Università Politecnica Delle Marche

Dipartimento di Ingegneria dell'Informazione

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Analisi e Benchmarking di algoritmi per lo Streaming Machine Learning e tecnologie per l'edge deployment

Analysis and Benchmarking of algorithms for Streaming Machine Learning and technologies for edge deployment

Relatore:

Prof. Adriano Mancini

Correlatore:

Dott. Mauro Mariniello

Candidato: 1102533

Lorenzo D'Agostino

Anno Accademico 2021 / 2022

A Mamma e Papà, grazie per aver reso possibile il raggiungimento di questo traguardo. Ai miei fratelli, per il vostro costante supporto.
A Raffaella, grazie per esserci sempre.

Ringrazio il Prof. Adriano Mancini per la sua competenza e professionalità, che sono e sempre saranno di ispirazione al mio lavoro e crescita professionale.

Grazie a Mauro Mariniello, Alessandro Conflitti e Paolo Filippelli per i preziosi consigli e per il costante supporto datomi nella stesura di questo lavoro di tesi.

Introduzione

Con la sempre crescente capacità computazionale dei dispositivi odierni, con l'aumento dei dispositivi *IoT* (*Internet of Things*), con la continua evoluzione dei settori di applicazione dell'intelligenza artificiale, l'*Edge AI* (*Edge Artificial Intelligence*) è una tecnologia di intelligenza artificiale che sta guadagnando sempre più piede nel panorama digitale odierno. Questo è certamente dovuto alla disponibilità di hardware in grado di reggere il peso computazionale di algoritmi di intelligenza artificiale, ma le motivazioni principali per cui questa tecnologia è sempre più utilizzata trovano le loro radici più profonde nelle tematiche riguardanti la privacy e sicurezza dei dati, che altrimenti dovrebbero essere mandati in un centro di calcolo centralizzato ubicato in un posto lontano dalla fonte generatrice del dato stesso, oltre che la necessità di applicazioni che elaborino i dati in tempo reale, come ad esempio la robotica, l'IoT, le applicazioni in realtà aumentata, e molte altre.

Tuttavia, la stragrande maggioranza delle applicazioni odierne si basa su tecnologie cloud. Risulta quindi necessaria una metodologia per la comunicazione tra dispositivi edge ed i cluster computazionali, comunicazione che molto spesso può venir meno a causa delle possibili interruzioni di rete cui i dispositivi sono sottoposti. Questo può rappresentare un problema, in quanto le applicazioni edge potrebbero interrompersi in caso di mancanza di connettività. Inoltre, la manutenzione ed il continuo aggiornamento del software all'edge risulta necessario al fine di evitare l'obsolescenza dei dispositivi.

L'obiettivo di questo lavoro di tesi è quello di effettuare un'analisi delle tecnologie utili all'implementazione di un software di Edge AI. Nello specifico, data

la natura dei dati edge e la sempre crescente numerosità di dati generati da sensori, si è approfondita una tecnologia emergente per quanto riguarda il campo dell'intelligenza artificiale ovvero lo *streaming machine learning*.

Nei capitoli che seguono viene presentato il lavoro svolto durante l'attività di tirocinio svolta presso l'azienda **Radicalbit**[1]. Vengono quindi definite le problematiche inerenti i paradigmi classici di machine learning, per poi passare ad un approfondimento delle tecnologie per lo streaming machine learning. Successivamente verranno approfondite le tecnologie per il deployment di applicazioni in architetture cluster con un focus in quelle specializzate sul tema dell'edge deployment. Infine viene effettuato lo sviluppo di una applicazione di Edge AI basata su un caso d'uso reale, utilizzando le tecnologie precedentemente descritte, con una attenzione particolare ai risultati ottenuti dai modelli di intelligenza artificiale nell'architettura classica ed edge.

[1] https://radicalbit.io/

Indice

Capitolo 1

Batch Machine Learning

In questo capitolo viene dato un quadro generale sul machine learning e sulle metodologie per l'apprendimento automatico, con particolare attenzione alle tecniche ed ai paradigmi correlati con il lavoro svolto in questa tesi.

1.1 Panoramica generale

L'apprendimento automatico, o **machine learning**, può essere descritto come la disciplina che cerca di estrarre automaticamente la conoscenza dai dati. Anche se è passato un po' di tempo dall'origine di questo campo (i primi studi risalgono ai primi anni Cinquanta [TURING, 1950] ed il termine apprendimento automatico viene coniato alla fine dello stesso decennio [Samuel, 1959]), ha visto una massiccia crescita sia in ambito industriale che accademico nell'ultimo decennio. Questo fenomeno può essere attribuito alla crescente quantità di dati oggi disponibili, ed in continua crescita, oltre che alle maggiori prestazioni computazionali dei computer moderni, che consentono di svolgere carichi di lavoro che in passato sarebbero stati proibitivi.

Nel dettaglio, il machine learning è quella branca dell'informatica che esplora lo studio e l'implementazione di algoritmi in grado di apprendere informazioni a partire dai dati che si hanno a disposizione, e forniscono la capacità di predire nuove informazioni, alla luce di quelle apprese, attraverso l'addestramento e l'ottimizzazione di modelli più o meno complessi.

Le applicazioni del machine learning possono essere molteplici e riguardare diversi ambiti. Si pensi ai sistemi di domotica in grado di regolare la temperatura, l'umidità o l'illuminazione in base alle nostre abitudini, all'utilizzo della nostra voce come input per alcuni dispositivi, ai filtri anti spam nelle e-mail ed i motori di ricerca, per citarne alcuni.

Il machine learning presenta profondi legami con il campo dell'**ottimizzazione matematica**, il quale fornisce teorie, metodi e domini di applicazione. Molti problemi di apprendimento automatico, infatti, vengono formulati come problemi di **minimizzazione** di una determinata funzione di obiettivo, chiamata *loss function*, nei confronti di un determinato set di dati contenente gli esempi, detto *training set*. Questa funzione esprime la discrepanza tra i valori predetti dal modello in fase di allenamento ed i valori attesi per ciascuna istanza di esempio. L'obiettivo finale è dunque quello di insegnare al modello a predire correttamente i valori attesi su un set di istanze non presenti nel set di train, il *test set*, mediante la minimizzazione della loss function in questo insieme di istanze. Questo porta portando ad una maggiore capacità di generalizzazione del modello, per cui a previsioni più accurate su dati mai visti prima.

1.2 Tassonomia del machine learning

Le diverse categorie del machine learning sono tipicamente riconducibili a tre ampie categorie caratterizzate dal tipo di feedback su cui si basa il sistema di apprendimento:

- apprendimento *supervisionato*;
- apprendimento *non supervisionato*;
- apprendimento *con rinforzo*.

Un altro metro di giudizio secondo il quale è possibile distinguere le diverse categorie di task è il tipo di output atteso da un certo sistema di machine learning che può essere *continuo* o *discreto*. In questa sezione vengono approfondite le tipologie di algoritmi di machine learning collocandoli nelle famiglie sopra citate.

Apprendimento supervisionato

Con l'apprendimento supervisionato i modelli vengono addestrati a ricondurre i dati in input a specifiche classi predefinite, apprendendo una regola generale (funzione matematica) in grado di mappare gli input negli output. I modelli appartenenti a questa famiglia di algoritmi sono caratterizzati dalla necessità di data set di addestramento contenenti **dati etichettati**, ovvero dati per cui sono conosciuti gli output attesi.
In questa famiglia di algoritmi possiamo collocare quelli che rispondono al problema di **classificazione** e **regressione**:

- **Classificazione:**

  algoritmi di questo tipo sono in grado di prevedere un risultato espresso come un valore discreto che indica l'appartenenza di un oggetto ad una classe specifica (figura 1.1). Alcuni esempi di task risolubili da questo tipo di algoritmi sono la tassonomia, ovvero la classificazione nel regno animale di ogni essere vivente, la sentiment analysis, il cui obiettivo è classificare un commento testuale come positivo o negativo, o la classificazione di istanze anomale nelle letture di un sensore (tema trattato in questo lavoro di tesi). Inoltre è possibile suddividere ulteriormente questo task in task di classificazione **binaria** e **multiclasse** in base al numero di classi disponibili all'interno del train set.

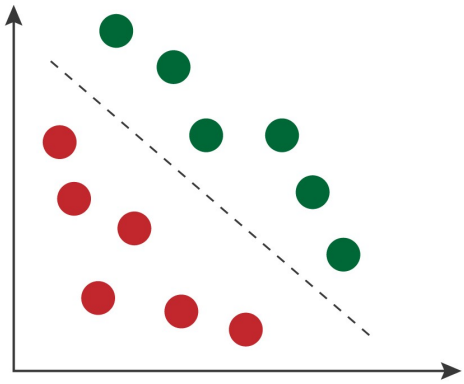


Figura 1.1: Task di classificazione

- **Regressione:**

  Il termine *regressione* è stato utilizzato per la prima volta dal biologo inglese Galton (fine Ottocento [Galton, 1886]) durante il suo studio delle relazioni nelle stature di genitori e figli. In questo studio il biologo notò che l'altezza dei figli si spostava (**regrediva**) verso la media, anche se i genitori erano caratterizzati da stature più estreme (molto alti/bassi). Da qui il concetto alla base della regressione statistica: lo studio della regressione verso la media. Algoritmi di questo tipo si distinguono dai precedenti perché i risultati vengono espressi come valori continui. L'idea di base è quella di trovare un insieme di pesi che siano in grado di predire correttamente l'output desiderato dato un vettore dei dati in input (figura 1.2).

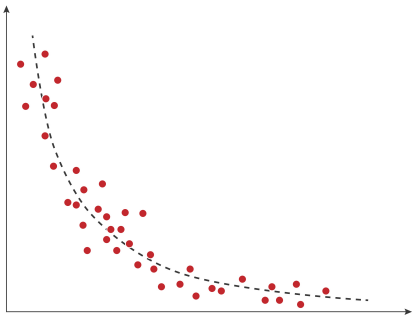


Figura 1.2: Task di regressione

Questa famiglia di modelli comprende algoritmi molto efficienti; di contro però hanno un grande limite ovvero non sono sempre applicabili ad ogni ambito in quanto il requisito fondamentale è la presenza di un training set etichettato, e spesso non è disponibile.

Apprendimento non supervisionato

Algoritmi di questo tipo vengono utilizzati quando il training set a disposizione non è provvisto di etichette o, più in generale, di un corrispondente valore di output. In questo caso lo scopo degli algoritmi appartenenti a questa famiglia hanno come obiettivo quello di cercare pattern nascosti nei dati che sfuggono all'osservazione, sia perché oscurati da altre informazioni, sia perché la quantità di dati è talmente grande da non poter essere osservata facilmente.
Il risultato dell'applicazione di algoritmi non supervisionati è una ripartizione del dataset di input in sottoinsiemi che hanno caratteristiche comuni o una relazione di similitudine (figura 1.3).

Questa famiglia di algoritmi è in grado di risolvere principalmente due tipi di task:

- **Clustering**:

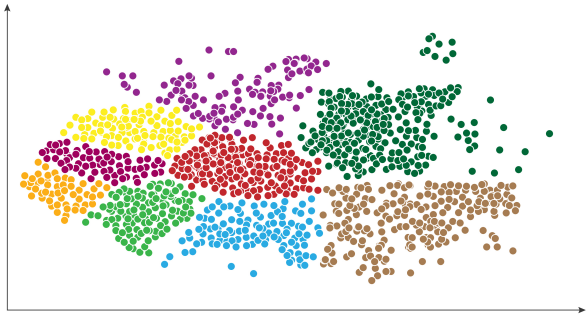


Figura 1.3: Apprendimento non supervisionato

permette di aggregare i dati in gruppi (detti **cluster**) che hanno come caratteristiche il fatto di essere ognuno omogeneo al proprio interno (secondo il concetto di similarità o distanza) ed essere invece tra loro disgiunti, quindi più possibile diversi e distanti tra loro. Questo si traduce in un modello che riconduce ogni dato ad uno ed un solo cluster che comprende tutti e soli dati simili a lui, al pari di ciò che accade nei modelli supervisionati di classificazione, con la differenza che non si conosce a priori la definizione del gruppo.

I settori di applicazione di questi algoritmi sono dei più disparati, e vanno dalla biologia, in cui vengono utilizzati per raggruppare la composizione degli amino-acidi di proteine e geni al fine di illustrare la sequenza evolutiva delle mutazioni da cui hanno avuto origine nuove specie, all'economia e marketing nelle ricerche di mercato e segmentazione dei target.

- **Analisi associative**:

  lo scopo delle analisi associative è quello di estrarre pattern frequenti dai dati tali per cui sia possibile dedurre le relazioni che li accomunano. La correlazione tra variabili e le **regole associative** fanno parte di questa famiglia. Un esempio di regole associative è la ricerca di regolarità tra i prodotti venduti in un supermercato, quindi la ricerca delle implicazioni nascoste,

al fine di indirizzare sconti e campagne, ma anche per definire la posizione dei prodotti sugli scaffali. Mentre potrebbe risultare più scontata una relazione tra l'acquisto di pane e marmellata, risulta assai meno scontata la relazione presentata da *Mark Madsen* nella sua presentazione intitolata *"Beer, diapers and correlation: A Tale of Ambiguity"* (Birra, pannolini e correlazione: una storia di ambiguità) in cui afferma che gli uomini intenti ad acquistare pannolini in un supermercato, sono molto propensi ad acquistare birra. La storia, racconta Madsen, nasce nel 1992 dalla rivista Chain Store Age, ma trovò effettivamente un'alta correlazione tra le vendite dei due prodotti solo nel 1997 effettuando analisi su dati provenienti da negozi di alimentari, diventando così l'esempio più utilizzato nell'illustrazione di questa famiglia di algoritmi.

Le analisi non supervisionate hanno il grande vantaggio di non aver bisogno di dataset etichettati per poter creare il modello; d'altra parte per lo stesso motivo possono risultare meno accurate. Per ovviare a questo sono state introdotte delle modalità ibride, che cercano di prendere sia i vantaggi di quelle supervisionate (alta accuratezza), sia delle non supervisionate (applicabili ad un maggior numero di dataset). Purtroppo gli algoritmi semi-supervisionati prendono anche parte degli svantaggi dalle due soluzioni, per cui soffrono di limitazioni che li rendono difficilmente applicabili.

Apprendimento con rinforzo

A differenza dei metodi di apprendimento presentati nelle sezioni precedenti, questo paradigma si rivolge verso problemi in cui la previsione da produrre non si basa soltanto sulle caratteristiche dei dati, ma dipende dallo stato attuale del sistema. L'idea su cui si basano questi algoritmi è guidata dall'introduzione nel sistema di apprendimento di **ricompense** (o punizioni) modellati come valori numerici indice della qualità della decisione presa, al fine di incoraggiare il modello ad effettuare le decisioni ritenute migliori. Anche questa famiglia di algoritmi presenta numerosi campi di applicazione che vanno dalla robotica alla genetica.

Un esempio di applicazione reale di questi algoritmi è quello mostrato da *Jim Gao* [Gao, 2014] in cui afferma ti aver ridotto, circa del 40%, i consumi energetici degli impianti di climatizzazione dei data center di Google utilizzando un algoritmo appartenente a questa famiglia.

1.3 Batch learning supervisionato

Come descritto nella sezione 1.2, nell'apprendimento batch supervisionato il compito dell'algoritmo di apprendimento è quello di prendere in input un insieme di dati, composto da coppie di punti dati ed etichette, restituendo in output un predittore (modello) in grado di dedurre le etichette per nuovi dati.

Al fine di analizzare un algoritmo di apprendimento, è necessario un modello matematico formale che descriva rigorosamente le relazioni ingresso uscita. Seguendo l'impostazione classica dell'apprendimento statistico [Bousquet et al., 2004], un *campione* è una coppia $(\mathbf{x}, y)$, dove $\mathbf{x} \in \mathcal{X}$ e $y \in \mathcal{Y}$ sono rispettivamente il vettore dei dati in input (*attributi*) e l'etichetta corrispondente. Andando più nel dettaglio $\mathcal{X}$, detto *domain set* (insieme di dominio), è un insieme arbitrario contenente i punti da etichettare, i cui elementi vengono solitamente rappresentati come vettori *n-dimensionali* di numeri, reali o interi, dove n rappresenta quindi il numero di attributi descrittivi dei dati. $\mathcal{Y}$ invece, detto *label set*, è l'insieme delle etichette, le cui caratteristiche possono variare in base al problema che si sta affrontando (per task di regressione sarà composto da elementi continui, mentre per task di classificazione i suoi elementi saranno discreti). L'insieme dei dati S che gli algoritmi usano per addestrare il predittore è un insieme finito di coppie di attributi ed etichette.

Dato quindi $S_m = \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m)\}$, con $(\mathbf{x_i}, y_i) \in \mathcal{X} \times \mathcal{Y}$, il train set composto da m campioni, ed A un dato algoritmo di machine learning, il risultato della esecuzione di A su S_m è un predittore h ($A(S_m) = h$), o *modello*, riconducibile ad una funzione matematica $h : \mathcal{X} \to \mathcal{Y}$ che può essere utilizzata per dedurre l'etichetta di un nuovo elemento di $\mathcal{X}$.

A questo punto è possibile definire un modello probabilistico assumendo che

ogni coppia $(\mathbf{x_i}, y_i)$ sia proveniente da un'estrazione casuale in una distribuzione di probabilità congiunta $\mathcal{P}$ su $\mathcal{X} \times \mathcal{Y}$. Un campione viene quindi descritto dalla coppia $(\mathbf{X}, Y)$ dove $\mathbf{X}$ e Y sono variabili casuali che seguono la distribuzione $\mathcal{P}$. Inoltre $\mathcal{P}$ può essere decomposta in $\mathcal{P} = \mathcal{P}_y(Y|\mathbf{X})\mathcal{P}_x(\mathbf{X})$ dove $\mathcal{P}_y(Y|\mathbf{X})$ è la probabilità condizionata dell'etichetta Y dato un vettore di attributi, mentre $\mathcal{P}_x(\mathbf{X})$ rappresenta la distribuzione dei dati. L'utilizzo di una distribuzione così rappresentata consente di effettuare due assunzioni: i dati seguono la distribuzione $\mathcal{P}_x(\mathbf{X})$ e non hanno la stessa probabilità di apparire, inoltre la presenza della probabilità condizionata sulle etichette, $\mathcal{P}_y(Y|\mathbf{X})$, implica che ad un dato vettore di attributi può essere assegnata più di una etichetta.

L'ultima assunzione necessaria è che ogni campione sia statisticamente indipendente dagli altri. Assunzione non sempre soddisfatta in situazioni reali, ma che rimane comunque un'ipotesi di lavoro nello sviluppo della teoria dell'apprendimento statistico.

Infine, come anticipato nella sezione 1.1, si necessita di una rigorosa definizione della *funzione di loss*, o funzione obiettivo, in grado di quantificare l'errore commesso dal predittore h nel tentativo di associare una etichetta ad un dato vettore degli attributi. La loss viene definita come $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$, una funzione matematica a valori reali non negativa.

A questo punto si definisce il rischio $\mathcal{R}$ associato al modello h come il valore atteso della funzione loss:

$$\mathcal{R}(h) = \mathbb{E}\left[l(Y, h(\mathbf{X})\right] = \int l(Y, h(\mathbf{X}))d\mathcal{P} \tag{1.1}$$

L'obiettivo finale di un algoritmo di apprendimento è trovare un'ipotesi h^* tra una classe fissa di funzioni $\mathcal{H}$ per cui il rischio atteso $\mathcal{R}(h)$ è minimo:

$$h^* = \arg\min_{h \in \mathcal{H}} R(h). \tag{1.2}$$

L'equazione 1.1 non è calcolabile in quanto non si conosce a priori la distribuzione dei dati, per cui l'unica possibilità è quella di calcolarne una sua approssimazione utilizzando i dati che abbiamo a disposizione. A tal proposito si introduce l'approccio di *minimizzazione del rischio empirico* (*ERM* dall'inglese

Empirical Risk Minimization [Zhang, 2010]), dove l'equazione del rischio è definita come la media delle funzioni di perdita ottenute da un modello h su un dato *training set* contenente n istanze:

$$\mathcal{R}_{\text{emp}}(h) = \frac{1}{n}\sum_{i=1}^{n} l(h(\mathbf{x_i}), y_i) \tag{1.3}$$

I risultati ottenuti da *Mohri et al* [Mohri et al., 2018] mostrano come questo approccio sia in grado di identificare, in base alla dimensione del train set e alla cardinalità di $\mathcal{H}$, un predittore h^*_{emp} le cui prestazioni sono vicine a quelle del miglior predittore possibile per la distribuzione $\mathcal{P}$, all'interno di $\mathcal{H}$ (h^*).
Data una distribuzione $\mathcal{P}$, una funzione di loss l ed un train set $\mathcal{S}$ di dimensione m, definiti h^* ed $h_{\mathcal{S}}$ i predittori contenuti in $\mathcal{H}$ che minimizzano rispettivamente il rischio atteso $\mathcal{R}$ ed il rischio empirico $\mathcal{R}_{emp}$, si può dimostrare [Shalev-Shwartz and Ben-David, 2014] che:

$$|\mathcal{R}(h^*) - \mathcal{R}_{emp}(h_{\mathcal{S}})| \leq \sqrt{\frac{2}{m}\log\left(\frac{2|\mathcal{H}|}{\delta}\right)} \tag{1.4}$$

è valida con una probabilità di almeno $1-\delta$ che varia in base all'estrazione casuale del train set $\mathcal{S}$, con $\delta \in (0,\ 1)$. Questo risultato sta a significare che con un'arbitraria probabilità, per un train set abbastanza grande rispetto alla cardinalità della classe dei predittori disponibili, le performance del predittore che minimizza l'errore sul training set sono simili a quelle del miglior predittore che l'algoritmo di apprendimento può produrre per il problema specifico. Risultati simili [Massart and Nédélec, 2006] si ottengono nel caso in cui la cardinalità di $\mathcal{H}$ non sia finita.

