ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN
INGEGNERIA E SCIENZE INFORMATICHE

ML IN AMBIENTE BROWSER: TRANSCODIFICA DI MODELLI PREDITTIVI PER LA MOBILITÀ URBANA

Tesi di laurea in Tecnologie Web

Relatore:
Dr. Giovanni Delnevo

Presentata da:
Matteo Monari

Correlatori:
Dr. Manuel Andruccioli
Dr. Kelvin Olaiya

Prima sessione di luglio 2026
Anno Accademico 2025-2026

con le Briciole si va in Carrozza ...

Introduzione

Il progressivo fenomeno dell'urbanizzazione globale ha trasformato radicalmente la morfologia delle città moderne, ponendo sfide infrastrutturali di criticità senza precedenti. All'interno di questo scenario, la gestione della mobilità e della congestione stradale rappresentano alcuni dei problemi più complessi per le amministrazioni pubbliche e per la qualità della vita dei cittadini. Gli approcci tradizionali al monitoraggio del traffico, basati su reti di sensori fisici e telecamere, si sono dimostrati intrinsecamente limitati dalla loro natura esclusivamente reattiva: essi permettono di quantificare un ingorgo stradale solamente nel momento in cui esso si è già formato, risultando inefficaci ai fini della prevenzione.

L'avvento del paradigma delle *Smart City* e la disponibilità massiva di dati storici (*Big Data*) hanno aperto la strada a una nuova prospettiva: la transizione dal monitoraggio reattivo all'analisi predittiva. Attraverso l'applicazione di algoritmi di *Machine Learning*, è oggi possibile estrapolare *pattern* spaziotemporali latenti dai dati di mobilità urbana, prevedendo i volumi di traffico con estrema precisione e con ore di anticipo. Tuttavia, l'implementazione pratica di questi sistemi si scontra sistematicamente con un ostacolo architetturale strutturale.

Attualmente, l'industria predittiva si affida quasi esclusivamente a un paradigma centralizzato (*Cloud Computing*): i modelli di Intelligenza Artificiale risiedono su server remoti, ai quali i dispositivi periferici inviano costantemente i dati di posizione degli utenti per ricevere in cambio l'inferenza. Questo approccio *Client-Server* genera ritardi legati alla latenza di

rete (*Round-Trip Time*), impone enormi costi di infrastruttura per scalare il servizio sotto carichi eccessivi e, soprattutto, espone le coordinate sensibili degli utenti a potenziali violazioni della *privacy*.

Il presente progetto di tesi si inserisce esattamente tra l'Intelligenza Artificiale e i sistemi distribuiti, proponendosi di risolvere il problema del *deployment* dei modelli predittivi attraverso l'adozione del paradigma Edge Computing. L'obiettivo primario di questo progetto è dimostrare la fattibilità e l'efficienza dello sviluppo di un'architettura software decentralizzata, in cui algoritmi di *Machine Learning* per la previsione del traffico vengono eseguiti direttamente sul dispositivo periferico dell'utente finale (*Client-Side*), all'interno dell'ambiente sicuro e isolato del *browser* web.

Spostando il calcolo matematico dal Cloud all'Edge, il sistema mira ad azzerare matematicamente la latenza di rete, garantire l'anonimato architetturale poiché il dato non lascia mai il dispositivo locale, e fornire una scalabilità orizzontale priva di costi infrastrutturali aggiuntivi.

Per concretizzare tale architettura, lo sviluppo ha seguito un percorso diviso in due macro-fasi: la modellazione predittiva dei modelli e la transcodifica a basso livello per ottimizzare il client-side.

Nella prima fase, partendo da un *dataset* storico di flussi veicolari orari, è stata condotta un'estesa analisi comparativa tra molteplici paradigmi di apprendimento automatico. Sono state analizzate architetture lineari, non parametriche (k-NN), *kernel-based* (SVR) e reti neurali artificiali (MLP). Un'attenzione particolare è stata riservada agli algoritmi basati su alberi decisionali e alle tecniche *Ensemble*, quali Random Forest, *XGBoost* e *LightGBM*. Per estrarre le massime prestazioni dalla topologia dei modelli, l'ecosistema è stato integrato con il *framework Optuna*, il quale ha permesso di ricercare gli iperparametri ideali sfruttando tecniche di Ottimizzazione Bayesiana basate sullo stimatore TPE.

