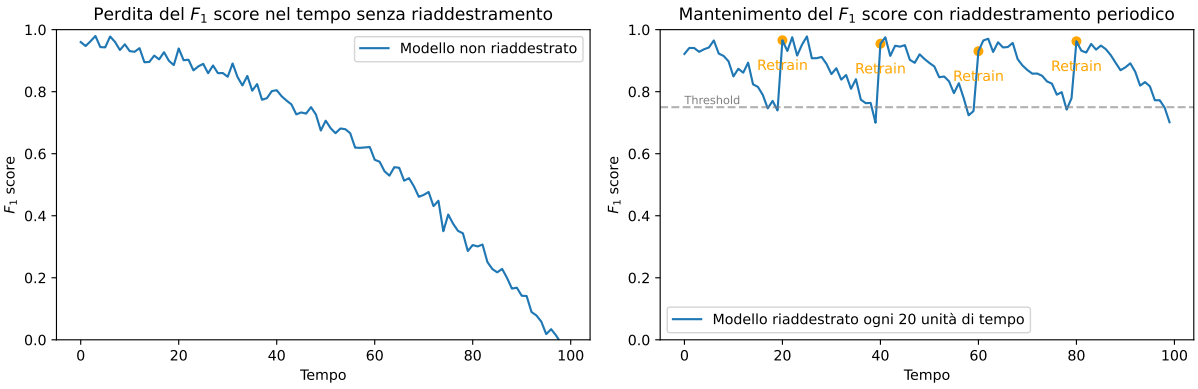

Figura 4.4: Processo di riaddestramento periodico del modello

Capitolo 5

Conclusioni e sviluppi futuri

Questo studio ha esplorato il problema di prevedere il successo o il fallimento dei job in una farm di calcolo mediante l'uso di tecniche di Machine Learning e Deep Learning. Dalle analisi effettuate nel capitolo 2, è stato identificato un particolare sottoinsieme di job che falliscono, denominati zombie. Questi, benché smettano di effettuare calcoli, non rilasciano l'host fisico, occupando improduttivamente delle risorse fino al loro timeout. L'obiettivo della tesi è stato quello di individuare questi job il più presto possibile, poiché identificarli nelle loro fasi iniziali risulta essere particolarmente vantaggioso in termini di risparmio di risorse derivante dalla loro rimozione. Dai risultati ottenuti nel capitolo 4, addestrando un apposito modello predittivo, è possibile affermare che i job zombie sono identificabili. Sono stati proposti e validati, su tutti i gruppi di utenti e con dati recenti (vedi figura 5.1), due modelli capaci di identificare i job che, con buona probabilità, diventeranno zombie (1 su 2). Le predizioni fornite dal modello possono essere utilizzate per impostare un filtro o un avviso, permettendo così di controllare manualmente i job sospetti o di stabilire una regola per la loro eliminazione.

Il lavoro può essere proseguito per sviluppi futuri, partendo dai notebook esistenti disponibili all'indirizzo https://github.com/alessioarcara/JobFailurePrediction-CNAF.

Il passo successivo per una ricerca potenziale consiste nell'integrare, anziché scegliere uno dei due classificatori, entrambi in un ensemble. Combinando le previsioni di diversi classificatori e applicando una regola di voto, è possibile compensare sorprendentemente

Metrica	Transformer	XGBoost
Precisione (classe minoritaria)	0.42	0.57
Recall (classe minoritaria)	0.55	0.42
F_1	0.73	0.74

Tabella 5.1: Risultati dei modelli Transformer e XGBoost nell'identificazione dei job zombie sul set di test del dataset del periodo 13-31 marzo 2023

le rispettive debolezze, ottenendo prestazioni migliorate [15]. Si consiglia, dunque, di esplorare ulteriori modelli e di integrarli in un ensemble.

Inoltre, l'incremento del numero di campionamenti da un'ora a 24 ore non ha portato a un corrispondente incremento nella precisione. Tuttavia, questo potrebbe essere dovuto al fatto di aver considerato più di un gruppo di utenti. Un'altra ricerca potenziale che potrebbe migliorare le prestazioni è, invece di considerare tutti i job di tutti i gruppi di utenti per addestrare un unico modello, addestrare un modello per ogni gruppo di utenti. In questo modo, si otterrebbe alla fine un array di modelli, che potrebbe mantenere le stesse prestazioni osservate con il gruppo di utenti ATLAS.

Un altro aspetto importante è la qualità dei dati. Infatti, la qualità e la quantità dei dati sono più importanti del modello utilizzato [18]. È utile arricchire le feature con dati provenienti da altre fonti. Considerando che il CNAF monitora non solo lo stato dei job, ma anche quello degli host fisici dove i job sono attualmente in esecuzione, con un campionamento ogni 3 minuti, un'ulteriore direzione di ricerca potrebbe essere quella di incrociare questi dati. In questo modo, si potrebbe ottenere informazioni anche sull'ambiente in cui i job sono in esecuzione.