Questi risultati definiscono l'approccio tipico da adottare nell'addestramento di un algoritmo supervisionato: in primis bisogna dividere il data set disponibile in train e test set; successivamente si può applicare l'approccio ERM sopra descritto al fine di ottenere un modello che minimizzi il rischio empirico sul train set, infine vanno calcolate le performance del modello sul test set al fine di valutare le sue capacità di generalizzazione del problema in esame.

1.4 Limiti dell'approccio Batch

Il paradigma tradizionale dell'apprendimento automatico descritto nella sezione 1.3 presenta delle limitazioni che ne rendono l'applicazione in alcuni scenari molto difficile. In primis per poter applicare un tipo di approccio batch è necessario che i dati siano disponibili prima che abbia inizio il processo di addestramento dell'algoritmo. Questo non è sempre possibile in quanto la dimensione dei data set può essere troppo grande per essere mantenuta interamente in un device di archiviazione di massa, come ad esempio nei dati provenienti da sensori o dal traffico in rete; inoltre i processi che generano dati costantemente li producono in maniera sequenziale (ad esempio la raccolta di dati circa il comportamento di utenti negli e-commerce). Solitamente gli algoritmi batch non tengono conto della sequenzialità dei dati, bensì il loro addestramento viene effettuato previo un ricampionamento casuale del data set di partenza.

Un altro grande problema dell'approccio batch consiste nell'assunzione che la distribuzione alla base dei campioni sia fissa. Questo non è sempre garantito in molti task attuali, come l'online advertising, in quanto possono presentare quello che viene chiamato **concept drift**. Questo implica che la distribuzione statistica alla base del processo di generazione dei dati è non stazionaria e dunque può variare nel tempo.

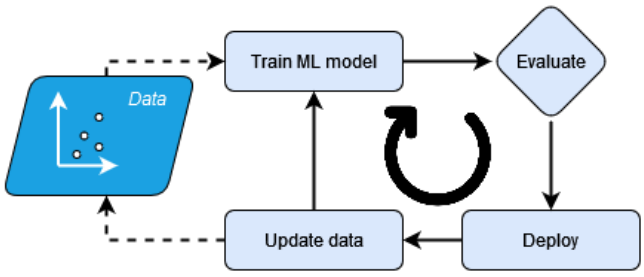


Figura 1.4: Schema di funzionamento batch learning

L'applicazione di algoritmi batch in scenari di questo tipo è soggetta ad un calo nelle performance predittive del modello. Inoltre risulta essere poco efficiente in quanto, per aggiornare un modello rendendolo in grado di riconoscere i

cambiamenti, è necessario effettuare un addestramento ex novo (figura 1.4) sulla nuova distribuzione.

Capitolo 2

Online Machine Learning

Il paradigma dell'online learning cambia radicalmente la struttura del processo di apprendimento. Nell'online learning supervisionato, i dati vengono processati dagli algoritmi sequenzialmente ed uno alla volta. Nello specifico, l'addestramento procede per passi successivi, in cui il modello riceve un campione $\mathbf{x}$ su cui effettua una previsione $\hat{y}$; a questo punto riceve il feedback contenente l'etichetta corretta y associata al campione $\mathbf{x}$ ed utilizza questa informazione per aggiornare i parametri interni migliorando le sue performance. In figura 2.1 viene mostrato lo schema di funzionamento degli algoritmi di online learning.

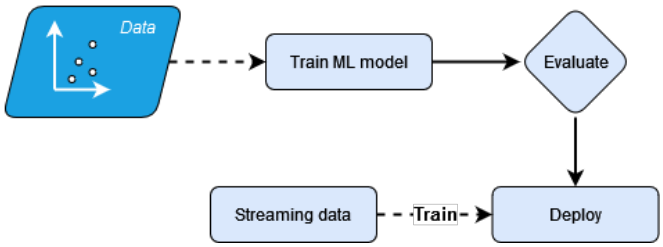


Figura 2.1: Schema di funzionamento online learning

2.1 Impostazione del problema

Il problema dell'online learning è definito come segue: sia $\mathbf{x}_t \in \mathcal{X}$ un singolo campione ricevuto dal modello all'istante t, ed $y_t \in \mathcal{Y}$ l'etichetta associata ad $\mathbf{x}_t$,

in maniera del tutto simile a come descritto nella sezione 1.3, un predittore, o modello, è una funzione $h : \mathcal{X} \to \mathcal{Y}$ appartenente alla classe $\mathcal{H}$. Definito quindi h_t come il modello all'istante t, una loss $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ ed infine la loss subita dal modello all'istante t come $l_t(h) = l(y_t, h_t(\mathbf{x}_t))$, il processo di apprendimento su una sequenza di T campioni viene descritta dall'algoritmo 1.

Algoritmo 1 Processo di apprendimento online supervisionato

1: Inizializzazione del predittore: h_1
2: **for** $t = 1, 2 \ldots, T$ **do**
3: Ricevere il campione $\mathbf{x}_t$
4: Calcolare la previsione: $\hat{y}_t = h_t(\mathbf{x}_t)$
5: Ricevere la vera etichetta y_t
6: Calcolare la loss $l_t(h_t)$
7: Aggiornare il modello $h_t \to h_{t+1}$
8: **end for**

Per La valutazione delle performance di questi algoritmi si introduce il *rischio sequenziale*:

$$\frac{1}{T}\sum_{t=1}^{T} l(y_t, h_t(\mathbf{x}_t)) = \frac{1}{T}\sum_{t=1}^{T} l_t(h_t) \tag{2.1}$$

Un'altra metrica molto utilizzata nel calcolo delle performance viene chiamata *regret* ($\mathcal{R}_T$) ed è definita come la differenza tra la loss accumulata su tutti i dati visti dal modello, e la loss che si genererebbe utilizzando il miglior predittore per quel set di dati, che può essere conosciuto solo dopo l'ultimo round:

$$\mathcal{R}_T = \sum_{t=1}^{T} l_t(h_t) - \min_{h \in \mathcal{H}} \sum_{t=1}^{T} l_t(h_t) \tag{2.2}$$

L'obiettivo degli algoritmi di online machine learning è quindi quello di ottenere un valore di *regret* che cresca sub-linearmente all'aumentare del numero di campioni T ($\mathcal{R}_T = o(T)$), facendo si che $\lim_{T\to\infty} \frac{\mathcal{R}_T}{T} = 0$ garantendo che l'algoritmo converga a performance simili a quelle del miglior predittore possibile per quella sequenza di dati.

Al fine di garantire prove teoriche alla convergenza di $\mathcal{R}_T$ bisogna fare alcune assunzioni sulla natura della funzione di loss. Data una classe di modelli parametrici ed una funzione di loss convessa, i problemi di online machine learning possono essere riformulati come task di *ottimizzazione convessa*

[Hoi et al., 2018]. Un esempio per questo tipo di algoritmi è la *Discesa online del Gradiente* (*OGD*, Online Gradient Descent [Zinkevich, 2003]) in cui si utilizza il gradiente della funzione di loss ad ogni round per calcolare la regola di aggiornamento $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla l_t(\mathbf{w}_t)$, dove $\mathbf{w}_t$ sono i parametri che definiscono il modello h_t, mentre η_t è il learning rate e determina l'entità dell'aggiornamento ad ogni passo.

2.2 Cambiamenti nelle distribuzioni: Concept drift

Gli algoritmi di apprendimento spesso lavorano in ambienti dinamici, che sono sottoposti a cambiamenti inaspettati. Questo porta ad una variazione nelle relazioni ingresso uscita delle distribuzioni alla base del processo in esame, con la conseguente obsolescenza dei modelli ottenuti dall'addestramento sui dati precedenti. Per cui una delle proprietà che si richiedono a questi tipi di algoritmi è la capacità di adattamento a questi cambiamenti, incorporando al loro interno i nuovi dati. Se il processo che genera i dati, input del modello che si utilizza, è **non stazionario** (accade in molte applicazioni reali), il concetto alla base del processo, che cerchiamo di predire, può cambiare nel tempo. In questa sezione viene approfondito il fenomeno del **concept drift** e vengono caratterizzati gli algoritmi in grado di adattarsi a questo fenomeno.

2.2.1 Definizioni ed impostazione del problema

In accordo con le definizioni di *Gama et al.* [Gama et al., 2014], definito un problema di learning supervisionato, caratterizzato da una distribuzione di probabilità congiunta $\mathcal{P}(\mathbf{X}, y)$, una classificazione può essere descritta dalla probabilità a priori della classe $\mathcal{P}(y)$ e dalla funzione densità di probabilità di ogni classe (funzione di *verosimiglianza*) $\mathcal{P}(\mathbf{X}|y)\ \forall\ y = 1, \ldots, c$ (c è il numero di classi), e la decisione viene presa in base alla probabilità a posteriori delle classi, che per la classe y viene rappresentata come:

$$\mathcal{P}(y|\mathbf{X}) = \frac{\mathcal{P}(y)\mathcal{P}(\mathbf{X}|y)}{\mathcal{P}(\mathbf{X})} \tag{2.3}$$

A questo punto si definisce formalmente il *concept drift* tra i dati agli istanti t_0 e t_1 come

$$\exists \mathbf{X} : \mathcal{P}_{t_0}(\mathbf{X}, y) \neq \mathcal{P}_{t_1}(\mathbf{X}, y) \tag{2.4}$$

dove $\mathcal{P}_{t_0}(\mathbf{X}, y)$ denota la distribuzione di probabilità congiunta al tempo t_0 tra il set delle variabili di input $\mathbf{X}$ e le variabili target y. Alla luce di queste definizioni, un cambiamento nei dati può essere caratterizzato con i cambiamenti nelle componenti di queste relazioni. In altri termini possono variare: la probabilità a priori delle classi $\mathcal{P}(y)$ e la funzione di verosimiglianza $\mathcal{P}(\mathbf{X}|y)$, con consecutiva variazione delle probabilità a posteriori delle classi $\mathcal{P}(y|\mathbf{X})$.

Nello specifico possiamo collocare le diverse tipologie di drift che possono verificarsi nelle distribuzioni in base alla loro capacità di influenzare o meno le performance predittive del modello decisionale, alle modalità di cambiamento delle distribuzioni ed anche alle caratteristiche temporali nell'evoluzione del drift.

Dato un set di train ed un modello decisionale, quest'ultimo riconducibile ad un confine decisionale (figura 2.2), un **concept drift** si presenta come una

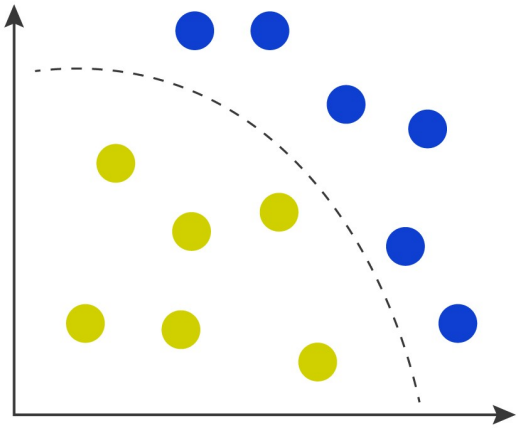
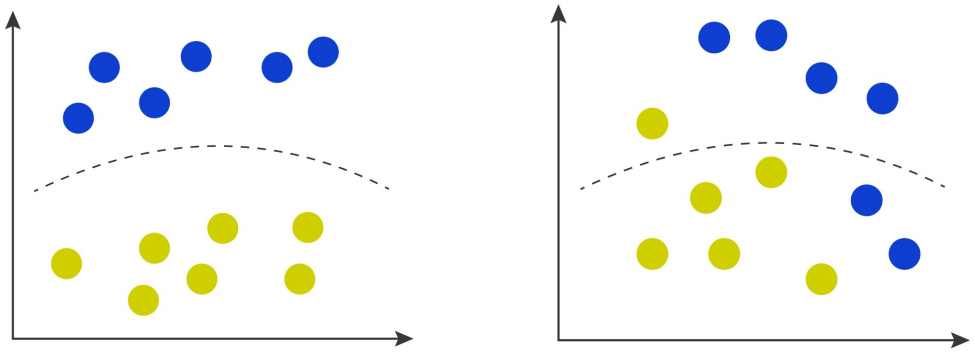


Figura 2.2: Train set e confine decisionale

variazione nella probabilità a posteriori della classe $\mathcal{P}(y|\mathbf{X})$ e può presentarsi sia con una variazione della distribuzione dei dati $\mathcal{P}(\mathbf{X})$ (figura 2.3a) che senza (figura 2.3b).



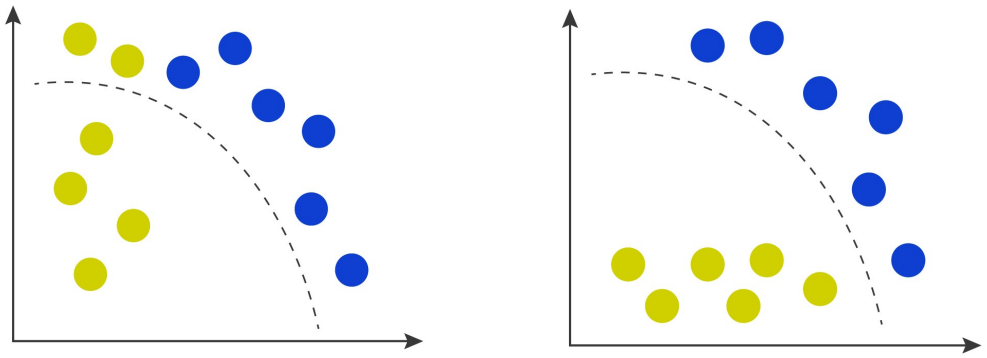
(a) con variazione di $\mathcal{P}(\mathbf{X})$

(b) senza variazione di $\mathcal{P}(\mathbf{X})$

Figura 2.3: Concept drift

La seconda famiglia viene definita **data drift** e fa riferimento unicamente ad una variazione nella distribuzione degli input $\mathcal{P}(\mathbf{X})$ (features drift) e/o degli output $\mathcal{P}(y)$ (label drift). In questo caso è possibile distinguere ulteriormente in drift **reali** o **virtuali**. Queste ultime definizioni fanno riferimento alla capacità o meno del drift di influenzare negativamente le performance del modello. È infatti possibile che il confine decisionale resti comunque valido nonostante le caratteristiche di input varino nel tempo (figura 2.4b). Nei casi opposti, in cui vengono invece influenzate le performance predittive del modello, si parla di drift **reale** (figura 2.4a).

Per fare un esempio concreto, consideriamo un flusso di notizie online di articoli sul settore immobiliare, e prendiamo in considerazione il task di classificazione di notizie *rilevanti* ed *irrilevanti* per un dato utente. Supponendo che l'utente stia cercando una nuova casa, allora le notizie contenenti informazioni su case in vendita saranno rilevanti, mentre le notizie sulle case vacanza no. Nel caso in cui dovesse cambiare lo stile di scrittura delle notizie le notizie rilevanti per il nostro utente resterebbero immutate. Questo scenario è riconducibile ad un data drift. Se invece a cambiare fossero le intenzioni dell'utente, in quanto, ad esempio, l'utente ha acquistato un'abitazione ed ora cerca invece case vacanza, in questo caso le notizie circa le case abitative diventano irrilevanti mentre quelle



(a) Reale (b) Virtuale

Figura 2.4: Data drift

inerenti case vacanza diventano rilevanti. Quest'ultimo è un esempio di concept drift.

Infine è possibile distinguere i drift in base alla loro evoluzione temporale. In figura 2.5 vengono riportate le forme temporali in cui un drift può manifestarsi.

- **improvviso**: un drift può manifestarti passando da un concetto all'altro improvvisamente,
- **incrementale**: consiste nella manifestazione di concetti intermedi fino alla stabilizzazione dei dati su un concetto,
- **graduale**: si manifesta presentando dati appartenenti ad un altro concetto in numero sempre maggiore,
- **ricorrente**: quando i concetti si alternano temporalmente.

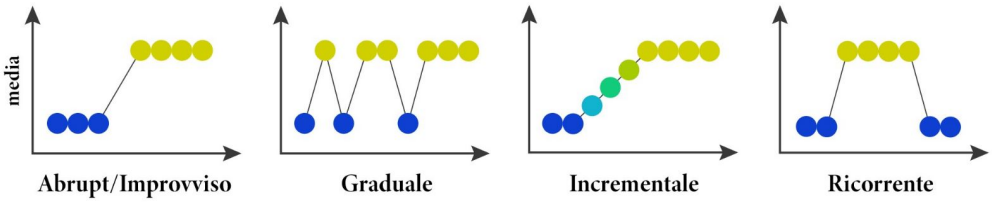


Figura 2.5: Evoluzione temporale dei drift

2.2.2 Procedura per l'apprendimento adattivo

I modelli predittivi che operano in questi contesti necessitano quindi di meccanismi in grado di rilevare ed adattare le proprie caratteristiche all'evoluzione dei dati. In questo senso i modelli predittivi devono essere in grado di:

1. rilevare il concept drift distinguendoli da rumore ed outliers;
2. non solo imparare dai nuovi dati, ma anche dimenticare le informazioni apprese dai vecchi;
3. operare in tempi inferiori alla frequenza di ricezione dei dati e con un limite alla memoria utilizzata.

Per quanto riguarda il primo punto esistono numerosi algoritmi in grado di rilevare la presenza di un drift nelle distribuzioni.
Ad esempio *ADWIN* [Bifet and Gavaldà, 2007] (ADaptive WINdowing) è un metodo per il rilevamento del drift molto popolare supportato da prove matematiche sul suo funzionamento. Semplificando, ADWIN utilizza una finestra di dimensione variabile per tenere traccia degli ultimi campioni W. Successivamente divide la finestra in due sotto finestre $(W_0,\ W_1)$ che verranno utilizzate nella valutazione della presenza o meno di un drift. L'idea sottostante all'algoritmo è semplice: ogni qual volta che due sotto finestre abbastanza grandi di W presentano **medie** sufficientemente distinte, si può concludere che le distribuzioni contenute al loro interno sono diverse, quindi si è verificato un drift. Quando si verifica un drift W_0 viene sostituita da W_1 e si inizializza una nuova finestra contenente nuovi dati. Per determinare se due finestre appartengono alla stessa distribuzione ADWIN utilizza un livello di significatività $\delta \in (0,\ 1)$.

Per quanto riguarda la capacità dei modelli di imparare dai nuovi dati distinguiamo due diverse modalità per l'apprendimento online:

- **Apprendimento a singolo esempio:**

  gli algoritmi in grado di apprendere in questa modalità sono in grado di adattare i propri parametri analizzando un dato alla volta, per cui non necessitano di un meccanismo in grado di mantenere in memoria gli ultimi dati visti. Solitamente l'update di questo tipo di modelli viene guidato

dall'errore commesso. Quando un nuovo campione viene osservato, viene effettuata la previsione dal modello corrente. Successivamente viene fornita l'etichetta che viene utilizzata nel calcolo della loss, utilizzata, se necessario, per aggiornare il modello.

- **Apprendimento ad esempi multipli:**

  un secondo approccio è quello di mantenere un modello predittivo consistente con un set di dati recenti mantenendoli in memoria ed utilizzandoli per l'addestramento. La sfida chiave in questa tipologia di algoritmi è quella di scegliere correttamente la dimensione della finestra da utilizzare. Nello specifico una finestra piccola riflette accuratamente la distribuzione corrente, quindi assicura un rapido adattamento al drift, ma, durante periodi stabili, finestre troppo piccole possono risultare in un degrado nelle prestazioni del sistema. D'altro canto finestre grandi danno prestazioni migliori in periodi stabili ma implicano una lenta reattività ai cambiamenti. Generalmente si predilige l'utilizzo di una finestra di dimensione variabile per algoritmi di questo tipo. In questo lavoro di tesi è stato approfondito un algoritmo facente parte di questa famiglia.

Infine gli algoritmi di apprendimento online necessitano di un meccanismo che li renda in grado di dimenticarsi (forgetting) dei dati precedenti al verificarsi di un drift, al fine di aumentare la loro capacità di adattarsi ai cambiamenti. Va precisato che avendo un meccanismo di questo tipo è necessario che gli algoritmi siano robusti al rumore. Anche in questo caso possiamo distinguere due tipologie di meccanismi:

- **abrupt** forgetting (improvviso):

  ad ogni istante di tempo, un insieme di osservazioni definisce una finestra di informazioni considerata nell'apprendimento. L'abrupt forgetting fa riferimento ad un meccanismo per il quale i dati sono o dentro o fuori la finestra di addestramento.

- **gradual** forgetting (graduale):

in questo caso nessun esempio visto viene scartato completamente ma viene associato un peso ad ogni dato rappresentante la sua età. L'intuizione di questi algoritmi è molto semplice e si basa sull'assunzione che l'importanza dei dati dovrebbe decrescere nel tempo.

Va notato che generalmente più gli algoritmi sono in grado di dimenticare rapidamente i dati meno recenti, più essi sono in grado di adattarsi ai nuovi, rendendoli quindi robusti al drift. D'altro canto dimenticare improvvisamente i dati rende gli algoritmi molto sensibili al rumore.