La seconda fase ha affrontato l'esecuzione *browser-based* ad alte prestazioni. I modelli ad albero addestrati in *Python* sono incompatibili con l'ambiente web e troppo costosi per un interprete standard. La tesi spiega l'implemen-

tazione di una *toolchain* di compilazione. Sfruttando la libreria *Treelite* (e il generatore *tl2cgen*), il grafo computazionale del modello ottimizzato è stato tradotto interamente in codice C nativo. Successivamente, tramite il compilatore *Emscripten*, l'algoritmo è stato transcodificato nel formato *WebAssembly*. Questo file binario, eseguibile all'interno della *sandbox* del browser, garantisce prestazioni pressoché identiche all'hardware nativo.

Infine, per certificare la solidità della soluzione, la tesi eseugue uno *Stress Test* telemetrico automatizzato confrontando il binario WASM nativo con gli standard del web, quali l'ONNX Runtime Web accelerato su CPU e scheda video.

Il documento è organizzato in tre capitoli principali, strutturati come segue:

- Il **Capitolo 1** inquadra il dominio del problema, analizzando le criticità della mobilità nelle *Smart City* e delineando l'evoluzione dei paradigmi computazionali dal Cloud Computing verso i sistemi distribuiti.

- Il **Capitolo 2** descrive la fase di *Data Ingestion* ed esplora le basi teoriche dei modelli di *Machine Learning* esaminati, analizzandone i fondamenti matematici e introducendo la *pipeline* di Ottimizzazione Bayesiana degli iperparametri.

- Il **Capitolo 3** rappresenta il cuore implementativo e ingegneristico della ricerca. Dettaglia passo dopo passo la trasformazione del codice Python in codice C, la successiva compilazione in WebAssembly, lo sviluppo dell'interfaccia utente in HTML/JS, e si conclude con l'analisi dei risultati dai *benchmark* eseguiti sul web.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Contesto, Teoria e Paradigmi Architetturali

Il ventunesimo secolo è caratterizzato da un fenomeno di urbanizzazione globale senza precedenti storici. Questa massiccia concentrazione demografica ed economica all'interno delle metropoli porta con sé sfide logistiche, ambientali e infrastrutturali di enorme portata. Tra le criticità più urgenti spicca la gestione della mobilità: la congestione del traffico urbano non rappresenta solamente un disagio per il singolo cittadino, ma si traduce in un danno economico tangibile e in un drastico incremento delle emissioni inquinanti di gas serra.

1.1 Introduzione alle Smart City e agli Intelligent Transportation Systems (ITS)

Negli ultimi due decenni si è consolidato il paradigma della **Smart City** (Città Intelligente). Una Smart City può essere definita come un ecosistema urbano che integra in modo pervasivo le Tecnologie dell'Informazione e della Comunicazione (ICT) e l'Internet of Things (IoT), ovvero una rete di oggetti fisici dotati di sensori, software e connessione internet, per ottimizzare l'efficienza dei servizi cittadini, ridurre gli sprechi e migliorare la qualità

della vita. In questo modello, la città cessa di essere un aggregato passivo di infrastrutture di cemento e asfalto, trasformandosi in una rete interconnessa capace di generare, raccogliere e analizzare flussi continui di dati in tempo reale.

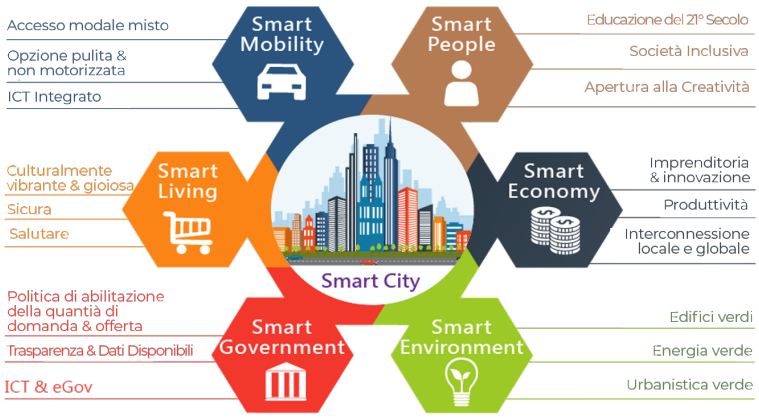

Figura 1.1: Architettura logica a livelli di una Smart City moderna. Fonte: [1]