Sempre in merito ai dati, un'altra ricerca futura potrebbe concentrarsi sul miglioramento della preparazione dei dati. Questo potrebbe includere l'aggiunta di metodi alla classe `Preprocessor` per pulire le serie storiche, ad esempio rimuovendo gli outlier o applicando una media mobile per ridurre la variabilità. Inoltre, si potrebbero apportare miglioramenti anche nell'etichettatura dei dati. Ad esempio, si potrebbe considerare come job zombie non solo quelli che terminano a causa di un timeout di 7 giorni, ma anche quelli in cui il `cputime` rimane invariato per un lungo periodo. Ciò garantirebbe di identi-

ficare con maggiore sicurezza i job zombie, evitando di classificare erroneamente quei job che richiedono più tempo per completare i calcoli.

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. 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. 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. URL: https://superfri.org/index.php/superfri/article/view/14.

[7] Gavin Cawley e Nicola Talbot. «On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation». In: *Journal of Machine Learning Research* 11 (lug. 2010), pp. 2079–2107.

[8] CERN. *Worldwide LHC Computing Grid*. 2023. URL: https://wlcg.web.cern.ch (visitato il 28/10/2023).

[9] 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. URL: http://dx.doi.org/10.1145/2939672.2939785.

[10] CNAF. *WLCG Tier-1 data center - Calcolo*. URL: https://www.cnaf.infn.it/calcolo/ (visitato il 28/10/2023).

[11] 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.

[12] 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.

[13] 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. URL: https://doi.org/10.1214/aos/1013203451.

[14] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994, pp. 325–330.

[15] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2ª ed. O'Reilly Media, 2019.

[16] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[17] Rui Guo et al. «Degradation state recognition of piston pump based on ICEEMDAN and XGBoost». In: *Applied Sciences* 10 (set. 2020), p. 6593. DOI: 10.3390/app10186593.

[18] Alon Halevy, Peter Norvig e Fernando Pereira. «The Unreasonable Effectiveness of Data». In: *IEEE Intelligent Systems* 24.2 (2009), pp. 8–12. DOI: 10.1109/MIS.2009.36.

[19] 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.

[20] H. He e Y. Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*. Wiley, 2013. ISBN: 9781118646335. URL: https://books.google.it/books?id=CVHx-Gp9jzUC.

[21] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[22] Sepp Hochreiter e Jürgen Schmidhuber. «Long Short-term Memory». In: *Neural computation* 9 (dic. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[23] Diederik P Kingma e Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML].

[24] 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.

[25] Yann A. LeCun et al. «Efficient BackProp». In: *Neural Networks: Tricks of the Trade: Second Edition*. A cura di Grégoire Montavon, Geneviève B. Orr e Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

[26] Jonathan Long, Evan Shelhamer e Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. 2015. arXiv: 1411.4038 [cs.CV].

[27] 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.

[28] 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.

[29] 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.

[30] Sebastian Raschka. *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning - Part IV*. 2018. URL: https://sebastianraschka.com/blog/2018/model-evaluation-selection-part4.html (visitato il 24/11/2023).

[31] Andrea Rendina. *INFN-T1 site report*. https://indico.cern.ch/event/1200682/contributions/5087586/attachments/2538178/4368754/20221031_InfnT1_site_report.pdf. Visitato il 28/10/2023. 2022.

[32] 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).

[33] 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.

[34] Ravid Shwartz-Ziv e Amitai Armon. «Tabular Data: Deep Learning is Not All You Need». In: *CoRR* abs/2106.03253 (2021). arXiv: 2106.03253. URL: https://arxiv.org/abs/2106.03253.

[35] 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.

[36] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[37] 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.

[38] Martin Wattenberg, Fernanda Viégas e Ian Johnson. «How to Use t-SNE Effectively». In: *Distill* (2016). DOI: 10.23915/distill.00002. URL: http://distill.pub/2016/misread-tsne.

Ringraziamenti

Ringrazio il Prof. Moreno Marzolla e il Dott. Stefano Dal Pra, per l'opportunità datami, per i consigli costruttivi e per il supporto offerto in tutte le fasi di svolgimento del lavoro.

Ringrazio Rita, il 6° Reparto Infrastrutture, i miei zii, Antonietta, Yonne e tutti gli amici conosciuti ai Salesiani per il loro sostegno e per non avermi fatto sentire solo in un periodo della mia vita in cui mi sono trovato da solo.

Ringrazio i colleghi ed amici della "prima fila" per il bel percorso universitario insieme.

Ringrazio la Fortuna che, in mezzo a molte situazioni non semplici, mi ha fatto incontrare Alessia.