2.3 XGBoost Adattivo

In questo lavoro di tesi si è esplorato un adattamento dell'algoritmo *XGBoost*, poiché non esiste una sua implementazione ufficiale nelle librerie più diffuse in tema di *online machine learning*. Questo algoritmo è stato scelto perché spesso è importante garantire la trasparenza dei modelli di machine learning, in modo che gli utenti finali, spesso non tecnici, possano comprenderne il funzionamento e non sollevare scetticismi. La "spiegabilità" di questi modelli può essere ottenuta attraverso delle metodologie che verranno descritte nei successivi capitoli, mentre per algoritmi *black box* come algoritmi *deep* è molto più difficile.

Il potenziamento estremo del gradiente (**XGB** *eXtreme Gradient Boosting*) è un popolare algoritmo di machine learning supervisionato, presentato da *Carlos Guestrin* e *Tianqi Chen* [Chen and Guestrin, 2016] nel 2016, basato su un insieme di algoritmi ad albero decisionale. Il suo funzionamento si basa sulla metodologia del *boost* del gradiente ([Friedman et al., 2000]) che utilizza un insieme di predittori detti *weak learner* i cui output contribuiscono al miglioramento delle performance del modello complessivo.

In questa sezione verranno introdotti i concetti teorici dietro questo algoritmo che ha rivoluzionato il panorama dell'apprendimento automatico, al fine di comprendere meglio il suo adattamento al paradigma dell'online learning.

2.3.1 XGBoost

XGBoost è un algoritmo di *boosting* per la classificazione e la regressione basato su alberi decisionali. Il boosting è un metodo ensemble che combina più modelli deboli per creare un modello forte. XGBoost utilizza una tecnica di gradient boosting che consiste nel costruire un modello sequenzialmente, dove ogni modello cerca di correggere gli errori del modello precedente.

Il modello di decision tree utilizzato in XGBoost è un albero di decisione (regularized decision tree), che utilizza la regolarizzazione per evitare l'overfitting. La regolarizzazione è ottenuta attraverso la limitazione della profondità degli alberi e della quantità di foglie.

Formalizzando, dato $S = \{(\mathbf{x}_i, y_y) : |S| = n, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R}\}$ un data set di n esempi, composti da coppie di vettori degli attributi m-dimensionali $\mathbf{x}_i$ ed etichette y_i, e dato F lo spazio degli alberi, un modello ensamble utilizza K funzioni additive per predire l'output $\hat{y}_i$:

$$\hat{y}_i = \Phi(\mathbf{x}_i) = y_0 + \sum_{k=1}^{K} \epsilon f_k(\mathbf{x}_i),\ f_k \in F, \tag{2.5}$$

dove y_0 ed ϵ (epsilon) rappresentano rispettivamente una predizione base ed il learning rate definibili dall'utente.

Definita la funzione di loss $l(\hat{y}_i, y_i)$ per una data istanza di S come una funzione convessa e differenziabile, XGBoost minimizza una funzione obiettivo **regolarizzata** al fine di definire il set di alberi che verranno usati nel modello finale:

$$L(\Phi) = \sum_{i=1}^{n} l(\hat{y}_i, y_i) + \sum_{k=1}^{K} \Omega(f_k), \tag{2.6}$$

dove il termine di regolarizzazione $\Omega(f_k) = \gamma T + \frac{1}{2}\lambda O^2$, che penalizza gli alberi strutturalmente più complessi, è composto da T che rappresenta il numero di nodi foglia dell'albero, O che rappresenta l'output, ed infine γ e λ sono dei termini definibili dall'utente che si utilizzano per incoraggiare il pruning dell'albero evitando l'overfitting.

L'equazione 2.6 contiene delle funzioni come parametri e non può essere ottimizzata direttamente con i metodi classici, per cui si addestra il modello in maniera

additiva. Definita quindi y_i^t come la previsione sull'istanza i-esima di S effettuata dall'albero f_t, aggiungiamo un nuovo albero f_{t+1} all'ensamble in modo che minimizzi la funzione obiettivo:

$$L(t) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^t + f_{t+1}(\mathbf{x}_i)) + \Omega(f_{t+1}). \tag{2.7}$$

Per semplificare l'ottimizzazione dell'equazione 2.7 utilizziamo una sua approssimazione derivata dal suo sviluppo di *Taylor* al secondo ordine, ottenendo:

$$\begin{cases} L(t) & = \sum_{i=1}^{n} [l(y_i, \hat{y}_i^t) + g_i f_{t+1}(\mathbf{x}_i) + \frac{1}{2} h_i f_{t+1}^2(\mathbf{x}_i)] + \Omega(f_{t+1}) \\ g_i & = \frac{\partial}{\partial \hat{y}_i^t} l(y_i, \hat{y}_i^t) \\ h_i & = \frac{\partial^2}{\partial^2 \hat{y}_i^t} l(y_i, \hat{y}_i^t) \end{cases} \tag{2.8}$$

A questo punto, eliminate le costanti, che non influiscono nella minimizzazione della funzione, e definito $I_j = \{i \mid q(\mathbf{x}_i) = j\}$ come l'insieme delle istanze $\mathbf{x}_i$ mappate dalla struttura dell'albero q nella foglia j, possiamo riscrivere l'equazione 2.8 espandendo $\Omega(f_{t+1})$:

$$\begin{aligned} L(t) &= \sum_{i=1}^{n} [g_i f_{t+1}(\mathbf{x}_i) + \frac{1}{2} h_i f_{t+1}^2(\mathbf{x}_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} O_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) O_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) O_j^2] + \gamma T \end{aligned} \tag{2.9}$$

Considerata un'unica foglia j, l'ottimizzazione dell'equazione 2.9 per j rispetto all'output O_j determina il suo valore di output ottimo come:

$$O_j^* = -\frac{\sum_{j \in I_j} g_i}{\sum_{j \in I_j} h_i + \lambda}. \tag{2.10}$$

L'equazione 2.10 può inoltre essere utilizzata per calcolare il guadagno G di uno split. Definiti I_R ed I_L t.c. $I_R \cup I_L = I_j$ come i set di istanze di S dei nodi rispettivamente destro e sinistro dopo lo split di I_j, il guadagno è definito come segue:

$$G = \frac{1}{2} \left[\frac{\sum_{j \in I_r} g_i}{\sum_{j \in I_r} h_i + \lambda} + \frac{\sum_{j \in I_l} g_i}{\sum_{j \in I_l} h_i + \lambda} - \frac{\sum_{j \in I_j} g_i}{\sum_{j \in I_j} h_i + \lambda} \right] - \gamma. \tag{2.11}$$

Nell'algoritmo vengono valutati solo valori di $G > 0$ e viene effettuato lo split che porta al guadagno magiore. Nell'equazione 2.11 il termine γ incoraggia il

pruning delle foglie dell'albero, per cui con valori di γ grandi si applicano split che portano ad un guadagno elevato.

Una formulazione di questo tipo permette all'algoritmo di essere utilizzato sia per task di regressione che per task di classificazione. Infatti, l'output ottimo definito dall'equazione 2.10 e l'equazione del guadagno G nell'equazione 2.11, sono definite nei termini di *gradienti* g ed *hessiane* h della funzione di loss (equazione 2.8).

Preso come esempio un task di classificazione binaria (task affrontato in questo lavoro di tesi) la funzione di loss maggiormente utilizzata in questi casi è la funzione **logistica** definita come $l(\hat{y}_j^i, y_j) = -[y_j \log(\hat{y}_j^i) + (1) - y_i \log(1 - \hat{y}_i^i)]$ dove $\hat{y}_i^i$ è da interpretare come la *probabilità* della classe predetta dal nodo i. Calcolando gradiente ed hessiana della logistica e sostituendo in 2.10, per un nodo foglia i si ottiene l'output:

$$O_i^* = \frac{\sum_{j \in I_j} (y_j - \hat{y}_j^{i-1})}{\sum_{j \in I_j} [\hat{y}_j^{i-1}(1 - \hat{y}_j^{i-1})] + \lambda}, \tag{2.12}$$

dove $\hat{y}_j^{i-1}$ è la probabilità predetta dal nodo $i-1$ padre del nodo i.

2.3.2 XGBoost adattivo

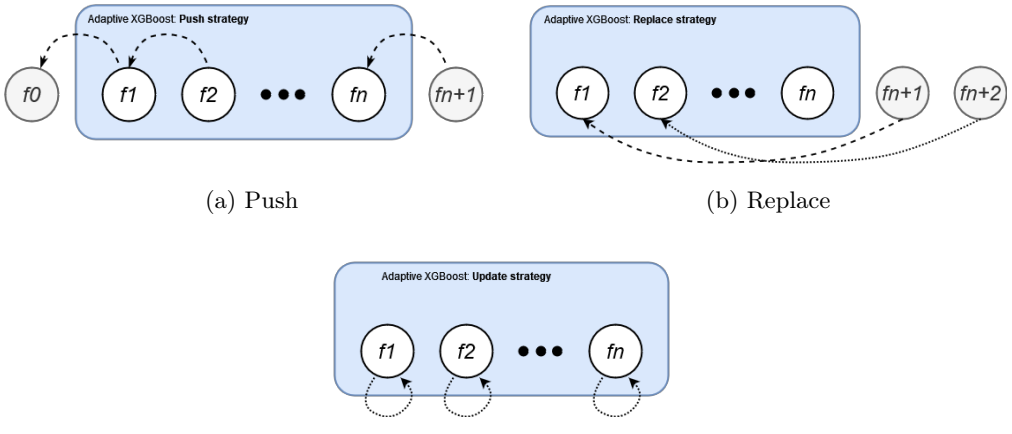
In questa sezione viene presentato un adattamento dell'algoritmo XGBoost descritto nella sezione 2.3.1 per dati streaming, quindi una sua implementazione online che viene presentata in [Montiel et al., 2020]. *Adaptive XGBoost* (AXGB) utilizza delle strategie incrementali per creare l'ensamble. Al contrario dell'algoritmo batch infatti, AXGB utilizza un sotto insieme dei dati provenienti da finestre non sovrapposte per addestrare i weak learner (f_i). Nello specifico, dato un buffer $w = \{(\mathbf{x}_i, y_y) \ : \ |w| = W, \mathbf{x}_i \in \mathbb{R}^m, y_i \in \mathbb{R}\}$ in cui vengono salvati i nuovi esempi, AXGB procede nell'addestramento di un nuovo weak learner f_k in maniera incrementale come per la sua versione classica:

$$\Phi(k) = y_0 + \sum_{k=1}^{K} \epsilon f_k(w_k) = \Phi(k-1) + \epsilon f_k(w_k). \tag{2.13}$$

Dato che il set di dati è potenzialmente a cardinalità infinita, è necessario definire delle strategie per l'aggiornamento e la sostituzione degli alberi nell'en-

samble. In questo senso, vengono presentate due diverse strategie provenienti dall'implementazione di *Montiel et al.* e la strategia introdotta in questo lavoro di tesi:

- **Push**: in figura 2.6a viene mostrato come l'ensamble viene implementato come una coda di dimensione fissa, gestita seguendo una strategia FIFO (*First In First Out*). Quando l'ensamble è pieno, vengono rimossi i modelli più "vecchi" ed introdotti i nuovi.
- **Replace**: in figura 2.6b viene mostrato come i nuovi modelli generati sostituiscono iterativamente i modelli all'interno dell'ensamble.
- **Update**: in figura 2.6c viene mostrato come i weak learner non vengono sostituiti direttamente, bensì vengono aggiornati utilizzando i nuovi dati disponibili. Questa strategia è stata poi suddivisa ulteriormente in update **dinamico**, in cui i weak learner vengono aggiornati iterativamente, e **statico** in cui i nuovi dati, oltre che per addestrare i nuovi modelli che entrano a far parte dell'ensamble, vengono utilizzati per aggiornare i vecchi modelli, quando l'ensamble è saturo i dati vengono utilizzati per aggiornare tutti i weak learner.



(a) Push

(b) Replace

(c) Update

Figura 2.6: Strategie di aggiornamento AXGBoost

Possiamo quindi collocare AXGB nella famiglia di algoritmi online ad apprendimento a esempi multipli. In particolare le strategie di push e replace contengono implicitamente un meccanismo per dimenticare i dati più vecchi.

Va notato inoltre che con una dimensione fissa della finestra di addestramento $|w| = W$, bisogna attendere $K \times W$ campioni per l'addestramento di un ensamble, caratterizzato da K weak learner. Uno dei problemi che potrebbero insorgere addestrando l'algoritmo in questa maniera, è che le performance sono sub ottimali all'inizio dello stream. Per appianare questo problema, AXGB utilizza una finestra di dimensione variabile. Nello specifico definite W_{min} e W_{max} come le dimensioni rispettivamente minima e massima della finestra di apprendimento, la dimensione della finestra all'istante i viene definita come $W_i = \min(2^i W_{min},\ W_{max})$.

Nonostante le strategie di update e replace descritte in precedenza risolvono indirettamente il problema del concept drift, l'adattamento alle nuove distribuzioni potrebbe essere molto lento. inoltre per la nuova strategia introdotta generalmente questo non è valido, per cui si necessita di un meccanismo efficiente per adattare l'algoritmo ai nuovi dati.
AXGB utilizza ADWIN per rilevare il drift e per innescare un meccanismo per aggiornare l'ensamble: quando viene rilevato un drift all'istante i, la dimensione della finestra di apprendimento W_i viene riportata al valore iniziale W_{min}. A questo punto si procede con un addestramento ex-novo. Nel caso della strategia push vengono eliminati i weak learner addestrati in precedenza, negli altri due casi l'indice di addestramento viene riportato al valore iniziale. Oltre a questo è stato introdotto il concetto di *background learner*: durante l'addestramento dell'ensamble viene addestrato un secondo insieme di weak learner della stessa dimensione del principale ma con caratteristiche diverse in termine di dimensioni della finestra di addestramento W_{min}, W_{max} al fine di ottenere un modello estremamente coerente unicamente con gli ultimi dati a disposizione. Nel caso in cui viene rilevato un drift, i weak learner del modello principale vengono sostituiti con quelli del background learner e si procede con l'addestramento.

Capitolo 3

Implementazione e benchmarks

In questo capitolo viene descritta l'implementazione dell'algoritmo AXGB descritto nella sezione 2.3.2.
L'algoritmo è stato sviluppato utilizzando il linguaggio di programmazione **Python** nella sua versione 3.8 [Van Rossum and Drake, 2009]. In particolare sono state estese le funzionalità di una delle librerie più in auge per quanto riguarda la tematica dello streaming machine learning: **River** [Montiel et al., 2021]. Successivamente vengono valutate le performance su dataset di benchmark messi a disposizione dalla libreria stessa ed infine valutate le performance su data set contenenti concept drift.

3.1 Implementazione

River[1] è una libreria open source per python il cui obiettivo è quello di mettere a disposizione del programmatore diversi modelli adatti alla tematica dello streaming machine learning. Oltre ai classici modelli di regressione e classificazione, sono disponibili anche gli algoritmi inerenti il rilevamento del concept drift, la definizione di algoritmi ensamble ed anche algoritmi specifici per lo streaming machine learning come ad esempio gli alberi di *Hoeffding* [Hulten et al., 2001] ed EFDT (Extremely Fast Decision Tree) [Bifet et al., 2017].

[1] https://github.com/online-ml/river

L'implementazione di questa libreria è il risultato dell'unione di due librerie meno recenti (*Creme*[2] e *Scikit-multiflow*[3], quest'ultima utilizzata nella vecchia implementazione di AXGB) e sta diventando lo standard de facto per quanto riguarda lo streaming machine learning.
Lo sviluppo dell'intero algoritmo è stato versionato utilizzando git[4], mentre la repository di sviluppo è stata strutturata sulla base del template per progetti di data science **cookiecutter**[5].

Nell'implementazione di AXGB sono state utilizzate le classi base di libreria. Nello specifico sono stati implementati:

1. Albero regolarizzato: un semplice wrapper del framework xgboost[6].

2. AXGB base: implementato come un ensamble contenente modelli della classe precedente. Questa è la classe che contiene le definizioni delle strategie di update di modelli, oltre che gli algoritmi per il loro addestramento, il meccanismo di windowing ed il salvataggio temporaneo dei dati, precedentemente descritti.

3. Classificatore AXGB: particolarizzazione della classe precedente con l'utilizzo delle funzioni di loss per i task di classificazione binaria e multiclasse. Inoltre qui vengono implementate le metodologie per il rilevamento del drift e consecutivo adattamento.

L'implementazione può essere riassunta dall'algoritmo 2. Dati in input W_m, W_M, N_L e *Data* rappresentanti la grandezza minima e massima della finestra di apprendimento, il numero di weak learners ed il data set, si inizializzano: la dimensione della finestra al valore minimo, l'indice di addestramento del weak learner ed il buffer dati. Iterando i dati contenuti nel data set, quando la dimensione del buffer dati raggiunge la dimensione della finestra si procede: addestrando e/o aggiornando l'i-esimo weak learner, si determina la nuova dimensione della

[2] https://github.com/MaxHalford/creme
[3] https://github.com/scikit-multiflow/scikit-multiflow
[4] https://git-scm.com/
[5] http://drivendata.github.io/cookiecutter-data-science/
[6] https://xgboost.readthedocs.io/en/stable/python/python_intro.html

finestra ed infine si aggiorna l'indice di addestramento i.

Successivamente si verifica la presenza di un concept drift e, in caso positivo, si reinizializzano le variabili, mentre i modelli costruiti precedentemente vengono sostituiti.

Algoritmo 2 Algoritmo AXGB

Input: $W_m, W_M, N_L, Data$

$W_s \leftarrow W_m$
i $\leftarrow 0$
buffer $\leftarrow$ empty
for all $d \in Data$ **do**
    add d to buffer
    if len(buffer) $\geq W_s$ **then**
        train_weak_learner(buffer, i)
        $W_s \leftarrow \min(2W_s, W_M)$
        buffer $\leftarrow$ empty
        i $\leftarrow$ i+1
        if i $> N_L$ **then**
            i $\leftarrow 0$
        end if
    end if
    if Drift = True **then**
        $W_s \leftarrow W_m$
        i $\leftarrow 0$
    end if
end for

3.2 Benchmarks

Nella valutazione delle performance generali del modello ottenuto sono stati utilizzati dei dataset di benchmark che vengono forniti dalla libreria *River*[7]. I risultati ottenuti vengono confrontati con quelli riportati nella documentazione facendo riferimento unicamente ai data set per la classificazione.

Di seguito, una breve descrizione dei data set utilizzati:

- *Bananas*[8]: un data set per la classificazione binaria generato artificialmente dove le istanze appartengono a diversi cluster aventi una forma a banana.

[7] https://riverml.xyz/0.13.0/benchmarks/

[8] bit.ly/bananas-dataset

Questo data set è composto da due attributi *At1* e *At2* che corrispondono rispettivamente agli assi x e y. Le etichette (-1 e 1) rappresentano invece una delle due forme a banana all'interno del data set.

- *Phishing*[9]: un data set per la classificazione binaria contenente attributi ed etichette relative a pagine web classificate come phishing oppure no. Con l'aumento del numero di transazioni *online*, il problema del *phishing* è diventato molto rilevante soprattutto nei settori dell'*e-banking* ed *e-commerce*. In questo data set sono state identificate diverse caratteristiche relative a siti web "legittimi" e di phishing in 1.353 siti internet differenti. Nello specifico sono stati identificati come phiscing 548 siti internet, contenenti in totale 702 URL (*Uniform Resource Locator*) di phishing e 103 URL sospetti.

- *Image segments*[10]: questo data set contiene attributi per mappare dei segmenti di immagini in sette (7) classi distinte: mattoni, cielo, fogliame, cemento, finestra, sentiero ed erba. Le istanze di questo data set sono state estratte randomicamente da un database di 7 immagini segmentate a mano per creare una classificazione per ogni pixel. Ogni istanza del data set ottenuto rappresenta una regione di immagine 3×3. Gli attributi sono generati tramite trasformazioni dei valori RGB, dei valori di saturazione e tinta e del contrasto tra i pixel adiacenti, nelle regioni considerate.

- *Insects*[11]: contenente letture sensoriali relative al riconoscimento del volo di insetti. Questo data set presenta 6 versioni in cui a variare sono il bilanciamento delle classi, oltre che la tipologia di drift presente all'interno del data set, rendendo più o meno complesso il task. I risultati riportati di seguito fanno riferimento al data set con classi bilanciate (il più semplice in termini di classificazione).

- *Keystroke*[12]: data set per la classificazione multi classe il cui task è l'identi-

[9] https://bit.ly/phishing-dataset
[10] https://bit.ly/image-segments-dataset
[11] https://bit.ly/insects-dataset
[12] https://bit.ly/keystroke-dataset

ficazione degli utenti che digitano una password. Il data set consiste quindi di informazioni circa i tempi di battitura di 51 soggetti, a ognuno dei quali è stato chiesto di digitare una password per 400 volte.

Nella tabella 3.1 vengono riportate le dimensionalità dei data set.