Come illustrato in Figura 1.1, la letteratura scientifica e istituzionale concorda tipicamente nel suddividere questo vasto ecosistema in sei macro-aree o pilastri fondamentali, ciascuno deputato all'ottimizzazione di uno specifico aspetto della vita urbana [2]:

- ***Smart People***: mira a favorire una società inclusiva, promuovendo l'educazione del ventunesimo secolo e l'apertura alla creatività civica;
- ***Smart Economy***: si focalizza su imprenditoria, innovazione e produttività, garantendo un'interconnessione fluida tra il mercato locale e quello globale;
- ***Smart Environment***: persegue la sostenibilità ecologica attraverso lo sviluppo di edifici a impatto zero, l'utilizzo di energia verde e un'urbanistica attenta all'ecosistema;

- ***Smart Government***: sfrutta l'ICT e i servizi di *e-Government* per garantire trasparenza, disponibilità dei dati e politiche abilitanti per la gestione della domanda e dell'offerta pubblica;

- ***Smart Living***: punta a garantire un ambiente urbano che sia culturalmente vibrante, gioioso, salutare e, soprattutto, sicuro per i cittadini;

- ***Smart Mobility***: promuove un accesso modale misto, incentivando opzioni di trasporto pulite e non motorizzate attraverso un'infrastruttura ICT profondamente integrata.

All'interno dell'ampio ecosistema delle Smart City, il pilastro fondamentale dedicato alla mobilità è costituito dagli ITS. I sistemi di trasporto intelligente rappresentano la convergenza tra ingegneria dei trasporti, telecomunicazioni e informatica. Il loro scopo è applicare tecnologie avanzate ai veicoli e alle infrastrutture stradali per monitorare e gestire i flussi di traffico.

Storicamente, l'architettura di un sistema ITS si articola in quattro fasi logiche consequenziali, le quali definiscono il ciclo di vita dell'informazione all'interno della rete stradale e strutturano la fondazione teorica delle architetture da noi investigate:

1. **Acquisizione dei Dati (*Data Ingestion*):** Questa fase rappresenta il punto di contatto fondamentale tra l'infrastruttura fisica e l'ecosistema digitale, determinando la qualità e la capillarità dell'intero sistema predittivo. L'acquisizione avviene tramite l'implementazione di una rete pervasiva e multi-sorgente di sensori appartenenti al paradigma dell'IoT. Tra i dispositivi di tipo fisso, le spire elettromagnetiche a induzione, inserite nel manto stradale, registrano le variazioni del campo magnetico provocate dal transito dei veicoli, fornendo misurazioni puntuali ed estremamente precise su volume, densità e velocità locale. A queste si aggiungono moderni sistemi di videosorveglianza equipaggiati con algoritmi di visione artificiale per il riconoscimento ottico dei caratteri delle targhe (*Automatic Number Plate Recognition* - ANPR) e

sensori radar a microonde. Parallelamente, le sorgenti di tipo mobile ricavano flussi continui di geolocalizzazione resi anonimi direttamente dai dispositivi mobili dei passeggeri e dai computer di bordo dei veicoli interconnessi. Tale eterogeneità permette di raccogliere un volume informativo massivo, catturando sia la densità puntuale sia la traiettoria distribuita della mobilità urbana.

2. **Trasmissione delle Informazioni:** Una volta raccolti in periferia, i flussi di dati grezzi devono essere veicolati verso i nodi computazionali deputati all'analisi. Le infrastrutture ITS si affidano a un'architettura di comunicazione ibrida e gerarchica. I sensori fissi stradali utilizzano prevalentemente dorsali cablate in fibra ottica, caratterizzate da un'elevata immunità ai disturbi elettromagnetici ambientali e da una banda passante simmetrica ad altissima capacità. Al contrario, i sensori mobili e i nodi periferici distribuiti sul territorio si interfacciano necessariamente con reti wireless ad alta velocità e bassissima latenza, quali lo standard cellulare di quinta generazione (5G). L'introduzione delle reti 5G, in particolare, abilita logiche avanzate di *network slicing*, ovvero una tecnologia che permette di dividere un'infrastruttura di rete fisica condivisa in più reti logiche e virtuali indipendenti, garantendo canali di comunicazione prioritari, isolati e protetti per il transito dei dati legati alla sicurezza e al monitoraggio stradale, minimizzando il rischio di perdite di pacchetti informativi durante le fasi critiche di congestione viaria o saturazione dello spettro radio nello stato fisico.