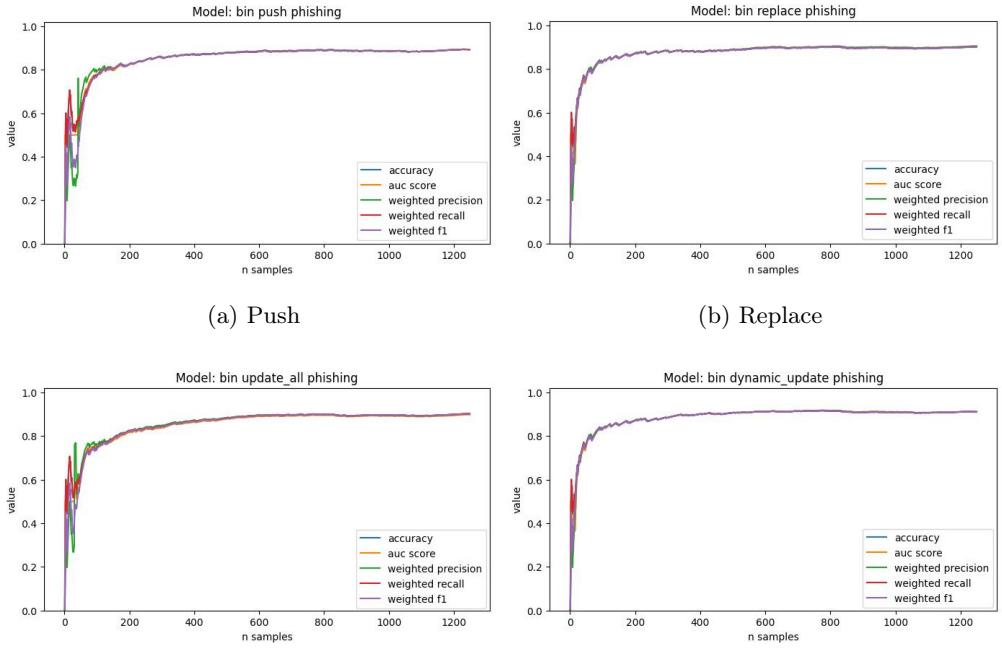
Nome	Classi	Istanze	Attributi
Bananas	2	5.300	2
Phishing	2	1.353	9
Image Segments	7	2.310	18
Insects	6	52.848	33
Keystroke	51	20.400	31

Tabella 3.1: Dimensionalità dei dataset di benchmark

Per la valutazione delle performance generali dell'algoritmo proposto in questo lavoro di tesi, AXGB è stato addestrato su questi dataset in tutte le sue varianti e per confrontato con le performance riportate dagli sviluppatori di river dei seguenti modelli: *alberi di Hoeffding* (HdT), *ADA Boost* (ADAB), *Extremely Fast Decision Tree* (EFDT) e *Random Forest adattiva (ARF)*. Nelle successive formulazioni faremo riferimento ai modelli AXGB ed alle strategie come: $AXGB_s$, $AXGB_d$, $AXGB_p$ ed $AXGB_r$ rispettivamente per le strategie di update statico, dinamico, push ed infine replace.
Le metriche utilizzate sono l'**accuratezza** dei modelli, oltre che lo **score-F1** ($F1 = 2\frac{Precision \cdot Recall}{Precision + Recall}$).

Prima di passare alla osservazione e valutazione dei risultati ottenuti, in figura 3.1 viene riportato un esempio di addestramento. Possiamo notare come inizialmente le performance del modello, che effettua previsioni casuali, scendano bruscamente. Successivamente il modello inizia ad apprendere le relazioni tra attributi ed etichette migliorando gradualmente le sue performance fino a stabilizzarsi intorno ad un valore.

In tabella 3.2 vengono mostrati i risultati dell'addestramento di AXGB nelle sue quattro varianti. I risultati mostrano come, per questi data set, le nuove



(a) Push

(b) Replace

(c) Update statico

(d) Update dinamico

Figura 3.1: Esempio di addestramento online

strategie di *update* hanno performance migliori delle strategie precedenti *push* e *replace*. In tabella 3.3 vengono invece messi a confronto i risultati ottenuti dal modello AXGB con i risultati ottenuti dai modelli precedentemente elencati. Possiamo notare come nei task di classificazione binaria, il modello proposto ottenga i migliori risultati, mentre per quanto riguarda il task di classificazione multi-classe il miglior modello sembra essere la random forest adattiva (ARF), eccezzion fatta per il data set *"image segments"* in cui AXGB performa meglio di tutti gli altri modelli.

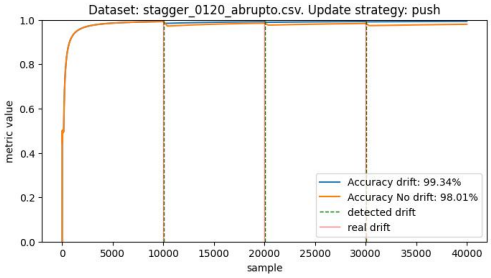
A questo punto passiamo alla valutazione dei meccanismi atti al rilevamento, e consecutivo adattamento, del concept drift. Per conoscere le prestazioni di un rilevatore di drift misurando le diverse metriche di rilevamento, è necessario sapere in anticipo dove e quando questi si verificano. Questo è però possibile solo utilizzando dei data set sintetici dove si conoscono esattamente gli istanti e le modalità con cui si verificano i drift nelle distribuzioni. A tal proposito sono

stati utilizzati dei data set generati dall'università di *Harvard*[13] contenenti drift improvvisi e graduali, descritti in sezione 2.2. Nello specifico, in tabella 3.4 sono riportati gli indici di inizio e fine del drift. Possiamo notare che per i drift improvvisi non abbiamo un id di fine in quanto la variazione della distribuzione avviene, appunto, improvvisamente (si passa da un concept all'altro bruscamente), mentre i drift graduali impiegano 1.000 istanze prima di passare completamente da un concept all'altro.

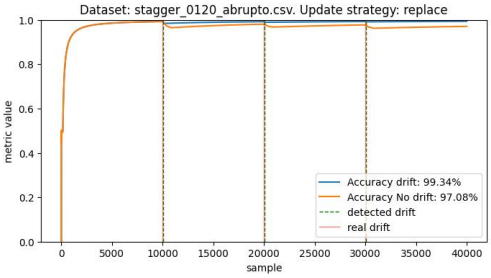
I modelli sono stati testati su 20 data set. Di seguito vengono riportati alcuni degli esempi più significativi al fine di discutere il funzionamento degli algoritmi. Nello specifico nelle figure 3.2 e 3.3 vengono riportate le evoluzioni temporali delle performance dei vari modelli in data set contenenti concept drift rispettivamente **improvvisi** e **graduali**. Come si può notare dalle figure 3.2a e 3.2b, le strategie di push e replace contengono intrinsecamente un meccanismo in grado di adattarsi al drift (in quanto i dati più vecchi vengono dimenticati automaticamente sostituendo i weak learners) per cui ottengono risultati molto buoni anche senza l'utilizzo di un meccanismo per il rilevamento del drift. Questo non si verifica nel caso delle strategie proposte (figure 3.2c e 3.2d) in cui il degrado delle performance nelle fasi successive al verificarsi del drift è molto evidente. Da queste figure possiamo notare come, grazie ad *ADWIN*, il drift venga rilevato tempestivamente e come l'adattamento dei modelli porti ad un rapido miglioramento nelle performance.

D'altro canto possiamo notare come in alcuni casi (figure 3.3a e 3.3b) le strategie di push e replace possano essere meno robuste nelle fasi più stabili delle distribuzioni, portando quindi al verificarsi di falsi positivi (vengono rilevati dei drift inesistenti). Questo è meno frequente nel caso delle strategie di update (figure 3.3c e 3.3d) portandole ad ottenere delle performance leggermente migliori delle precedenti.

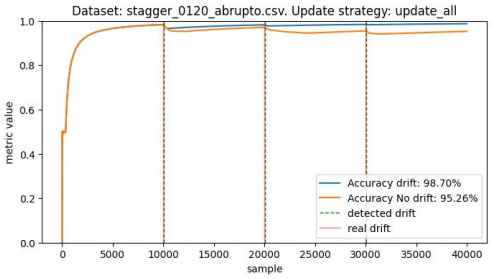
[13] https://bit.ly/concept-drift-dataset



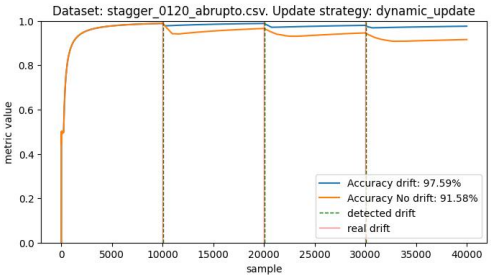
(a) Push



(b) Replace

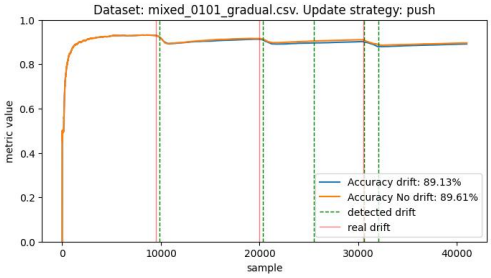


(c) Update statico

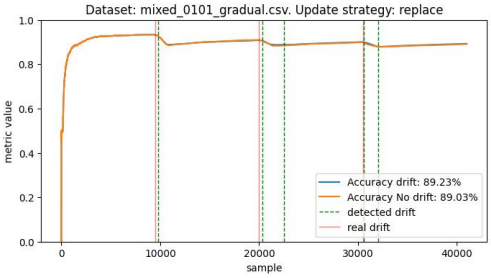


(d) Update dinamico

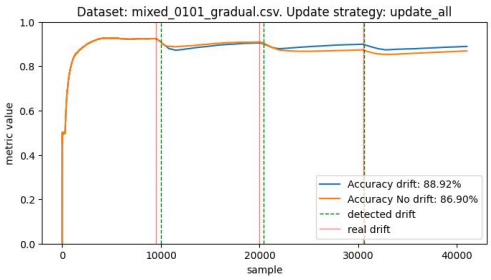
Figura 3.2: Esempio di addestramento con concept drift **improvviso**



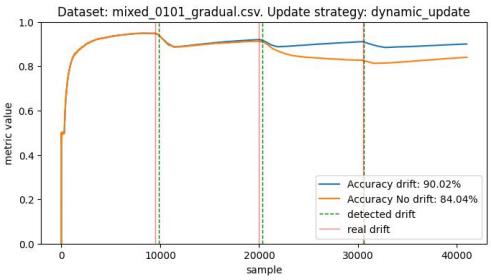
(a) Push



(b) Replace



(c) Update statico



(d) Update dinamico

Figura 3.3: Esempio di addestramento con concept drift **graduale**

Dataset	Strategia	Accuratezza%	F1-score%
Bananas	Replace	86,26	86,00
	Push	84,42	84,22
	Update Dinamico	**87,85**	**87,60**
	Update Statico	87,70	87,45
Phishing	Replace	90,16	90,08
	Push	89,28	89,13
	Update Dinamico	**91,04**	**90,93**
	Update Statico	90,16	89,95
Image Segments	Replace	89,74	89,79
	Push	89,25	89,18
	Update Dinamico	**91,58**	**91,40**
	Update Statico	91,17	91,18
Insects	Replace	54,84	54,51
	Push	42,39	42,62
	Update Dinamico	**60,49**	**59,95**
	Update Statico	59,29	58,85
Keystroke	Replace	64,64	64,78
	Push	52,86	54,28
	Update Dinamico	80,79	80,82
	Update Statico	**83,21**	**83,21**

Tabella 3.2: Risultati di AXGB sui benchmarks

Dataset	Modello	Accuratezza%	F1-score%
Bananas	AXGBd	**87.85**	**87.60**
	HdT	64.22	50.34
	EFDT	62.52	45.14
	ARF	87.70	87.45
	ADAB	67.79	64.50
Phishing	AXGBd	**91.04**	**90.93**
	HdT	87.99	86.06
	EFDT	88.79	87.34
	ARF	90.87	89.69
	ADAB	87.83	86.36
Image segments	AXGBd	**91.58**	**91.40**
	HdT	77.61	76.31
	EFDT	62.54	63.26
	ARF	81.85	81.41
	ADAB	80.47	79.78
Insects	AXGBd	60.49	59.95
	HdT	53.73	52.74
	EFDT	65.26	65.09
	ARF	**74.66**	**74.43**
	ADAB	56.35	55.46
Keystroke	AXGBs	83.21	83.21
	HdT	64.82	64.72
	EFDT	85.63	85.61
	ARF	**96.92**	**96.92**
	ADAB	84.16	84.31

Tabella 3.3: Comparazione modelli online sui benchmarks

Tipologia di drift	Drift I		Drift II		Drift III	
	ID Inizio	ID Fine	ID Inizio	ID Fine	ID Inizio	ID Fine
Improvviso	10.000	-	20.000	-	30.000	-
Graduale	9.500	10.500	20.000	21.000	30.500	31.500

Tabella 3.4: Evoluzione dei drift nei dati

Capitolo 4

Classificazione di anomalie

In questo capitolo viene approfondito un caso di applicazione reale a cui sono stati applicati gli algoritmi di online machine learning descritti in precedenza. Nello specifico, il task che viene presentato riguarda la classificazione di anomalie (**classificazione binaria**) in macchinari industriali, basata su letture sensoriali di un macchinario specifico dell'industria chimica.

In primo luogo andremo ad effettuare un'analisi esplorativa al fine di comprendere meglio i dati di cui si dispone, per poi valutare le performance dei modelli su questo data set.

Tutte le analisi riportate di seguito sono state effettuate utilizzando il linguaggio di programmazione *Python* e le più famose librerie di data science: *Pandas*[1] per la gestione di dati tabulati, *Numpy*[2] per la gestione ottimizzata del calcolo scientifico, *MatplotLib*[3] per la visualizzazione.

4.1 Analisi esplorativa

In questo caso d'uso, i dati che si hanno a disposizione provengono, come già anticipato, da letture sensoriali facenti riferimento ad uno specifico processo chimico, per cui i sensori (contrassegnati da un indice univoco) rappresentano letture fisi-

[1] https://pandas.pydata.org/
[2] https://numpy.org/
[3] https://matplotlib.org/

che di *pressione*, *temperatura*, *umidità*, etc. Nello specifico abbiamo a disposizione dati provenienti da **sette** sensori diversi provenienti da uno stesso macchinario in funzione per circa **sei mesi**. Nella tabella 4.1 viene riportato un riassunto della composizione del data set con una breve descrizione delle caratteristiche di cui esso è composto. Possiamo notare come il data set, composto da circa 4.900.000 istanze, non presenti alcun valore nullo.

Colonna	Descrizione	Istanze
timestamp	data di arrivo dell'istanza (unix timestamp)	4.903.795
value	lettura del sensore (dato raw)	4.903.795
quantity	ID sensore (da 0 a 6)	4.903.795
state	stato del processo chimico (ON-OFF)	4.903.795

Tabella 4.1: Descrizione data set

In tabella 4.2 vengono riassunte le letture dei sensori disponibili. Come possiamo notare, le letture hanno frequenze di campionamento molto diverse, in particolare possiamo vedere come il sensore 3, relativo ad un sensore di pressione, sia il più presente all'interno del data set, mentre il meno presente (con una frequenza media di campionamento di circa tre ore) è invece il sensore con id 2.

ID Sensore	Istanze	Data Inizio	Data Fine	Frequenza
0	53.044	01-15 00:13:34	06-04 14:10:43	00:03:49.0254
1	15.971	01-15 00:04:33	06-04 14:20:52	00:12:40.6111
2	**1.061**	01-15 00:53:21	06-04 13:00:59	**03:10:48.9789**
3	**4.639.957**	01-15 00:00:01	06-04 14:28:20	**00:00:02.6181**
4	83.318	01-15 00:00:26	06-04 14:28:54	00:02:25.8049
5	77.343	01-15 07:06:53	06-04 14:26:52	00:02:36.7393
6	33.101	01-15 00:17:07	06-04 14:26:10	00:06:06.9693

Tabella 4.2: Istanze letture

In figura 4.1 vengono sintetizzati gli orari di inizio/fine del processo di lavorazione in esame. Si nota come le macchine vengano messe in funzione intorno alle

ore 9 e come il processo termini mediamente in orari notturni (prevalentemente intorno alla mezzanotte). È inoltre stato calcolato che la durata media di un processo chimico di questo tipo è di circa 12 ore e 46 minuti con una deviazione standard di 2 ore e 43 minuti ($12:46 \pm 02:43$), mentre i macchinari restano spenti per circa 12 ore e 27 minuti con una deviazione standard di 5 ore e 12 minuti circa ($12:27 \pm 05:12$).

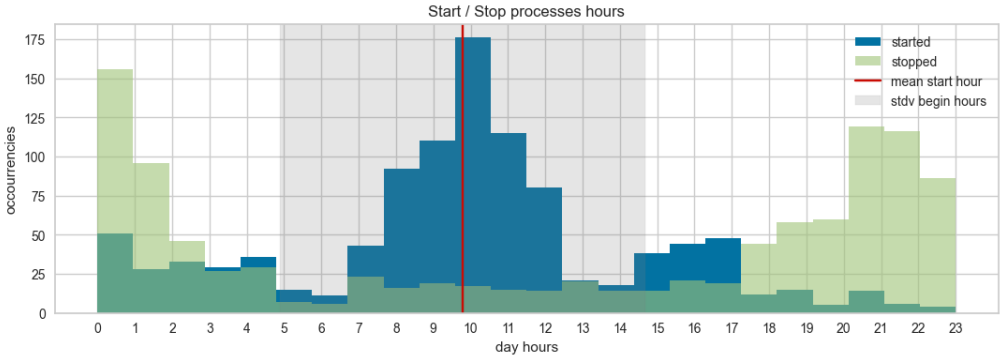


Figura 4.1: Inizio/Fine processo chimico

4.2 Feature engineering

Dopo un'analisi preliminare dei dati a disposizione, risulta necessario definire i target (etichette) che saranno utilizzati in fase di addestramento, al fine di effettuare una fase di processamento dei dati il cui output sarà un data set *model-ready* che potrà essere utilizzato per addestrare i modelli decisionali.

A tal proposito è stato valutato il sensore di pressione avente il maggior numero di istanze nel data set ($ID = 3$). In figura 4.2 viene riportata l'evoluzione temporale del sensore, la cui lettura esprime valori di pressione in *millibar*. Dalla figura 4.2 possiamo notare come la ripetitività nelle letture stia a rappresentare lo stesso processo di lavorazione in periodi temporali diversi. In particolare, nelle casistiche in cui la lettura si aggira intorno ai $1.000\, m\, bar$ (pressione atmosferica), il macchinario in esame è spento, mentre nelle fasi di lavorazione i valori nelle letture salgono per poi scendere bruscamente nella fase finale del processo.

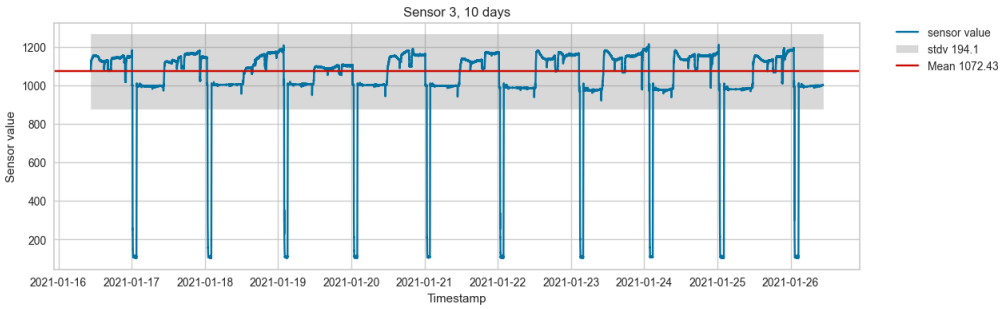


Figura 4.2: Evoluzione temporale sensore 3

Nella definizione delle anomalie sono state considerate unicamente le letture in cui il processo chimico è in atto ($state = ON$). Va precisato che, in un caso ideale, la curva temporale del sensore dovrebbe restare costante e superiore ad una data soglia durante tutto il processo di lavorazione, per cui sono state classificate come anomale le istanze in cui la pressione del sensore subisce un abbassamento inaspettato.

Nel processo di definizione delle istanze anomale è stata utilizzata una *sliding window* valutando il valore medio delle letture all'interno della finestra ed il valore della deviazione standard. La dimensione della finestra utilizzata è stata selezionata sperimentalmente, effettuando esperimenti e valutando visivamente l'output dell'algoritmo. La soglia impostata sul valore medio effettua una prima scrematura delle istanze non anomale selezionando i valori nell'intorno dei picchi, mentre, con il fine ultimo di voler selezionare anche le istanze in cui la pressione inizia a scendere, è stata definita una soglia per il valore di deviazione standard all'interno della finestra.

In figura 4.3 viene mostrato come l'applicazione della metodologia sopra descritta sia in grado di rilevare correttamente le istanze anomale nelle fasi del processo. Va inoltre precisato che è stata impostata una soglia massima nel valore medio della *sliding window* al fine di non etichettare come anomale le istanze facenti riferimento all'ultimo periodo della lavorazione in cui il valore di pressione deve scendere bruscamente.

A questo punto, etichettate le istanze anomale e non anomale, il passaggio finale, atto alla costruzione di un data set pronto per essere utilizzato nell'adde-

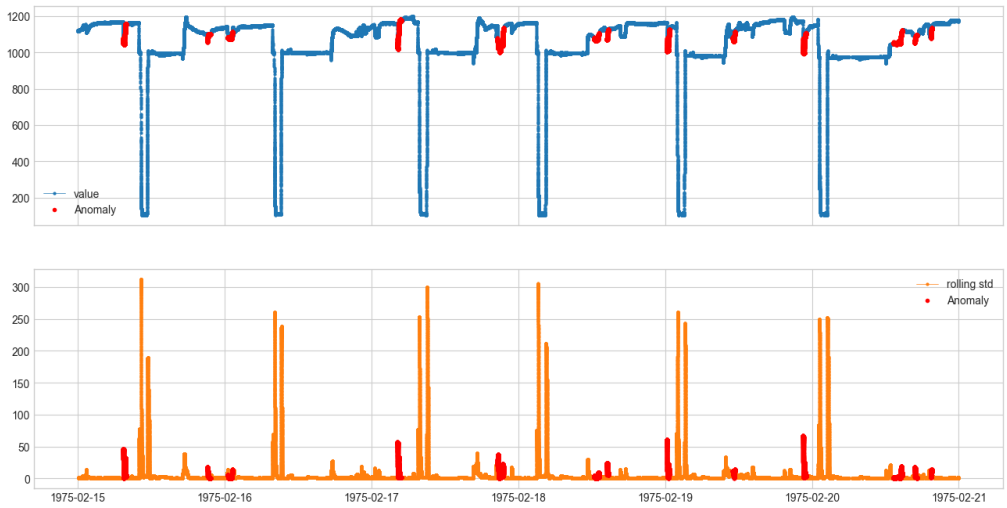


Figura 4.3: Feature engineering: definizione delle anomalie

stramento di un modello, è quello di effettuare una operazione di pivot. Utilizzando i dati nella tabella originale è stata generata una tabella, di dimensionalità diverse dalla precedente, contenente le letture di ogni sensore per ogni istanza del nuovo data set, oltre all'etichetta di classificazione. L'ottenimento di questa nuova tabella priva di valori nulli è stato raggiunto tramite un modello *zero hold*: per ogni istanza sensore viene mantenuto l'ultimo valore osservato fino al presentarsi di una sua nuova lettura. Questo è necessario in quanto le etichette sono assegnate unicamente alle istanze relative alla lettura del sensore 3 ed il data set originale presenta due colonne relative all'Id del sensore ed alla sua lettura. Quello di cui si necessita, invece, è un data set contenente:

- Sensori: una colonna per ogni sensore contenente le sue letture (da 0 a 6),
- Stato: macchinario acceso o spento (già presente).
- Etichetta: istanza anomala o no, definite tramite l'algoritmo descritto in precedenza.

Effettuato il pivoting, l'output finale della feature engineering è un data set contenente le colonne sopra elencate. Il data set risultante è in ogni caso altamente sbilanciato. Nello specifico, per un totale di 4.903.795 istanze, solo il 14% circa dei dati è etichettato come anomalo. Questo presenta un problema

nella fase di addestramento dei modelli in quanto essi potrebbero essere portati a predire molte più istanze come non anomale. A tal proposito risulta inefficace valutare l'accuratezza dei modelli in quanto questa metrica non tiene conto dello sbilanciamento nelle etichette, per cui sono state valutate le metriche di precisione, recall e score-f1 che verranno descritte successivamente.

4.3 Addestramento dei modelli

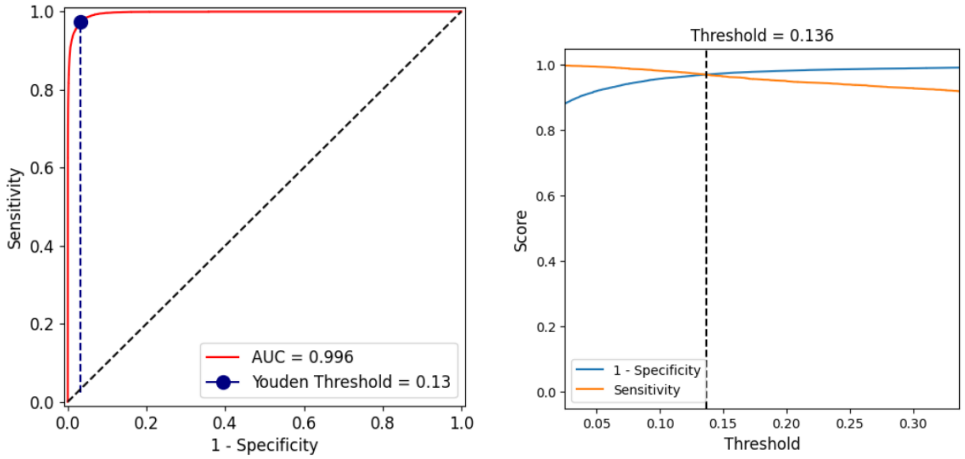
In questa sezione viene utilizzato il data set ottenuto dalla fase di *feature engineering*, descritta in sezione 4.2, per effettuare l'addestramento dei modelli di apprendimento online, al fine di effettuare una valutazione preliminare degli algoritmi.

4.3.1 Addestramento batch

In primo luogo è stato effettuato un addestramento di un modello XGBoost batch al fine di ottenere una baseline di riferimento riguardo alle performance sul data set che si aveva a disposizione.
Osservando la documentazione di XGBoost[4], questo algoritmo permette all'utilizzatore la definizione di molti iper-parametri. Utilizzando la tecnica della *k-fold cross validation* [Anguita et al., 2012], con $k = 5$ in questo caso, sono stati ottimizzati gli *iper-parametri* del modello in esame utilizzando il framework **optuna** [Akiba et al., 2019] massimizzando il valore della metrica **F1** mediata sui diversi data set utilizzati nella fase di validazione.
Inoltre, è stato utilizzato l'*indice di Youden* [Schisterman et al., 2008] al fine di calcolare un valore ottimo per la soglia di classificazione che massimizzi i valori di *specificità* (capacità del modello di classificare correttamente le istanze negative/non anomale) e *sensitività* (capacità del modello di classificare correttamente le istanze positive/anomale). In figura 4.4 si vede come al variare della soglia utilizzata per la classificazione, varino anche i valori di sensitività e specificità del modello. Nello specifico in figura 4.4b si mostra come il valore ottimo della

[4] https://xgboost.readthedocs.io/en/stable/

soglia corrisponda al punto di intersezione delle due curve, mentre in figura 4.4a viene mostrato il punto sulla curva *ROC* [Gneiting et al., 2019] corrispondente al valore di soglia ottenuto.



(a) Curva roc e Youden threshold

(b) Specificità e Sensitività

Figura 4.4: Soglia di Youden

Utilizzando quindi gli iper-parametri e la soglia ottenuti dal passaggio precedente, in figura 4.5 viene riportata la matrice di confusione ottenuta sul test set, mentre in figura 4.6 vengono riportate le metriche ottenute nelle fasi di addestramento, validazione e test per le istanze anomale (1) e non (0).
I risultati sono soddisfacenti, seppur il modello generi dei falsi positivi/negativi, risultati riconducibili allo sbilanciamento del data set.

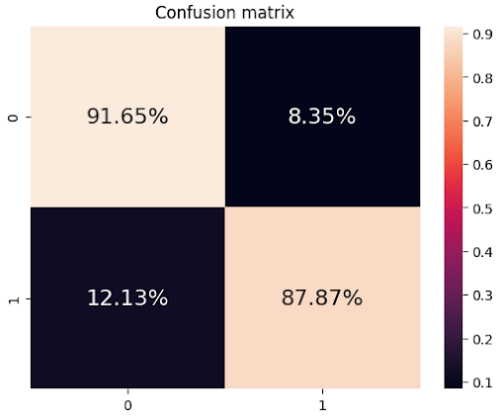


Figura 4.5: Matrice di confusione

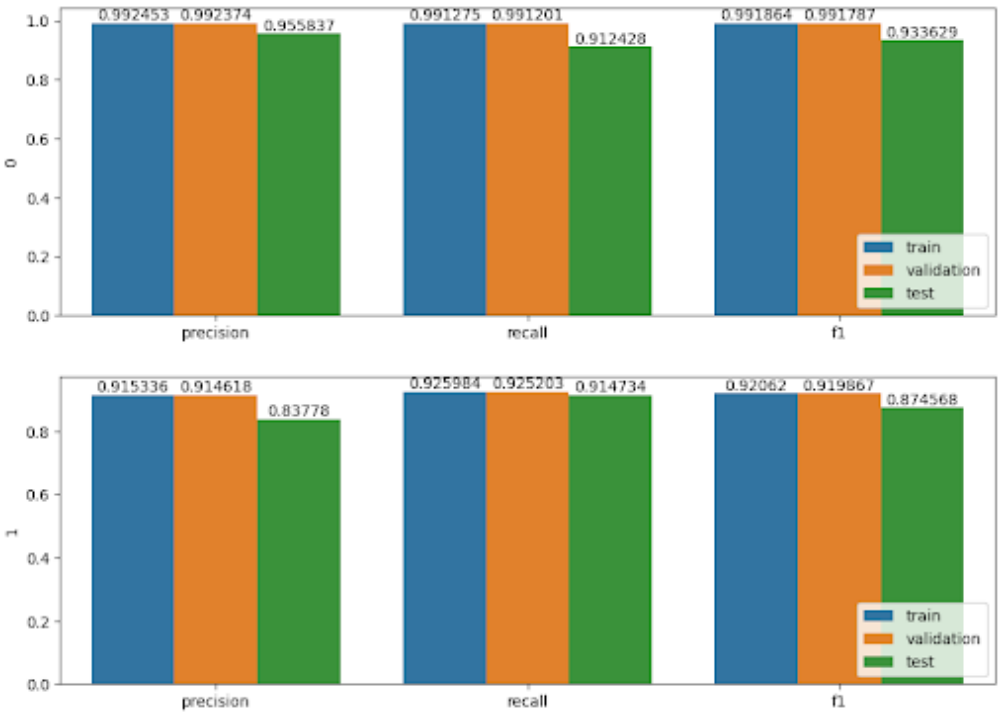


Figura 4.6: Metriche di addestramento

Infine sono stati utilizzati i metodi introdotti da *XAI* (*eXplainable Artificial Intelligence* [Barredo Arrieta et al., 2020]) con l'obiettivo di interpretare gli output generati dal modello di apprendimento. Nello specifico è stata utilizzata la metodologia **SHAP** [Lundberg and Lee, 2017] implementata dall'omonimo framework[5] per python.

La figura 4.7 mostra come l'output del modello sia fortemente condizionato da valori alti del valore *state* e da valori intermedi dell'attributo inerente le letture del sensore 3, in completo accordo con quanto ci aspettiamo. Infatti, nei casi in cui il macchinario è spento e le lavorazioni non sono in atto (caso *state*= 0) non ha senso classificare le istanze come anomale, per cui l'output del modello sarà fortemente spinto verso valori negativi e quindi verso una classificazione dell'istanza come non anomala. L'output è fortemente condizionato anche dai valori delle letture del sensore con *ID* 3 in quanto l'intera generazione delle istanze anomale si è basata su questo attributo. In particolare vediamo che per valori medi nella

[5] https://shap.readthedocs.io/en/latest/

lettura del sensore, l'output del modello sarà molto positivo, portando quindi ad una classificazione dell'istanza come anomala. Questo risulta perfettamente in linea con quanto ci aspettiamo: le istanze anomale si verificano quando la lettura di questo sensore è di poco inferiore ad una certa soglia.

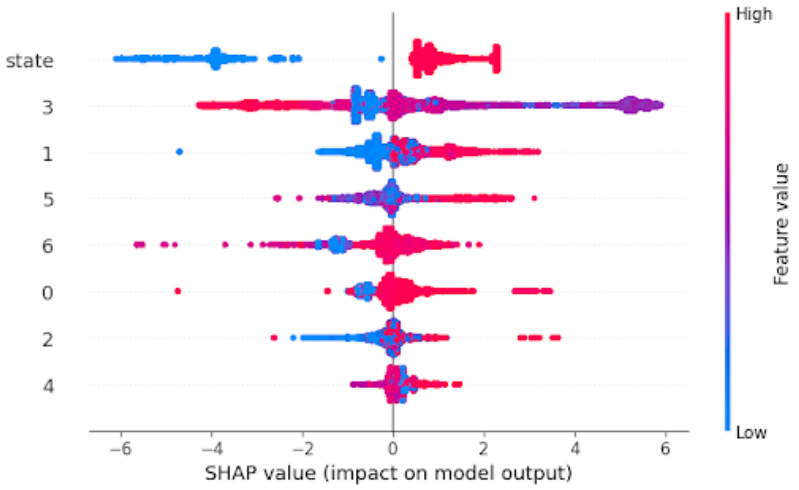


Figura 4.7: Model explainability, SHAP per XGBoost

4.3.2 Addestramento online

A questo punto si è passato all'addestramento dei modelli online. In particolare sono stati addestrati gli stessi modelli della libreria river che sono stai utilizzati nella sezione 3, mentre per quanto riguarda AXGB è stata valutata una strategia tra le vecchie push e replace, ed una tra quelle proposte in questo lavoro di tesi (selezionando quelle con risultati migliori). La definizione degli iper-parametri dei modelli è stata in questo caso effettuata ottimizzando il valore dello score F1 valutato su un test set costituito da un sotto insieme bilanciato di dati appartenente all'ultima settimana. Successivamente il calcolo delle metriche di performance è stato effettuato sulla falsa riga della metodologia proposta da River nel calcolo dei benchmarks, metodologia riassunta dall'algoritmo 3: dati in input il modello, gli attributi e le etichette, per ogni dato si effettua la previsione prima che il modello possa apprendere dal dato, si aggiornano le metriche e solo alla fine si aggiorna il modello permettendogli di apprendere dall'ultima istanza.

Algoritmo 3 Addestramento online

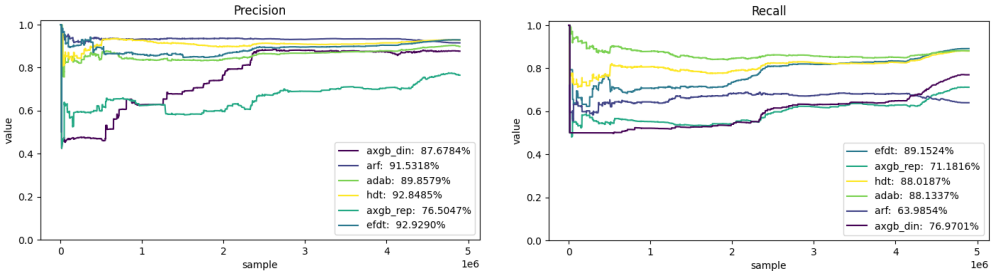
Input: $Model, Attributi, Etichette$

1: inizializza la metrica
2: **for all** $x, y \in (Attributi,\ Etichette)$ **do**
3: $\quad \hat{y} = Model(x)$
4: $\quad$ aggiorna_metrica(y, $\hat{y}$)
5: $\quad$ aggiorna_modello($Model$, x, y)
6: **end for**

Definiti gli iper-parametri di ogni modello, sono stati effettuati l'addestramento e la valutazione delle metriche di *precisione*, *recall*, *score-F1* ed infine è stato valutato l'*errore di classificazione*. I risultati degli addestramenti vengono riassunti in figura 4.8. In particolare possiamo notare come per quanto riguarda gli algoritmi XGBoost Adattivi, presentino una convergenza più lenta rispetto agli altri. Questo accade perché l'algoritmo rientra a far parte nella famiglia di algoritmi di apprendimento ad istanza multipla. Per questo specifico use case la problematica della definizione della dimensione ottima della finestra di addestramento risulta essere molto accentuata, portando a delle metriche di classificazione che scendono bruscamente nella prima fase di addestramento. Questo fenomeno è ulteriormente accentuato nella strategia di replace in quanto il modello dimentica costantemente i dati visionati in precedenza.

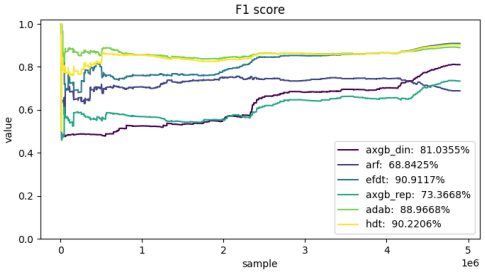
In tabella 4.3 vengono riassunte le metriche ottenute dai vari modelli. Si nota come gli algoritmi di apprendimento a singola istanza risultano essere migliori nelle performance predittive in questo use case specifico, a conferma di quanto riportato nelle figure di cui sopra.

Infine va menzionato il tentativo di addestramento di modelli deep per dati streaming. Il *Deep Learning* è una branca dell'intelligenza artificiale che si concentra sull'utilizzo di reti neurali artificiali al fine di risolvere problemi complessi. Il funzionamento delle reti neurali è ispirato al funzionamento del cervello umano e sono in grado di imparare da esempi di addestramento forniti. Queste metodologie hanno avuto molto successo in molte applicazioni, come la *computer vision*, il riconoscimento del linguaggio naturale e la generazione di testo. In questo caso d'uso particolare è stato utilizzato un algoritmo chiamato *Hedge Backpropaga-*

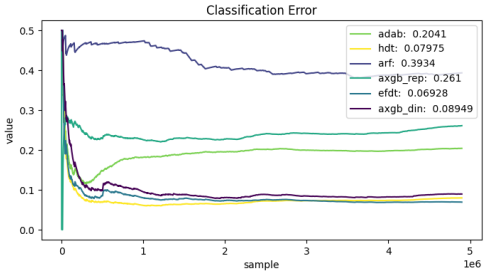


(a) Precision

(b) Recall



(c) F1-score



(d) Classification Error

Figura 4.8: Addestramento modelli online

tion [Sahoo et al., 2017] il cui funzionamento si basa sul sovradimensionamento iniziale della rete neurale, con consecutivo adattamento di pesi associati ad ogni *layer* della rete.

Tuttavia, l'elevata numerosità dei dati rende ostico l'addestramento di questo tipo di modelli, che in ogni caso sembrano avere delle performance, sia in termini computazionali che di classificazione, ridotte in fase di testing del modello finale (sono molto soggetti ad *overfitting* per questo specifico caso d'uso). Questi risultati sono anche confermati da [Grinsztajn et al., 2022] in cui viene mostrato come per dati tabulati i classici algoritmi con strutture ad albero "performino" meglio di algoritmi deep. Per questi motivi si è proseguito con lo studio degli algoritmi sopra descritti.

Model	Accuracy %	Precision %	Recall %	F1-score %
AXGBu	93.28	89.94	80.78	84.50
AXGBr	90.77	81.85	78.51	80.04
HdT	91.42	81.21	**89.93**	84.63
ARF	89.34	85.00	65.94	70.53
ADAB	94.45	89.87	86.62	88.14
EFDT	**94.50**	**90.10**	86.55	**88.20**

Tabella 4.3: Metriche di classificazione online

Capitolo 5

Edge computing

Il *cloud computing* è una tecnologia che consente di accedere a risorse informatiche, come calcolo, archiviazione ed applicazioni, in modo flessibile e scalabile tramite la rete. Tuttavia, ci sono alcune problematiche legate all'utilizzo del cloud, come la latenza nell'elaborazione dei dati, la sicurezza dei dati e la dipendenza dalla disponibilità della connessione a internet. Inoltre, con la costante evoluzione tecnologica dei dispositivi collegati alla rete ed al loro impiego in settori specializzati, viene a crearsi una nuova classe di applicazioni che richiedono caratteristiche stringenti per quanto riguarda le modalità di interazione tra la sorgente di produzione dei dati ed il data center che effettivamente li elabora. Le specifiche richieste spesso non possono essere garantite lungo la rete che porta i dati dai dispositivi al cloud a causa della loro *distanza fisica* ed alla non sempre scontata connessione ad internet dei dispositivi.

A tal proposito, un nuovo paradigma di calcolo è in rapido sviluppo in molti settori di applicazione, l'**edge-computing**. L'edge computing consente di elaborare i dati in modo distribuito utilizzando dispositivi di calcolo collocati presso la "periferia" della rete, vicini ai dispositivi che generano i dati. Ciò consente di ridurre la quantità di dati che devono essere trasferiti verso i centri di elaborazione centralizzati del cloud, migliorando la latenza e aumentando la sicurezza dei dati. L'edge computing è particolarmente utile per le applicazioni che richiedono una rapida elaborazione dei dati in tempo reale e che non possono essere gestite

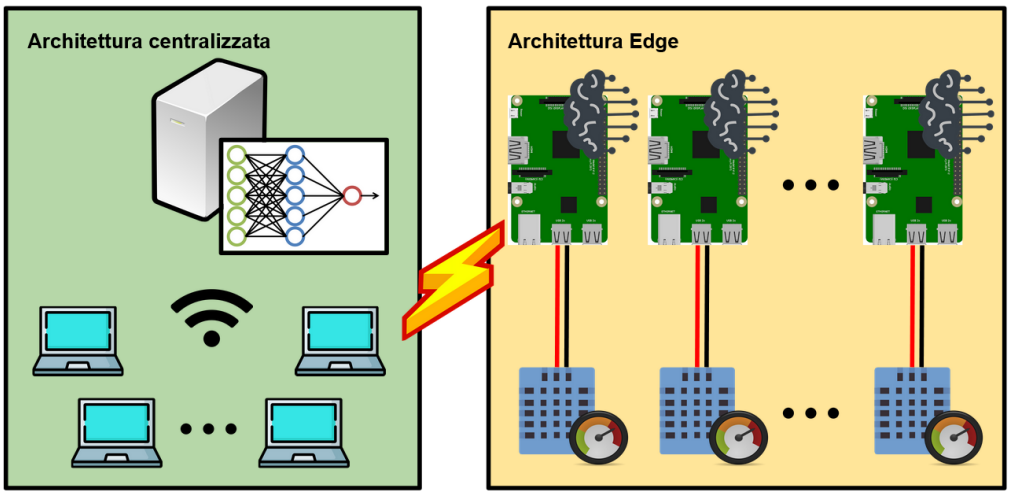


Figura 5.1: Architetture: Centralizzata ed Edge

efficacemente dal cloud, come la guida autonoma dei veicoli, la realtà aumentata e la videosorveglianza.