3. **Elaborazione e Analisi:** Questo livello rappresenta il nucleo logico, matematico e decisionale dell'intero sistema ITS. In questa fase, i dati grezzi provenienti dalle fasi precedenti vengono sottoposti a stringenti procedure di pulizia, normalizzazione, filtraggio del rumore e allineamento temporale, al fine di eliminare letture spurie o malfunzionamenti intermittenti dei sensori. Successivamente, le informazioni strutturate vengono processate tramite modelli statistici e algoritmi di *Machine*

Learning per estrarre pattern comportamentali ricorrenti, calcolare le matrici di origine-destinazione degli spostamenti e identificare in tempo reale le anomalie macroscopiche, quali incidenti stradali, cantieri temporanei o colli di bottiglia strutturali.

4. **Erogazione dei Servizi:** L'anello di chiusura del sistema ITS traduce l'output dell'elaborazione analitica in azioni concrete, coordinate e tempestive sulla rete stradale, con l'obiettivo di ripristinare l'equilibrio del flusso veicolare. L'erogazione si divide in interventi diretti sull'infrastruttura e comunicazioni rivolte all'utenza. Nel primo caso, i sistemi semaforici adattivi rimodulano dinamicamente i tempi di ciclo, di verde e di rosso in base alla densità di traffico stimata in tempo reale, ottimizzando le intersezioni critiche. Nel secondo caso, le informazioni predittive vengono veicolate sia attraverso dispositivi fisici stradali, come i pannelli a messaggio variabile (PMV) dislocati lungo le arterie principali, sia mediante canali digitali integrati nei sistemi di navigazione satellitare e nelle applicazioni di mobilità dei conducenti, suggerendo percorsi alternativi dinamici prima del verificarsi effettivo dell'ingorgo.

Al fine di comprendere l'impatto di questi quattro stadi strutturali nell'evoluzione della mobilità intelligente, è opportuno analizzare in che modo i parametri operativi si siano evoluti rispetto ai sistemi classici. Come evidenziato in dettaglio nella Tabella 1.1, l'introduzione dei sistemi ITS ha radicalmente trasformato ogni singola caratteristica della gestione stradale, ridefinendo il flusso informativo, le logiche di controllo semaforico, i canali di comunicazione e le modalità di intervento sulla rete viaria urbana.

Caratteristica	Gestione Tradizionale	Gestione ITS
Flusso Dati	Statico o assente. Le misurazioni sono campionarie, storiche e raccolte manualmente tramite campagne di rilevamento periodiche a costi elevati e a bassa frequenza.	Dinamico in tempo reale. Flusso continuo, automatizzato e pervasivo proveniente da reti di sensori IoT eterogenei e interconnessi su scala urbana.
Semafori	A ciclo fisso temporizzato. I piani semaforici sono rigidi, tarati su medie statistiche del passato e totalmente incapaci di reagire ad anomalie o eventi imprevisti.	Adattivi in base alle code. Algoritmi di controllo ottimizzano istante per istante i cicli di transizione delle intersezioni, minimizzando i tempi morti di attesa.
Comunicazione	Pannelli standard. La diffusione delle informazioni è localizzata geograficamente, limitata a stringhe di testo brevi e visibile solo in prossimità fisica del supporto.	*Routing* in-app e V2X. Comunicazione bidirezionale pervasiva tra veicolo e infrastruttura (*Vehicle-to-Everything*) direttamente sui terminali di bordo degli utenti.
Intervento	Puramente reattivo. L'azione correttiva da parte delle autorità o del sistema avviene unicamente a congestione già formata, prolungando i tempi di smaltimento del blocco.	Proattivo tramite *Machine Learning*. Modelli predittivi anticipano la formazione dei colli di bottiglia, consentendo una pianificazione preventiva dei flussi viari.

Tabella 1.1: Evoluzione dettagliata dei paradigmi operativi e prestazionali nella gestione della mobilità urbana.

Nonostante l'enorme salto qualitativo garantito dall'adozione degli ITS, la maggior parte delle infrastrutture attuali opera ancora secondo un paradigma puramente *reattivo*. In altre parole, il sistema rileva la congestione (la coda formatasi a un incrocio) e solo successivamente interviene per tentare di

mitigarla. Tuttavia, nel momento in cui l'ingorgo stradale viene rilevato, il decadimento delle performance della rete viaria è già in atto, e le tempistiche necessarie per il deflusso veicolare risultano fisiologicamente lunghe.