In questo capitolo vengono esplorate le tecnologie per il calcolo all'edge, con il fine ultimo di implementare le architetture riportate in figura 5.1, per poi effettuare una comparazione dei modelli di online machine learning nella loro versione centralizzata ed edge.

5.1 Container runtime: Docker

Un *container runtime* è un software in grado di eseguire dei "pacchetti" di applicazioni contenenti le dipendenze di cui essi necessitano; pacchetti che vengono appunto chiamati **container**. Ciò consente di isolare le varie applicazioni tra di loro e di distribuirle in modo più efficiente, poiché i container non hanno bisogno di un ambiente di esecuzione specifico per funzionare. In generale, l'utilizzo dei container runtime consente una maggiore flessibilità, portabilità e scalabilità delle applicazioni, rendendole più facili da distribuire e gestire sia in ambienti locali che in cloud.

Ci sono diversi container runtime, tra cui *CRI-O*[1] e *containerd*[2]. L'ecosi-

[1] https://cri-o.io/
[2] https://containerd.io/

stema *Docker* [Merkel, 2014] (che utilizza *containerd* e *runc*[3] nella gestione dei container) risulta essere la tecnologia più popolare e largamente utilizzata.

Nello specifico, Docker utilizza una architettura *client-server* attraverso il quale le sue componenti sono in grado di interagire, tramite delle *API rest*, con il fine ultimo di generare ed eseguire i container nella macchina host su cui è installato.

I principali oggetti su cui è basato il funzionamento sono:

- **Immagini**: componenti in sola lettura che forniscono le istruzioni per la generazione dei container. Generalmente una imagine Docker si basa su una o più immagini già esistenti, e sulla consecutiva personalizzazione.
- **Container**: definito come una istanza di una immagine. Gode di un isolamento le cui caratteristiche vengono definite sia dalla sua immagine, sia in fase di creazione.

I componenti principali che permettono a Docker di effettuare le operazioni di creazione ed eliminazione dei container:

- **Docker Daemon (dockerd)**: è il componente principale e si occupa della gestione delle risorse, immagini, volumi, container, porte e rete in generale.
- **Docker Client**: è l'interfaccia tra l'utente e dockerd. Permette agli utenti di interagire con il demone di sopra al fine di generare e gestire i container e le risorse.
- **Docker Registries**: è il componente che contiene le immagini per la generazione dei container. Un esempio è Docker Hub[4], una piattaforma gratuita che offre la possibilità di scaricare immagini pronte, oltre che di condividere e salvare le proprie.

In figura 5.2 viene sintetizzata l'architettura docker descritta in precedenza.

[3] https://github.com/opencontainers/runc
[4] https://hub.docker.com/

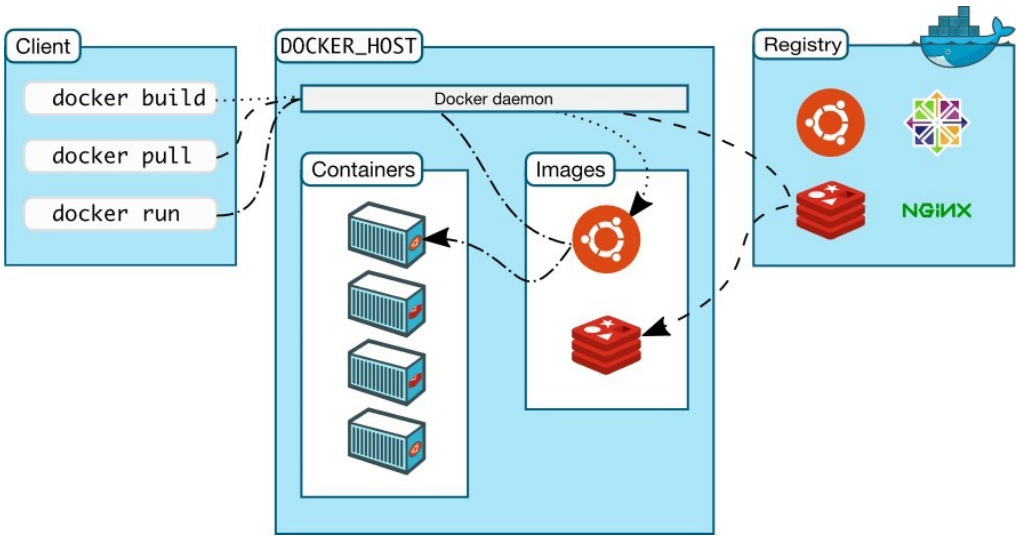


Figura 5.2: Architettura Docker

5.2 Orchestrazione di servizi: Kubernetes

Kubernetes [Rensin, 2015] è una piattaforma portabile ed estensibile per la gestione di carichi di lavoro e servizi in **container**. La piattaforma è open-source ed è stata creata da Google nel 2014. Orchestra l'infrastruttura informatica, di rete e di archiviazione per conto degli utenti dei carichi di lavoro ed è stato progettato anche come piattaforma per la creazione di un ecosistema di componenti e strumenti per semplificare l'implementazione, il ridimensionamento e la gestione delle applicazioni. Inoltre, il piano di controllo (*control plane*) di Kubernetes è basato sulle stesse API disponibili per sviluppatori e utenti rendendo possibile quindi lo sviluppo di risorse custom e di controller.

Come per tutte le infrastrutture distribuite, un cluster Kubernetes deve necessariamente essere composto da un *nodo master*, responsabile della schedulazione dei servizi e dell'esposizione delle API, e può contenere uno o più *nodi worker* su cui vengono schedulati i servizi e si occupano quindi della parte computazionale delle diverse applicazioni.

In generale Kubernetes utilizza una *CRI* (*Container Runtime Interface*) per poter eseguire le applicazioni tramite container, per cui ogni nodo facente parte di un cluster Kubernetes necessita di un runtime (ad esempio quello fornito

dall'ecosistema Docker: containerd).

Prima di poter approfondire l'architettura di un cluster Kubernetes, risulta necessario familiarizzare con i suoi componenti principali, che costituiscono il meccanismo di funzionamento del sistema:

- **Pod**: rappresentano l'unità più piccola generabile. Kubernetes, a differenza di altre architetture, non schedula direttamente i container, bensì genera un *wrapper* (Pod), ad un livello di astrazione più alto, che contiene uno o più container. I Pod schedulati all'interno di un cluster sono in grado di comunicare agevolmente, grazie al modello *"flat"*[5] adottato nell'architettura di rete di Kubernetes, e possono essere *scalati* in numero in base alle necessità e a quanto definito dall'utente.

- **Deployment**:

  in generale i Pod non vengono mai creati direttamente dall'utente, bensì vengono gestiti ad un livello più alto di astrazione utilizzando i *Deployments*. Tramite questo componente si possono definire le specifiche inerenti i Pod che verranno eseguiti nel cluster, come ad esempio il numero di repliche o le immagini da utilizzare. I Pod generati verranno costantemente monitorati al fine di mantenere le specifiche definite nel deployment.

- **Service**:

  I Pod in esecuzione all'interno di un cluster sono raggiungibili direttamente tramite il loro indirizzo IP (*Internet Protocol*), questo però è disagevole in quanto l'indirizzo IP è variabile. I *Services* in Kubernetes hanno lo scopo di rendere disponibili i Pod all'interno del cluster senza conoscerne direttamente l'indirizzo ip. Nella creazione di un service è possibile definire una *risorsa Endpoint* (tramite l'utilizzo di appositi *selettori ed etichette*[6]) verso cui instradare le richieste effettuate ad un nome di dominio ed una rotta, definiti anch'essi nel service. Successivamente sarà Kubernetes stesso

[5] https://kubernetes.io/docs/concepts/services-networking/

[6] https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

ad instradare le richieste verso la risorsa *Endpoint* in base al loro carico di lavoro.

- **Ingress**:

  tramite l'utilizzo dei service, i Pod vengono resi disponibili all'interno del cluster, ma non è ancora possibile raggiungerli dall'esterno. Un *Ingress* è il componente che gestisce l'accesso esterno ai servizi interni al cluster.

- **Namespace**:

  Con il concetto di namespace è possibile fornire una segregazione logica delle risorse Kubernetes. Questo concetto è un astrazione del concetto più ampio di "Ambiente". Ad esempio quando si voglio dividere in gruppi le applicazioni o controllare la visione del cluster da parte degli utenti vengono usati i *namespaces*. In questo modo si riesce ad avere una separazione logica delle applicazioni.

Definiti i componenti principali attraverso il quale un utente può gestire un cluster Kubernetes, possiamo approfondire l'architettura stessa del framework. Per ulteriori informazioni e componenti Kubernetes si può far riferimento alla documentazione ufficiale[7].

In figura 5.3 viene mostrata l'architettura classica di un cluster Kubernetes. Facendo riferimento alla documentazione ufficiale, **control plane** è un insieme di componenti responsabili di tutte le decisioni globali sul cluster, oltre che al rilevamento e consecutiva esecuzione dei vari eventi cui un cluster può essere sottoposto (come ad esempio la creazione di un nuovo Pod). In generale i componenti del control plane possono essere eseguiti su qualsiasi nodo del cluster, ma per semplicità si tende ad eseguire tutti i componenti sulla stessa macchina, in genere il nodo **master**.
Volendo approfondire i componenti di cui il control plane è composto:

- **kube-apiserver**:

[7] https://kubernetes.io/docs/home/

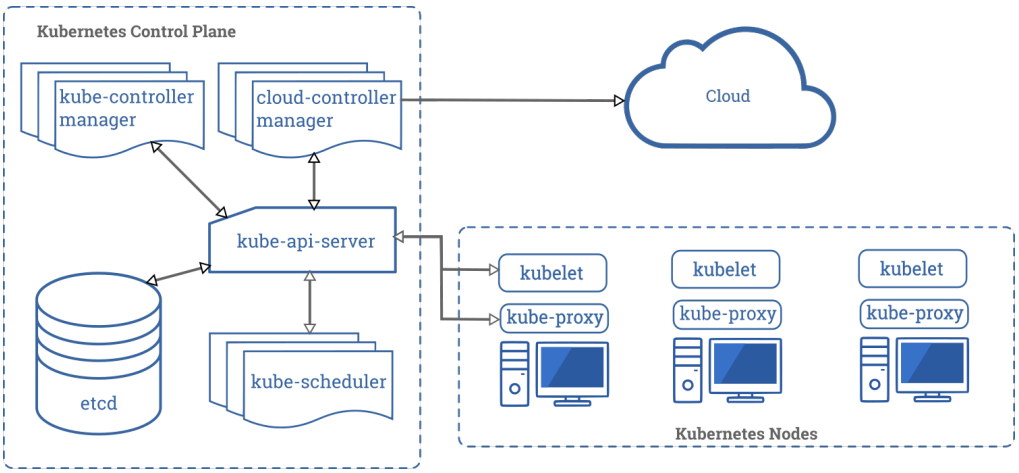


Figura 5.3: Componenti di Kubernetes

è il componente che espone le *API* (*Application Program Interface*) di Kubernetes attraverso il quale è possibile interagire con il cluster utilizzando un client (come ad esempio *kubectl*[8]).

- **etcd**:

  è un database di tipo *chiave-valore* in cui vengono salvate tutte le informazioni relative al cluster. È il componente che viene osservato al fine di mantenere costante le repliche dei Pod o, più in generale, quanto definito dagli utenti. In genere è buona norma mantenere un backup del database etcd al fine di essere in grado di effettuare un ripristino totale del cluster in caso di rotture.

- **kube-scheduler**:

  è il componente che osserva i Pod che sono stati **appena** creati e che non hanno ancora nessun nodo assegnato. Identificati i Pod, utilizza dei fattori, come la richiesta di risorse del Pod, i vincoli delle risorse hardware/software/policy e molti altri, al fine di identificare un nodo del cluster su cui eseguire il Pod.

- **kube-controller-manager**:

[8] https://kubernetes.io/docs/tasks/tools/

è il componente che gestisce i **controller**. I controller sono componenti software che si occupano, attraverso un "dialogo" con l'API server, di portare il sistema dallo stato attuale a quello desiderato. Alcuni esempi di controller possono essere: il *controller dei Nodi* che gestisce appunto i nodi nel cluster occupandosi, ad esempio, della gestione delle operazioni da effettuare quando un nodo non è più disponibile, il *controller delle Repliche* che, come suggerito dal nome, si occupa della gestione delle repliche dei Pod all'interno del cluster.

- **cloud-controller-manager**:

  è un componente che aggiunge delle logiche di controllo specifiche per un *cloud provider*, ossia l'ente che mette a disposizione le macchine che si utilizzano. Se si ha una istallazione *on-premise* (le macchine sono di nostra proprietà) il cloud controller manager non sarà presente.

Inoltre ogni nodo deve necessariamente eseguire dei componenti specifici atti alla comunicazione con l'api server di kubernetes. Nello specifico vengono eseguiti due *agenti*:

- **kubelet**:

  un agente che si assicura che i container siano eseguiti in un Pod. La *kubelet* riceve in input un insieme di specifiche per i Pod e le utilizza per controllare che i container descritti in queste specifiche siano "sani" e funzionino correttamente.

- **kube-proxy**:

  è un proxy (applicazione che gestisce il traffico di informazioni tra più terminali) che si occupa della gestione delle regole di networking sul nodo che lo ospita, permettendo la raggiungibilità dei vari *workloads* tra i nodi del cluster.

Inoltre, a partire dalla versione 1.7, Kubernetes introduce il concetto di **CRD** (*Custom Resource Definition*), delle risorse attraverso il quale è possibile definire

nuove API al fine di estendere le funzionalità di Kubernetes. Attraverso l'introduzione delle *CRD* gli utenti sono in grado di definire delle risorse proprie che specificano una nuova configurazione. Tuttavia non è sufficiente in quanto esse definiscono le configurazioni, ma non le gestiscono. A tal proposito sono stati introdotti, nella stessa versione, gli *Operatori*, cioè dei controller associati alle risorse custom.

Lo sviluppo di risorse custom e degli operatori specifici portano a delle evoluzioni del framework di tutti i tipi. In questo lavoro di tesi è stato sperimentato un ulteriore framework basato su questo tipo di risorse che analizzeremo nelle sezioni successive.

5.2.1 Un package manager per Kubernetes: Helm

Per poter effettuare il deploy di oggetti all'interno di Kubernetes è necessario scrivere file in formato YAML (o in formato JSON) ed "applicarli" dialogando con il kube-api-server. La scrittura di file per il deployments, replica sets e services può diventare un'attività complessa ed onerosa in termini di tempo. Proprio per questo in Kubernetes sono stati introdotti *package manager* per lo sviluppo di package che possono semplificare le attività di deploy all'interno del cluster. Helm [Spillner, 2019] è un package manager che consente di eseguire le attività di:

- recupero pacchetti software da repository,
- configurare il deploy del software come insieme di template, ognuno dei quali definisce una diversa risorsa Kubernetes, i cui placeholders vengonono risolti tramite i valori nel file *values.yaml*,
- installare il software e le sue dipendenze,
- effettuare l'aggiornamento dei software e delle loro dipendenze.

Per poter interagire con Helm vengono utilizzati pacchetti software, chiamati **charts**, che possono essere installati a partire da una directory locale oppure tramite un repository remoto.

Facendo riferimento alla documentazione ufficiale[9], un chart è definito come segue:

- **values.yaml**: contiene i valori della configurazione usati dal pacchetto
- **requirements.yaml**: contiene l'elenco delle dipendenze del pacchetto
- **Chart.yaml**: all'interno di questo file sono posizionati tutti i metadata necessari al chart, come il nome, la versione del pacchetto da implementare
- **charts**/: cartella contenente le dipedenze del chart, che possono essere utilizzate (o che non possono essere utilizzate), accessibili tramite una sorgente internet. All'interno del file requirements.yaml è possibile specificare le dipendenze da utlizzare
- **templates**/: cartella contenente i template che sono stati referenziati all'interno di values.yaml.

5.3 Kubernetes per l'edge-deployment: Kubeedge

Kubeedge[10] è un sistema open source che si pone l'obiettivo di creare una piattaforma aperta per sfruttare l'edge computing, estendendo le capacità di orchestrazione nativa delle applicazioni agli host Edge che si basano su Kubernetes. Kubeedge fornisce alle applicazioni un supporto di infrastruttura fondamentale per la rete, la distribuzione di app e la sincronizzazione dei metadati tra cloud e edge. Il progetto è stato recentemente incubato presso la *CNCF* (Cloud Native Computing Foundation), per cui è ragionevole attenderne rapide evoluzioni, con frequenti rilasci che maturino la tecnologia e integrino eventuali nuove estensioni, anche grazie alla costante crescita cui il tema dell'edge-computing è sottoposto.

La piattaforma è supportata dalle diverse tipologie di architetture di processori (*arm* e *x86*) ed è stata sviluppata per essere una estensione naturale di Kubernetes estendendo le sue funzionalità tramite la definizione di *CRD* ed i relativi *Operatori*.

[9] https://helm.sh/docs/
[10] https://Kubeedge.io/en/

Kubeedge sfrutta il protocollo di comunicazione *MQTT* (Message Queue Telemetry Transport), standard ISO che poggia sullo stack TCP/IP e si basa sul pattern publish-subscribe, progettato per gli scenari d'impiego in cui è richiesto un basso overhead (es., basso impatto computazionale dovuto al protocollo) e dove il throughput (larghezza di banda) utile alle comunicazioni può essere limitato, esattamente come può avvenire nel caso di device IoT. Dato il pattern publish-subscribe implementato da MQTT, Kubeedge richiede la presenza di un message broker, *Mosquitto* [Light, 2017] in questo caso, nei propri nodi di computing posti all'Edge, un servizio software aggiuntivo responsabile della ricezione/distribuzione dei messaggi applicativi provenienti dai device IoT che rilevano le metriche di interesse.

Nello specifico Kubeedge implementa risorse custom, disponibili a livello cloud, che vengono automaticamente aggiornate dalla piattaforma edge di riferimento, tramite protocollo *Web-Socket* [Fette and Melnikov, 2011]. In questo modo se la connettività dei nodi edge dovesse venire a mancare, l'aggiornamento sarebbe interrotto per poi riprendere quando possibile in maniera automatica, ma in ogni caso lato cloud si avrebbero sempre a disposizione delle informazioni, anche in mancanza di connettività.
L'architettura utilizzata da Kubeedge per raggiungere questo scopo è riassunta in figura 5.4.

Come mostrato in figura 5.4, risulta necessaria l'implementazione di due diversi strati software: quello inerente il *cloud* e quello inerente l'*edge*.

Specificatamente, per la parte cloud (sovrastante) vengono menzionati:

- **K8S api server**: esattamente l'api server kubernetes descritto in sezione 5.2

- **EdgeController**:

  gestisce i metadati dei nodi di computing all'Edge e dei Pod su essi eseguiti, con l'obiettivo di selezionare lo specifico nodo di computing cui indirizzare dati. Esso assolve la funzione di bridging tra l'api Server e l'EdgeCore in esecuzione su ciascun nodo Edge.

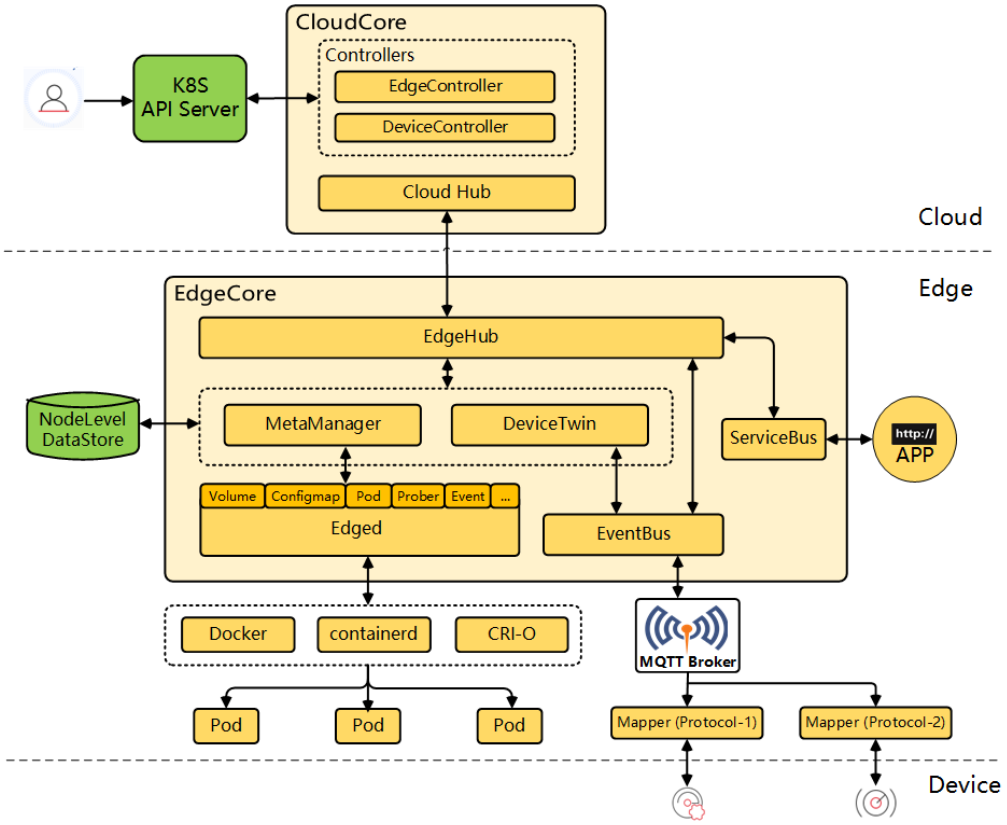


Figura 5.4: Architettura Kubeedge

- **DeviceController**:

  modulo responsabile della gestione degli end-device, implementata utilizzando le CRD Kubernetes, per descrivere sia metadati e stato degli end-device, sia il controller che ne sincronizza gli aggiornamenti tra edge e Cloud. Per implementare la gestione dei dispositivi, fa uso delle risorse *DeviceModel*[11] (descrizione delle proprietà del dispositivo "edge" esposte ai lettori, rappresenta un template riutilizzabile che consente di creare e gestire molti dispositivi) e *DeviceInstance* (rappresenta un'istanza del DeviceModel che fa riferimento alle proprietà definite in quest'ultimo; contiene dati che cambiano dinamicamente, quali lo stato attuale riportato dal dispositivo e lo stato desiderato di una proprietà di esso).

- **CloudHub**:

  server WebSocket responsabile della comunicazione, presentazione e visualizzazione delle variazioni di stato sul Cloud oltre che del caching e dell'invio/ricezione dei messaggi di sincronizzazione alla/dalla controparte Edge.

Per quanto riguarda invece i componenti fondamentali che vengono installati ed eseguiti sui nodi Edge abbiamo:

- **MQTT Broker**:

  viene utilizzato per generare gli eventi di aggiornamento. In riferimento alla documentazione ufficiale[12], è possibile inviare dei messaggi a topic ben definiti che verranno utilizzati per effettuare l'aggiornamento delle risorse di tipo *DeviceInstance* precedentemente descritte.

- **DataStore**:

  DBMS (DataBase Management System) che offre la persistenza eventualmente richiesta ai dati applicativi memorizzati localmente al nodo Edge.

[11] https://Kubeedge.io/en/docs/developer/device_crd/

[12] https://Kubeedge.io/en/docs/developer/message_topics/

- **EdgeD**:

  agente in esecuzione sui nodi di computing all'Edge che gestisce il ciclo di vita dei Pod che incapsulano le relative applicazioni containerizzate e consente agli amministratori di sistema di distribuirne i workload nei nodi.

- **EdgeHub**:

  client WebSocket responsabile dell'interazione di EdgeCore con CloudCore, per cui della sincronizzazione degli aggiornamenti delle risorse tra Edge e Cloud.

- **DeviceTwin**:

  memorizza lo stato dei device e lo sincronizza con il Cloud.

In generale questa tecnologia, come tutte le altre, presenta vantaggi e svantaggi rispetto alle tecnologie classiche per il deployment di servizi su macchine host anche con scarse capacità computazionali. In primo luogo si tratta di una tecnologia completamente open-source di cui è possible visionare il codice[13] oltre che contribuire al suo sviluppo, ed è una tecnologia nativa kubernetes, che ne rende possibile in maniera più o meno agevole l'integrazione in cluster già definiti. Oltre a questo, la sua architettura è ottimizzata per essere eseguita in scenari con risorse limitate ed in generale su qualsiasi tipo di architettura hardware (ARM, x86, 32/64 bit). D'altro canto però la tecnologia è ancora molto giovane e la documentazione confusa. Questo rende la sua installazione e configurazione non semplici.

[13] https://github.com/Kubeedge/Kubeedge

Capitolo 6

Realizzazione delle architetture

Una volta approfondite tutte le tecnologie necessarie alla realizzazione delle architetture per effettuare un paragone tra modelli centralizzati ed edge, possiamo passare alla descrizione delle metodologie utilizzate nella loro implementazione. In questo capitolo vengono descritte la piattaforma cloud utilizzate, la realizzazione delle immagini Docker necessarie ed infine l'implementazione dell'helm chart per il deployment di entrambe le architetture.

6.1 Piattaforma cloud utilizzata: Google Cloud

Figura 6.1: Logo Google Cloud Platform

Google Cloud Platform (GCP, logo in figura 6.1) è la suite di servizi di cloud computing offerta dal colosso di Mountain View. Sebbene sia stata lanciata nel 2011, un anno dopo *Microsoft Azure* e ben cinque anni a distanza dalla nascita di *Amazon Web Services* (AWS), nel 2020 GCP è stata riconosciuta leader insieme ai suoi due competitor nel *Magic Quadrant di Gartner* per la categoria *Cloud*

Figura 6.2: Magic Quadrant 2022

Infrastructure and Platform Services e tutt'ora rimane una dei leader del mercato (figura 6.2).

I servizi di Google Cloud Platform sono suddivisi in alcuni macro segmenti, tra i quali: computing, archiviazione e database, big data e machine learning, networking, strumenti per sviluppatori ed identità e sicurezza. All'interno di ciascun ambito, le singole risorse possono essere rilasciate scegliendo una delle 24 aree geografiche che vedono la presenza dei data center di Google.

- **Computing**: Per quanto riguarda il "calcolo", i servizi di GCP prevedono diverse tipologie che vanno dallo IaaS (Infrastructure as a Service) di Compute Engine, che propone macchine virtuali, al PaaS (Platform as a Service) di App Engine che fornisce hosting di applicazioni a elevata scalabilità, da Kubernetes Engine con cui gestire i cluster del noto orchestratore open source al FaaS (Function as a Service) di Cloud Function che funge da ambiente serverless con cui eseguire il codice senza gestione del server e con

cui connettere servizi cloud anche di terze parti. A perfezionare l'offerta di GCP lato computing c'è anche Cloud Run, piattaforma completamente gestita per eseguire il deployment e scalare applicazioni containerizzate.

- **Archiviazione e database**: Sul fronte dello storage, sono ben nove i servizi di Google Cloud Platform riconducibili in questa area. Senza elencarli tutti, si posso citare Cloud Storage, Cloud SQL e Cloud Bigtable. Il primo consente storage illimitato con la possibilità di recupero dei dati in base alla frequenza desiderata, da quella standard per dati ad accesso frequente come quelli utilizzati per siti web e app per dispositivi mobili, fino ai dati che possono essere conservati per almeno 365 giorni come quelli per i registri normativi.

  Cloud SQL è un servizio completamente gestito per database relazionali MySQL, PostgreSQL e SQL Server. Permette di automatizzare backup, repliche, patch di crittografia e gli incrementi di capacità, con una disponibilità superiore al $99,95\%$ in tutte le parti del mondo. Bigtable, infine, è un database NoSQL scalabile con cui è possibile passare da centinai a milioni di operazioni di lettura/scrittura al secondo senza interruzioni, aggiungendo o rimuovendo nodi cluster.

- **Big data e machine learning**: BigQuery, nella categoria dei big data e machine learning, è il data warehouse multi-cloud serverless di Google Cloud Platform. Proposto in modalità SaaS (software-as-a-service), integra algoritmi di machine learning per ottenere insight in tempo reale su vari processi aziendali. Cloud Dataflow, invece, abilita l'elaborazione unificata dei dati sia in streaming sia in chiave storica o batch. Ha l'obiettivo di ridurre al minimo i tempi di latenza della pipeline, abbattendo anche i costi di elaborazione tramite la scalabilità automatica delle risorse, grazie alla capacità di partizionarle automaticamente e ridistribuirne l'utilizzo. Altri servizi GCP per big data e machine learning, poi, si focalizzano sull'esecuzione in ambienti open source, come Cloud Composer basato su Apache Airflow, o Dataproc per l'esecuzione di cluster Hadoop e Spark.

- **Networking**: Google Cloud Platform contempla anche funzionalità di networking come Virtual Private Cloud (VPC) con cui è possibile connettere le risorse GCP e mettere in grado i team di isolarle all'interno dello stesso spazio IP privato condiviso. Grazie al servizio software-defined Cloud Load Balancing, inoltre, la distribuzione delle risorse e il bilanciamento del carico in una o più aree geografiche in prossimità degli utenti può soddisfare requisiti di alta disponibilità, poiché è applicabile al traffico HTTP(S), TCP/SSL e UDP. A completare la parte di networking la funzionalità Cloud Interconnect estende la rete on-premises al network Google attraverso una connessione ad alta disponibilità e bassa latenza.

- **Strumenti per sviluppatori**: Sono tre gli strumenti principali al servizio degli sviluppatori compresi nella piattaforma cloud di Google: Cloud Scheduler, Container Registry e Cloud Tasks. Il primo è un servizio di livello enterprise con cui pianificare i vari job, inclusi job batch, job per big data e operazioni dell'infrastruttura cloud. Permette, anche da un'unica console, di automatizzare le attività, riducendo così il lavoro e gli interventi manuali. Container Registry è il sistema GCP concepito per archiviare e gestire le immagini del container Docker. Oltre a consentire la configurazione delle pipeline CI/CD, aiuta a rilevare le vulnerabilità in modo aggiornato con le nuove minacce. Cloud Tasks, infine, è lo strumento per scalare i microservizi in modo indipendente, giacché abilita l'esecuzione di attività asincrone e si presta a organizzare con meno code un numero elevato di attività distribuite.

- **Identità e sicurezza**: In merito alle esigenze di identità e sicurezza, i servizi di Google Cloud Platform si articolano in strumenti come Cloud Key Management per la gestione di chiavi di crittografia simmetriche e asimmetriche, nonché in servizi per l'individuazione, la classificazione e la protezione dei dati sensibili (Cloud Data Loss Prevention). Inoltre, il modello di compliance della piattaforma tiene conto delle differenze geografiche e quindi fa in modo di rendere conforme la gestione della privacy

dei dati, per esempio, alla direttiva statunitense *HIPAA* (*Health Insurance Portability and Accountability Act*) o a quella europea *GDPR* (*General Data Protection Regulation*). Inoltre viene fornita una piattaforma *IAM* (*Identity Access Management*) che consente agli amministratori di definire chi può intervenire su risorse specifiche, garantendo un controllo e una visibilità totali per la gestione centralizzata delle risorse Google Cloud.

6.2 Containerizzazione delle applicazioni

Nella progettazione della applicazione si è utilizzata una architettura a micro servizi. Nello specifico necessitiamo di tre immagini Docker:

- **Modello**: si necessita di un contenitore del modello ottenuto in grado di esporre delle api a cui poter fare richieste di inferenza.
- **Generatore**: un software in grado di leggere i dati da una sorgente (in questo caso un file csv) e di interrogare il modello di apprendimento automatico.
- **Plotter**: un software in grado di intercettare le risposte del modello e salvare i risultati in un file ed eventualmente di disegnare i grafici delle metriche ottenute.

È da specificare che nell'intera implementazione viene utilizzato un broker mqtt condiviso tra i vari microservizi, utilizzando un topic il cui nome segue le convenzioni proprie di Kubeedge.

Modello

Nella containerizzazione del modello di apprendimento automatico, si necessita di una metodologia per poter effettuare delle interrogazioni contenenti i dati su cui effettuare inferenza e, nel caso di modelli online, su cui apprendere i pattern. A tal proposito è stato utilizzato MlFlow [Chen et al., 2020], un framework per Python attraverso il quale è possibile definire delle logiche custom, all'interno di

un metodo di una classe del framework, che verranno eseguite al momento di una chiamata api rest al sistema.

Nello specifico mlflow costruisce un server http attorno al modello decisionale selezionato, serializzato in formato pickle[1]. Una volta serializzato il modello, quest'ultimo viene caricato in memoria quando viene eseguito il software mlflow. Successivamente il framework mette a disposizione tre endpoint interrogabili tramite richieste http: *ping* o anche *health*, viene utilizzato per controllare l'effettiva disponibilità del server e non torna alcuna risposta se non lo status della richiesta, *version* restituisce la versione di mlflow che si sta utilizzando, ed infine *invocations* è l'endpiont customizzabile cui è possibile inviare dati all'interno del body della richiesta http che possono essere processati. I dati inviati devono necessariamente seguire delle linee guida che vengono riportate nella documentazione[2] del framework.

Generatore

Il generatore dei dati è stato implementato utilizzando il linguaggio **GO**[3], un linguaggio di programmazione *compilato* e *fortemente tipizzato*, sviluppato da Google rilasciato alla versione 1.0 nel 2012. Seppur molto giovane rispetto ad altri linguaggi, gode di una stabilità tale che lo ha reso uno dei linguaggi più utilizzati al mondo. Basti pensare infatti che è il linguaggio di programmazione su cui si basa lo sviluppo di Kubernetes e della stra grande maggioranza dei software *cloud-native*[4].

Nello sviluppo del generatore di dati sono state utilizzate molte delle funzionalità base del linguaggio, appartenenti alla libreria standard installata con Go stesso. Nello specifico l'applicazione è in grado di leggere dati da un file csv ed interrogare il modello precedentemente descritto tramite chiamate api rest. Va precisato che il software è stato sviluppato in maniera tale che le richieste mantenessero la frequenza di campionamento originale al fine di ottenere una

[1] https://docs.python.org/3/library/pickle.html
[2] https://mlflow.org/docs/latest/models.html#id58
[3] https://go.dev/
[4] https://www.cncf.io/

simulazione il più veritiera possibile.
Ottenuta la risposta dal modello, il generatore è anche responsabile della loro pubblicazione formattata nel topic mqtt secondo le regole definite da Kubeedge.

Plotter

L'ultimo micro servizio necessario per il funzionamento delle architetture è quello responsabile della raccolta e salvataggio delle metriche di addestramento. A tal proposito è stata sviluppata l'ultima immagine docker in Python, utilizzando una libreria per la connettività al broker mqtt (paho mqtt[5]) oltre che le librerie per la generazione di immagini e per la gesione di dati tabulati: matplotlib, pandas e numpy.

Il plotter si sottoscrive al topic in gui il generatore pubblica i messaggi per l'aggiornamento delle custom resource definite da Kubeedge, effettua il parsing dei dati contenuti al loro interno ed infine salva i risultati ottenuti su un file di tipo csv. Successivamente effettua il plot dei dati contenuti all'interno del csv ogni n istanze.

6.3 Helm chart e deployment delle applicazioni

A questo punto del lavoro si è generato un unico helm chart in grado di effettuare il deployment di entrambe le architetture di cui si necessita per il nostro scopo finale. Va notato che sono state utilizzate le stesse immagini generate per entrambe le tipologie di architetture.

In figura 6.3 viene riportato uno schema di funzionamento dell'architettura edge decentralizzata. Nella figura 6.3 possiamo osservare, oltre ai container, il flusso dei dati. Partendo infatti dal file contenente i dati, che vengono letti dal generatore, essi vengono elaborati e poi inviati sotto forma di richiesta HTTP al container contenente il modello decisionale. Successivamente le risposte generate vengono formattate e pubblicate in un topic MQTT predefinito. Questi messaggi vengono intercettati sia dagli agenti di Kubeedge, e quindi utilizzati per l'aggior-

[5] https://pypi.org/project/paho-mqtt/

namento delle risorse disponibili a livello cloud, sia dal plotter che li utilizza per salvare i risultati in un file.

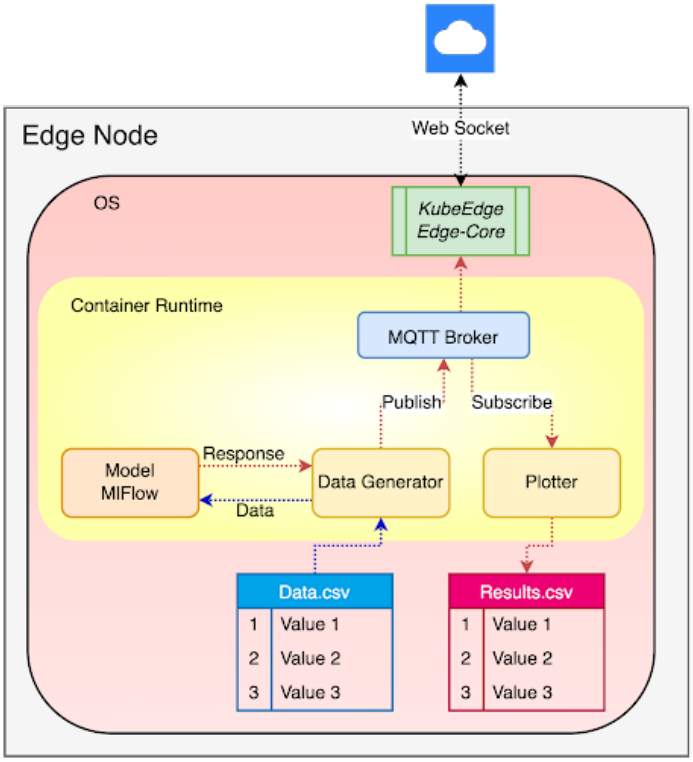


Figura 6.3: Architettura decentralizzata realizzata

In figura 6.4 possiamo osservare come le stesse immagini docker di prima vengano riutilizzate nella realizzazione dell'architettura centralizzata. In questo caso il modello decisionale ed il plotter esistono all'interno del cluster come unica istanza e, in questo caso, vengono "deployati" sul nodo master del cluster Kubernetes.

Per poter riutilizzare le stesse immagini viene "deployato" anche qui un container contenente il broker MQTT. La differenza sta nel fatto che in questo caso ogni nodo edge ha il proprio generatore di dati che effettua richieste ad un modello che è lo stesso per tutti, mentre nell'architettura precedente ogni nodo edge ha il proprio modello decisionale ed il proprio plotter.

A questo punto possiamo descrivere la struttura dell'Helm chart generato al fine di effettuare il deployment di entrambe le architetture. In figura 6.5 viene

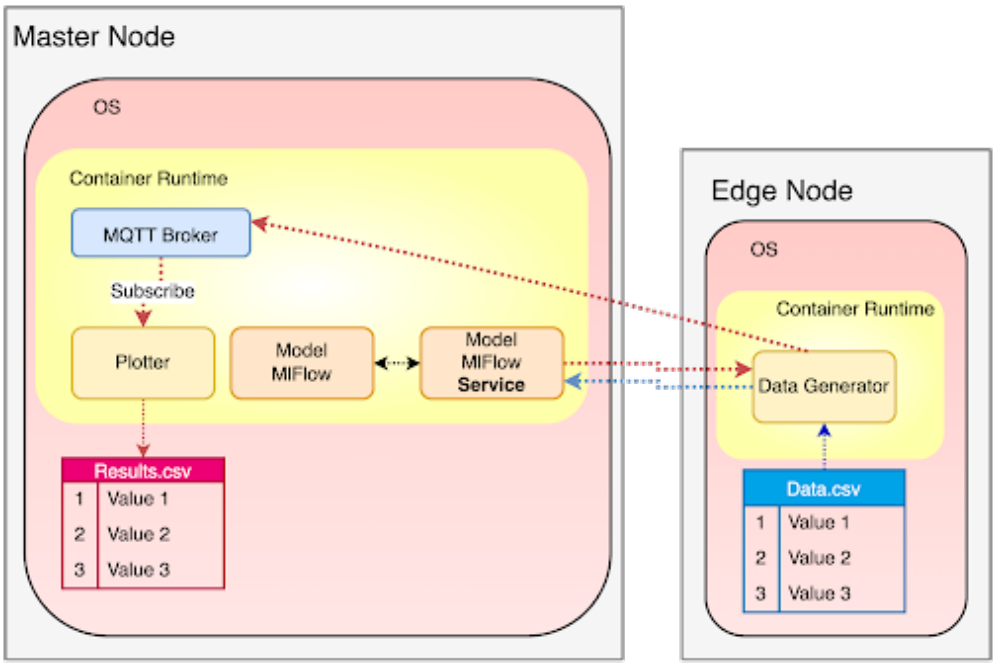


Figura 6.4: Architettura centralizzata realizzata

mostrata la struttura della directory contenente i file per il deployment delle applicazioni. Si può notare come la repository sia strutturata secondo gli standard del software Helm e presenti per cui, oltre ai file *Chart.yaml*, *values.yaml* ed il file *README.md*, le sotto cartelle **charts** e **templates**, la prima contenente i chart per il deployment dei container descritti in sezione 6.2, la seconda contenente i file di configurazione di Kubeedge (nel caso di deployment decentralizzato) e del broker MQTT Mosquitto (nel caso di deployment centralizzato).

Risulta doverosa una breve descrizione dei sotto-elementi che compongono il grafico:

- **generator**:

  contiene unicamente il deployment dell'immagine docker del generatore dati. Le repliche dei pod che verranno generate dipendono dal numero di nodi worker (edge) che si hanno a disposizione. Infatti, sia in caso di deployment centralizzato che decentralizzato verrà generato un pod per ogni nodo

```
helm-chart/
├── Chart.yaml
├── README.md
├── charts/
│   ├── generator/
│   │   ├── Chart.yaml
│   │   ├── templates/
│   │   │   └── deployment.yaml
│   │   └── values.yaml
│   ├── model/
│   │   ├── Chart.yaml
│   │   ├── templates/
│   │   │   ├── deployment.yaml
│   │   │   └── service.yaml
│   │   └── values.yaml
│   └── plotter/
│       ├── Chart.yaml
│       ├── templates/
│       │   └── deployment.yaml
│       └── values.yaml
├── templates/
│   ├── kubeedge/
│   │   ├── device-model.yaml
│   │   └── device-instance.yaml
│   └── mosquitto/
│       ├── config.yaml
│       ├── deployment.yaml
│       └── service.yaml
└── values.yaml
```

Figura 6.5: Struttura dell'Helm chart

utilizzando le regole di *anti affinity*[6] definite da Kubernetes.