È in questo specifico limite tecnologico che si inserisce l'Intelligenza Artificiale applicata alle serie storiche, segnando il passaggio cruciale dal paradigma reattivo a quello *proattivo e predittivo*. Prevedere il volume di traffico di un incrocio in anticipo permette alle amministrazioni di agire *prima* che la saturazione si verifichi, ottimizzando l'infrastruttura preventivamente.

In questo contesto di innovazione, abbiamo ideato e sviluppato una soluzione architetturale che mira a decentralizzare l'intelligenza predittiva degli ITS. Affinché i sistemi di previsione del traffico possano scalare su migliaia di incroci garantendo risposte in tempo reale, esploreremo come l'elaborazione del Machine Learning possa essere distaccata dai classici server Cloud centralizzati per essere portata direttamente sui dispositivi periferici (Edge Computing). Questo approccio non solo abbatte i tempi di latenza, ma introduce intrinseci vantaggi legati alla scalabilità economica e alla preservazione della privacy dei dati di geolocalizzazione cittadina.

1.2 Fondamenti di *Machine Learning* per l'Analisi Predittiva

Il *Machine Learning* rappresenta un profondo cambio di paradigma rispetto all'ingegneria del software tradizionale: anziché codificare esplicitamente una serie di regole rigide e istruzioni condizionali per risolvere un problema, si fornisce a un algoritmo un vasto volume di dati storici. [3].

L'obiettivo primario di qualsiasi modello di *Machine Learning* non è la mera memorizzazione dei dati storici, bensì la **generalizzazione**, ovvero la capacità di restituire previsioni accurate e affidabili quando esposto a dati futuri mai osservati in precedenza. Durante lo sviluppo e la taratura di un modello predittivo, i due rischi speculari che si contrappongono alla capacità

di generalizzazione derivano dal noto teorema del ***Bias-Variance Trade-off*** (Dilemma Bias-Varianza) [4, 5]:

- ***Underfitting*** **(Sottoadattamento):** si verifica quando l'architettura dell'algoritmo è matematicamente troppo rigida o computazionalmente troppo semplice per catturare le non-linearità e la complessità intrinseca dei dati (ad esempio, il tentativo di approssimare una fluttuazione parabolica del traffico con una semplice retta lineare). Il modello pecca di eccessiva approssimazione concettuale. Dal punto di vista delle metriche di valutazione, un modello in *underfitting* esibirà prestazioni scadenti fin da subito, mostrando un elevato errore predittivo sia sui dati storici di addestramento (*training error*) sia sui dati futuri.

- ***Overfitting*** **(Sovradattamento):** rappresenta la criticità più frequente nello sviluppo dell'Intelligenza Artificiale. Si manifesta quando il modello possiede un'eccessiva capacità espressiva (troppi parametri o gradi di libertà) e finisce per "imparare a memoria" non solo i *pattern* fisiologici del fenomeno, ma anche il rumore statistico, le fluttuazioni casuali e le anomalie (*outlier*) presenti esclusivamente nel set di addestramento. Un modello in *overfitting* perde totalmente la sua capacità deduttiva: presenterà prestazioni apparentemente perfette (errore prossimo allo zero) sui dati storici, ma fallirà drasticamente nella realtà operativa, esibendo un errore altissimo sui dati di collaudo (*validation/test error*).

L'arte dell'ingegneria del *Machine Learning* consiste proprio nel calibrare la complessità del modello per trovare il punto di minimo globale della funzione d'errore, fermando l'addestramento prima che l'algoritmo inizi a memorizzare il rumore (fenomeno contrastabile tramite tecniche di regolarizzazione e *early stopping*).