- **plotter**:

  come sopra, in quanto non bisogna contattarlo direttamente, contiene unicamente il file per il deployment dell'immagine. In caso di deployment centralizzato verrà generato un unico pod che, tramite il *node selector*, verrà eseguito nel nodo master del cluster Kubernetes. In caso invece di deployment dell'applicazione decentralizzata, verranno generati più pod che vengono invece eseguiti su ogni nodo edge.

- **model**:

  il depoloyment del modello invece necessita di un service in caso di deployment centralizzato. Questo perchè in caso di deployment decentralizzato tutti i software che girano sullo stesso nodo condividono la stessa rete, per cui è semplice effettuare le chiamate rest conoscendo la porta esposta da mlflow. In caso di deployment decentralizzato i microservizi non vengono eseguiti sullo stesso nodo, dunque gli indirizzamenti vengono risolti tramite *FQDN* (*Full Qualifier Domain Name*) univoci. Anche qui è possibile generare un pod eseguito sul nodo master, oppure un pod per ogni nodo edge.

- **kubeedge**:

  nella cartella kubeedge vengono definite le custom resource tipiche del framework. Nello specifico il file *device-model.yaml* contiene la definizione delle informazioni che dovranno essere visibili a livello cloud, in questo caso le metriche ed i valori delle letture sensoriali come valori in **sola lettura**, mentre il file *device-instance.yaml* contiene l'istanza del device-model rappresentante un dato nodo edge. Infatti, in caso di deployment decentralizzato, verranno generate un unico modello (*device-model* è riutilizzabile) e sulla base del modello vengono generate n istanze, una per ogni nodo. Nel caso di deployment centralizzato questi file non vengono considerati.

[6] https://bit.ly/pod-antiaffinity

- **mosquitto**:

  necessario nel caso di deployment centralizzato, vengono definite le sue configurazioni, il deployment e, come del caso del modello, il servizio che rende possibile ai generatori di connettersi al broker in maniera agevole.

Capitolo 7

Conclusioni

In questo capitolo verranno in primo luogo riportati ed esaminati i risultati ottenuti tramite le due architetture realizzate. Successivamente verranno discussi i possibili sviluppi futuri sia per quanto riguarda la realizzazione di modelli di online machine learning, sia per la parte operations con un occhio all'edge computing.

7.1 Risultati

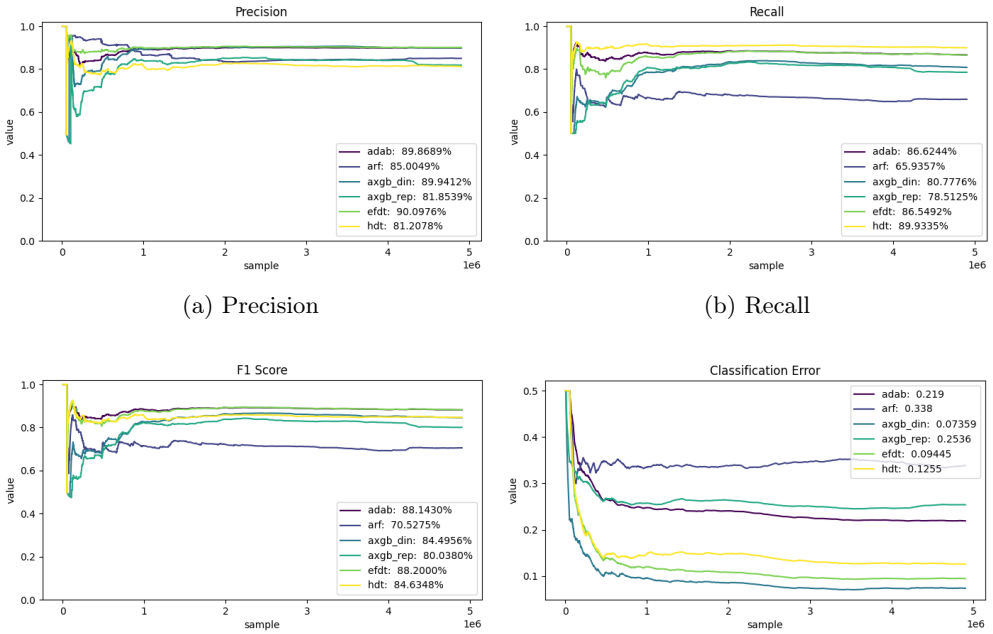
Al fine di poter testare il corretto funzionamento delle architetture, è stato realizzato un cluster Kubernetes utilizzando GCP. Nello specifico il cluster era composto da una macchina master con architettura x86 a 64 bit e 16 Gbyte di memoria RAM. Per i nodi worker sono state selezionate le macchine con la minor potenza computazionale disponibile all'interno della piattaforma al fine di simulare dei *Raspberry PI*[1]. Le macchine utilizzate disponevano quindi di una cpu con architettura arm e soli 4 Gbyte di memoria RAM. Il cluster è stato quindi composto da 3 nodi worker con Kubeedge installato, ed altri 6 nodi worker senza kubeedge per la simulazione dell'architettura centralizzata.

Il data set di partenza è stato suddiviso in 6 sotto data set al fine di simulare sensori provenienti da macchinari diversi.

[1] https://www.raspberrypi.org/

A questo punto è stato effettuato l'addestramento dei modelli nella loro versione centralizzata e decentralizzata.

In figura 7.1 viene riportato l'andamento delle metriche di classificazione ottenute dai modelli centralizzati, aventi gli stessi iper-parametri utilizzati in sezione 4.3.2. Le performance vengono poi riassunte in tabella 7.1.



(a) Precision (b) Recall

(c) F1-score (d) Classification Error

Figura 7.1: Addestramento con architettura centralizzata

Si può notare come anche in questo caso vengono confermate le ipotesi precedenti secondo le quali i modelli di apprendimento a singola istanza (nello specifico *EFDT*) risulta essere migliore dei modelli adattivi e di apprendimento ad istanza multipla, anche se si ottengono risultati buoni anche in questi ultimi casi.

Inoltre possiamo notare dalla curva di precisione e recall dei modelli *AXGB* come questi ultimi abbiano una convergenza più lenta degli altri. Questo è dovuto alla dimensione della finestra di addestramento che per questo use case risulta essere particolarmente grande.

Le problematiche inerenti la determinazione di una dimensione ottima per una finestra di addestramento sono ancora più evidenti nel caso dell'addestramento

Model	Accuracy %	Precision %	Recall %	F1-score %
AXGBu	93.28	89.94	80.78	84.50
AXGBr	90.77	81.85	78.51	80.04
HdT	91.42	81.21	**89.93**	84.63
ARF	89.34	85.00	65.94	70.53
ADAB	94.45	89.87	86.62	88.14
EFDT	**94.50**	**90.10**	86.55	**88.20**

Tabella 7.1: Addestramento con architettura centralizzata

decentralizzato, di cui viene riportato un esempio in figura 7.2.

In tabella 7.2 vengono riassunti i risultati in termini di precisione e recall dei vari modelli per ogni data set generato a partire da quello di partenza. I risultati mostrano come anche qui i modelli di apprendimento a singola istanza sono migliori per questo use case in particolare.

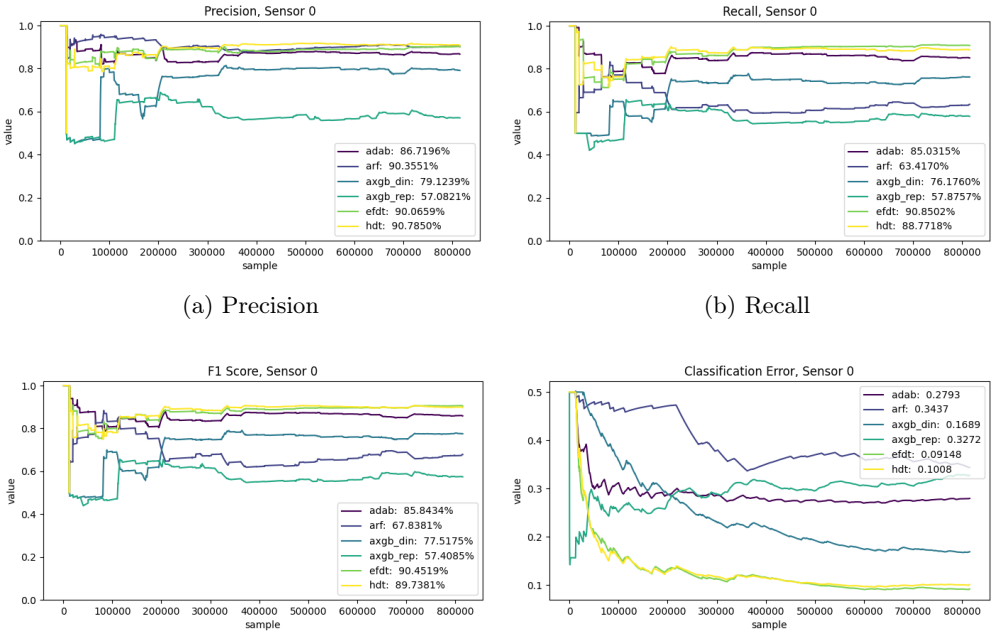
Model	Machine 0		Machine 1		Machine 2		Machine 3		Machine 4		Machine 5	
	Pre%	*Rec%*	*Pre%*	*Rec%*	*Pre%*	*Rec%*	*Pre%*	*Rec%*	*Pre%*	*Rec%*	*Pre%*	*Rec%*
AXGBu	79.12	76.18	76.07	65.24	83.15	62.06	77.11	69.87	56.92	51.61	82.15	71.02
AXGBr	57.08	57.88	62.09	63.14	59.01	64.17	60.61	63.26	60.36	63.15	50.19	50.15
HDT	90.78	88.77	70.09	68.87	91.61	83.95	74.72	69.21	67.86	54.83	68.96	53.51
ARF	**90.36**	63.42	90.04	63.62	**94.64**	73.01	**95.04**	73.85	**91.83**	67.49	**93.34**	70.26
Ada B	86.72	85.03	84.42	81.58	90.74	78.01	88.67	82.88	88.54	86.63	76.74	88.47
EFDT	90.07	**90.85**	**93.41**	**93.27**	88.33	**89.43**	88.33	**87.32**	90.29	**92.15**	90.65	**92.25**

Tabella 7.2: Addestramento con architettura decentralizzata

7.2 Considerazioni finali

Alla luce dei risultati ottenuti possiamo affermare che i modelli di apprendimento online possono essere utilizzati nel caso d'uso proposto (classificazione di anomalie in sensori caratteristici dell'industria chimica), ottenendo risultati comparabili ai classici modelli batch.

Inoltre possiamo anche affermare che generalmente le versioni centralizzate dei modelli proposti hanno delle performance di classificazione leggermente superiori



(a) Precision

(b) Recall

(c) F1-score

(d) Classification Error

Figura 7.2: Addestramento con architettura decentralizzata

alla loro versione decentralizzata. Questo è probabilmente dovuto alla diversa cardinalità dell'insieme dei dati osservati dai modelli, di molto superiore nella versione centralizzata. Per quanto riguarda i modelli *AXGB* e più in generale per i modelli di apprendimento a istanza multipla, le loro versioni centralizzate risultano essere migliori in quanto l'alternarsi delle istanze anomale e non anomale, dovute al parallelismo con cui i dati vengono mandati al modello, permette un addestramento dei weak learner più "omogeneo" rendendo la classificazione più semplice.

Generalmente, dato uno use case di questo tipo ed alla luce dei risultati ottenuti, si può affermare che l'utilizzo di modelli di apprendimento a singola istanza è preferibile, anche a causa della maggiore semplicità nella determinazione degli iper-parametri.

Con un occhio invece alla parte più infrastrutturale/architetturale, possiamo dire che effettuare il deploy dei modelli all'edge, permette una ottimizzazione dell'utilizzo delle risorse computazionali. In figura 7.3 viene mostrato l'output

ottenuto dal *metrics server*[2] di Kubernetes per entrambe le architetture realizzate. Si può notare come nonostante le cpu siano abbastanza scariche in entrambe le casistiche, nel caso decentralizzato si ha un utilizzo ottimizzato della memoria RAM (viene utilizzato il 90% circa di quella disponibile), a dimostrazione del fatto che anche con dell'hardware limitato è possibile eseguire agevolmente algoritmi di machine learning oltre che l'infrastruttura software necessaria al funzionamento di Kubeedge e Kubernetes in generale.

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
edge-ai-kubeedge-1	30m	3%	1558Mi	40%
edge-ai-kubeedge-2	35m	3%	1596Mi	41%
edge-ai-kubeedge-3	29m	2%	1568Mi	40%
edge-ai-master	287m	7%	4346Mi	27%
edge-ai-node-1	30m	3%	1311Mi	34%
edge-ai-node-2	32m	3%	1380Mi	36%
edge-ai-node-3	34m	3%	1430Mi	37%
edge-ai-node-4	29m	2%	1439Mi	37%
edge-ai-node-5	32m	3%	1372Mi	35%
edge-ai-node-6	34m	3%	1402Mi	36%

(a) Macchine senza carico di lavoro

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
edge-ai-kubeedge-1	87m	8%	3438Mi	89%
edge-ai-kubeedge-2	86m	8%	3482Mi	90%
edge-ai-kubeedge-3	76m	7%	3553Mi	92%
edge-ai-master	327m	8%	4741Mi	29%
edge-ai-node-1	41m	4%	1409Mi	36%
edge-ai-node-2	34m	3%	1471Mi	38%
edge-ai-node-3	40m	4%	1557Mi	40%
edge-ai-node-4	34m	3%	1527Mi	39%
edge-ai-node-5	36m	3%	1469Mi	38%
edge-ai-node-6	34m	3%	1542Mi	40%

(b) Macchine con carico di lavoro

Figura 7.3: Utilizzo di risorse computazionali

Per quanto riguarda infine l'utilizzo di Kubeedge come piattaforma per la distribuzione di servizi all'edge, come detto anche in sezione 5.3, il framework è ancora molto giovane, instabile e la documentazione molto insidiosa, per cui l'utilizzo di questa tecnologia in realtà aziendali potrebbe essere rischiosa. Tuttavia questa tecnologia possiede tutte le potenzialità necessarie per un suo rapido sviluppo, soprattutto in un periodo in cui le infrastrutture basate su Kubernetes sono sempre crescenti e si ha sempre più necessità di trovare delle soluzioni per quanto riguarda l'edge computing.

Queste motivazioni, affiancate alla community ed i partner che partecipano allo sviluppo fa pensare che nei prossimi anni Kubeedge sarà sempre più utilizzato anche in ambito industriale.

[2] https://kubernetes-sigs.github.io/metrics-server/

7.3 Lavori futuri

In conclusione, sono stati identificati i prossimi passi per il futuro di questo lavoro di tesi, riguardanti sia gli algoritmi di online machine learning, sia lo sviluppo di applicazioni basate su Kubeedge.

Tra i principali sviluppi futuri correlati a questo lavoro, si segnalano:

- Gradient boost per River:

  uno dei punti principali all'interno della *roadmap*[3] di sviluppo di river è proprio lo sviluppo dell'algoritmo del gradient boost, algoritmo cardine per XGBoost.

- Alberi regolarizzati ed alberi di Hoeffding:

  successivamente si potrebbe valutare l'implementazione dei weak learners adottati da XGBoost come alberi di Hoeffding. Questo non solo porterebbe ad una implementazione nativa di XGBoost per la libreria river, eliminando quindi la dipendenza dal framework xgboost batch, ma renderebbe l'algoritmo un modello di apprendimento a singola istanza eliminando molte delle problematiche viste in precedenza.

- Versioning dei modelli:

  risulta indispensabile un efficace metodologia per il versioning dei modelli. In questa maniera in caso una macchina dovesse rompersi non si perderebbero i modelli a cui si era giunti. Inoltre in caso di concept drift si potrebbero recuperare modelli vecchi e riutilizzarli.

- Sistema di feedback:

  in questo lavoro di tesi si è studiato uno use case al fine di valutare le performance in termini di classificazione ed anche di risorse computazionali. A tal proposito il feedback relativo ad una data istanza dei dati viene fornita con i dati stessi, caso ideale che non si verifica nella realtà. Risulta quindi

[3] https://bit.ly/river-roadmap

necessaria l'implementazione di un sistema in grado di fornire il feedback al modello decisionale in qualunque momento.

- Applicazione cloud:

  per quanto riguarda la parte applicativa, può risultare interessante realizzare una applicazione in grado di interagire con le api di Kubernetes al fine di ottenere le metriche tramite le custom resource di Kubeedge. In questa maniera si può implementare una interfaccia per il monitoring dei modelli in tempo reale.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[Akiba et al., 2019] Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2623–2631, New York, NY, USA. Association for Computing Machinery.

[Anguita et al., 2012] Anguita, D., Ghelardoni, L., Ghio, A., Oneto, L., and Ridella, S. (2012). The 'k' in k-fold cross validation. In *The European Symposium on Artificial Neural Networks*.

[Barredo Arrieta et al., 2020] Barredo Arrieta, A., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., and Herrera, F. (2020). Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82 – 115.

[Bifet and Gavaldà, 2007] Bifet, A. and Gavaldà, R. (2007). Learning from time-changing data with adaptive windowing. In *Learning from Time-Changing Data with Adaptive Windowing*, volume 7.

[Bifet et al., 2017] Bifet, A., Zhang, J., Fan, W., He, C., Zhang, J., Qian, J., Holmes, G., and Pfahringer, B. (2017). Extremely fast decision tree mining for evolving data streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1733–1742, New York, NY, USA. Association for Computing Machinery.

[Bousquet et al., 2004] Bousquet, O., Boucheron, S., and Lugosi, G. (2004). *Introduction to Statistical Learning Theory*, pages 169–207. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Cesa-Bianchi and Orabona, 2021] Cesa-Bianchi, N. and Orabona, F. (2021). Online learning algorithms. *Annual Review of Statistics and Its Application*, 8.

[Chen et al., 2020] Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S. A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Singh, A., Xie, F., Zaharia, M., Zang, R., Zheng, J., and Zumar, C. (2020). Developments in mlflow: A system to accelerate the machine learning lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, DEEM'20, New York, NY, USA. Association for Computing Machinery.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In Krishnapuram, B., Shah, M., Smola, A. J., Aggarwal, C., Shen, D., and Rastogi, R., editors, *KDD*, pages 785–794. ACM.

[Fette and Melnikov, 2011] Fette, I. and Melnikov, A. (2011). The websocket protocol. RFC 6455, RFC Editor. http://www.rfc-editor.org/rfc/rfc6455.txt.

[Friedman et al., 2000] Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion). *The Annals of Statistics*, 28(2):337–374.

[Galton, 1886] Galton, F. (1886). Regression towards mediocrity in hereditary stature. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 15:246–263.

[Gama et al., 2014] Gama, J., Zliobaite, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Comput. Surv.*, pages 44:1–44:37.

[Gao, 2014] Gao, J. (2014). Machine learning applications for data center optimization.

[Gneiting et al., 2019] Gneiting, T., Vogel, P., and Walz, E.-M. (2019). Receiver operating characteristic (roc) curves.

[Grinsztajn et al., 2022] Grinsztajn, L., Oyallon, E., and Varoquaux, G. (2022). Why do tree-based models still outperform deep learning on tabular data?

[Hoi et al., 2018] Hoi, S. C. H., Sahoo, D., Lu, J., and Zhao, P. (2018). Online learning: A comprehensive survey. *Neurocomputing*.

[Hulten et al., 2001] Hulten, G., Spencer, L., and Domingos, P. (2001). Mining time-changing data streams. In *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 97–106. ACM Press.

[Light, 2017] Light, R. A. (2017). Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265.

[Lundberg and Lee, 2017] Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Massart and Nédélec, 2006] Massart, P. and Nédélec, É. (2006). Risk bounds for statistical learning. *The Annals of Statistics*, 34(5):2326 – 2366.

[Merkel, 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

[Mohri et al., 2018] Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2018). *Foundations of Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2 edition.

[Montiel et al., 2021] Montiel, J., Halford, M., Mastelini, S. M., Bolmier, G., Sourty, R., Vaysse, R., Zouitine, A., Gomes, H. M., Read, J., Abdessalem, T., et al. (2021). River: machine learning for streaming data in python.

[Montiel et al., 2020] Montiel, J., Mitchell, R., Frank, E., Pfahringer, B., Abdessalem, T., and Bifet, A. (2020). Adaptive xgboost for evolving data streams. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

[Rensin, 2015] Rensin, D. K. (2015). *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472.

[Sahoo et al., 2017] Sahoo, D., Pham, Q., Lu, J., and Hoi, S. C. H. (2017). Online deep learning: Learning deep neural networks on the fly.

[Samuel, 1959] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

[Schisterman et al., 2008] Schisterman, E., Faraggi, D., Reiser, B., and Hu, J. (2008). Youden index and the optimal threshold for markers with mass at zero. *Statistics in medicine*, 27:297–315.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press.

[Spillner, 2019] Spillner, J. (2019). Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications. *arXiv/CoRR*.

[TURING, 1950] TURING, A. M. (1950). I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460.

[Van Rossum and Drake, 2009] Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace.

[Zhang, 2010] Zhang, X. (2010). *Empirical Risk Minimization*, pages 312–312. Springer US, Boston, MA.

[Zinkevich, 2003] Zinkevich, M. (2003). Online convex programming and generalized infinitesimal gradient ascent. In Fawcett, T. and Mishra, N., editors, *ICML*. AAAI Press.