Per prevenire queste criticità e valutare in modo scientifico la reale capacità di generalizzazione del sistema, la metodologia standard prevede la

ripartizione del *dataset* iniziale in tre sottoinsiemi disgiunti e mutuamente esclusivi. Come illustrato graficamente in Figura 1.2, le tre partizioni assumono ruoli ben distinti nel ciclo di vita dello sviluppo algoritmico:

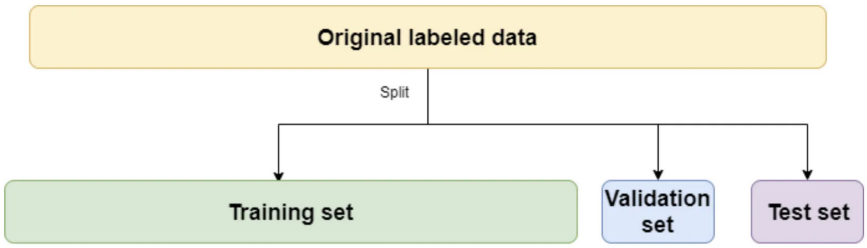

Figura 1.2: Schema metodologico della ripartizione del *dataset* in *Training*, *Validation* e *Test set*. Fonte: [6]

- ***Training set*:** Costituisce la porzione più corposa dei dati (tipicamente tra il 70% e l'80% del totale). È l'unico sottoinsieme che l'algoritmo "osserva" attivamente durante la fase di apprendimento per calcolare e ottimizzare i propri pesi interni e i parametri matematici.

- ***Validation set*:** Rappresenta una porzione intermedia (circa il 10-15%). Non viene utilizzato per calcolare i pesi dell'algoritmo, ma è fondamentale per la fase di taratura degli iperparametri, configurazioni strutturali del modello, come la profondità massima di un albero decisionale nei modelli ad albero. Svolge inoltre il ruolo di "termometro" durante l'addestramento: misurando periodicamente l'errore sul *validation set*, è possibile interrompere anticipatamente l'apprendimento (*early stopping*) non appena le prestazioni iniziano a degradare, prevenendo così l'innesco dell'*overfitting*.

- ***Test set*:** Costituisce l'ultimo 10-15% dei dati e viene segregato fino al termine assoluto della fase di sviluppo. È l'ambiente di collaudo definitivo e imparziale. Poiché il modello non ha mai interagito con questi dati né direttamente né indirettamente, l'errore misurato sul *test set*

rappresenta la stima più verosimile di come si comporterà l'algoritmo nel mondo reale, ovvero di come prevederà il traffico futuro negli incroci indicati.

All'interno dell'ampio spettro di questa disciplina, il nostro problema di previsione del traffico si colloca nella macro-categoria dell'**Apprendimento Supervisionato** (*Supervised Learning*). In questo paradigma, all'algoritmo viene fornito un *dataset* contenente coppie di input e output noti. Nel nostro contesto, l'input è rappresentato dalle variabili spaziali e temporali, mentre l'output noto, comunemente definito *variabile target* o *label*, è il numero effettivo di veicoli transitati in quel momento.

Come schematizzato in Figura 1.3, l'obiettivo della fase di addestramento è minimizzare la funzione di errore, definita come la discrepanza tra le previsioni generate dal modello e i valori reali presenti nel set di dati. Una volta minimizzato tale errore, il modello è considerato addestrato e pronto per la fase di previsione su dati futuri mai osservati prima.

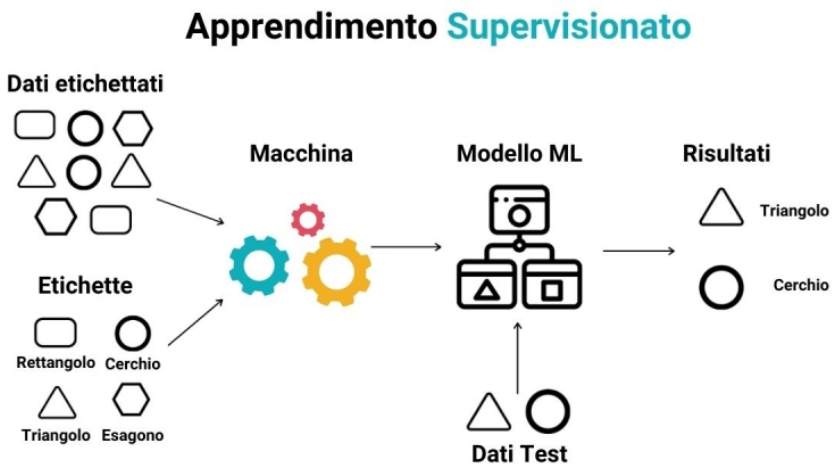

Figura 1.3: Schema logico del flusso di Apprendimento Supervisionato: dai dati di addestramento all'inferenza predittiva. Fonte: [7]

L'apprendimento supervisionato si divide ulteriormente in due branche principali, differenziate matematicamente dalla natura topologica dello spa-

zio dei dati di *output* che si desidera prevedere e dalla conseguente scelta della funzione di costo (*loss function*) da minimizzare:

- **Classificazione:** viene applicata quando lo spazio dei *target* $\mathcal{Y}$ è un insieme discreto, finito e non metrico di elementi, definiti classi o etichette (*label*). Formalmente, l'algoritmo mira a mappare uno spazio di *input* multidimensionale $\mathcal{X} \in \mathbb{R}^n$ in un'etichetta categoriale, definendo una funzione di decisione $f : \mathcal{X} \to \mathcal{Y}$. A seconda della cardinalità dell'insieme $\mathcal{Y}$, si distingue tra classificazione binaria ($|\mathcal{Y}| = 2$) e classificazione multi-classe ($|\mathcal{Y}| > 2$).

  Un esempio classico e di fondamentale importanza nell'ambito degli ITS è la determinazione del Livello di Servizio (*Level of Service* - LOS) di un'infrastruttura stradale, in cui lo stato di congestione viene segmentato in categorie discrete e qualitative (ad esempio: "flusso libero", "flusso stabile", "flusso instabile", "congestione forzata"). Dal punto di vista algoritmico, il modello non si limita a predire l'etichetta pura, ma calcola una distribuzione di probabilità su tutte le classi possibili, assegnando l'elemento alla classe che massimizza la probabilità a posteriori.

  La valutazione delle performance di un classificatore non si basa sulla distanza numerica dall'errore, bensì su metriche combinatorie derivate dalla matrice di confusione (*confusion matrix*), quali l'accuratezza (*accuracy*), la precisione (*precision*), il richiamo (*recall*) e il punteggio (*F1-score*), i quali pesano in modo differenziato l'impatto dei falsi positivi e dei falsi negativi a seconda dei requisiti operativi del sistema;

- **Regressione:** viene impiegata quando la variabile dipendente da stimare appartiene a uno spazio continuo, tipicamente rappresentato dall'insieme dei numeri reali o da un suo sottoinsieme denso, per cui $\mathcal{Y} \in \mathbb{R}$. L'obiettivo geometrico ed elaborativo della regressione è l'approssimazione di una superficie o di una funzione iper-dimensionale $f : \mathcal{X} \to \mathbb{R}$

che interpoli al meglio i punti campionari del *training set*, minimizzando i residui di predizione.

Un possibile esempio è la creazione di un algoritmo sviluppato per effettuare il *forecasting* quantitativo del flusso veicolare atteso a una specifica strada o incrocio (*variabile di target*), restituendo come *output* un valore numerico continuo che esprime il flusso di veicoli nell'unità di tempo.

A differenza della classificazione, l'errore nella regressione possiede una proprietà metrica intrinseca: lo scostamento tra il valore predetto $\hat{y}$ e il valore reale y viene quantificato attraverso funzioni di penalità continue. Le metriche standard da noi adottate per la validazione includono l'errore quadratico medio (*Mean Squared Error* - MSE), che penalizza quadraticamente le deviazioni macroscopiche fungendo da rilevatore di anomalie, l'errore assoluto medio (*Mean Absolute Error* - MAE), che offre una misura lineare e robusta dell'errore medio, e il coefficiente di determinazione (R^2), fondamentale per quantificare la proporzione di varianza dei dati catturata dall'architettura predittiva.

Quando si affrontano problemi di regressione su dati strutturati (dati organizzati in tabelle con righe e colonne ben definite, come i classici database relazionali o file CSV), lo stato dell'arte della ricerca scientifica dimostra che le architetture basate su Reti Neurali Profonde (*Deep Learning*) risultano spesso sovradimensionate e computazionalmente inefficienti. Per i dati tabulari, disposti in righe e colonne, i risultati migliori in termini di accuratezza e velocità di convergenza sono tipicamente ottenuti dai modelli ad albero, noti come **Alberi Decisionali** (*Decision Trees*) [8].

1.2.1 Fondamenti degli Alberi di Decisione (CART)

Un Albero Decisionale è un modello predittivo non parametrico che mappa le osservazioni di un elemento in conclusioni sul suo valore target attraverso una struttura gerarchica ad albero. Nel nostro specifico dominio (la

previsione di un valore numerico continuo come il flusso veicolare), si fa riferimento agli **Alberi di Regressione**, la cui implementazione standard segue l'algoritmo CART (*Classification and Regression Trees*).

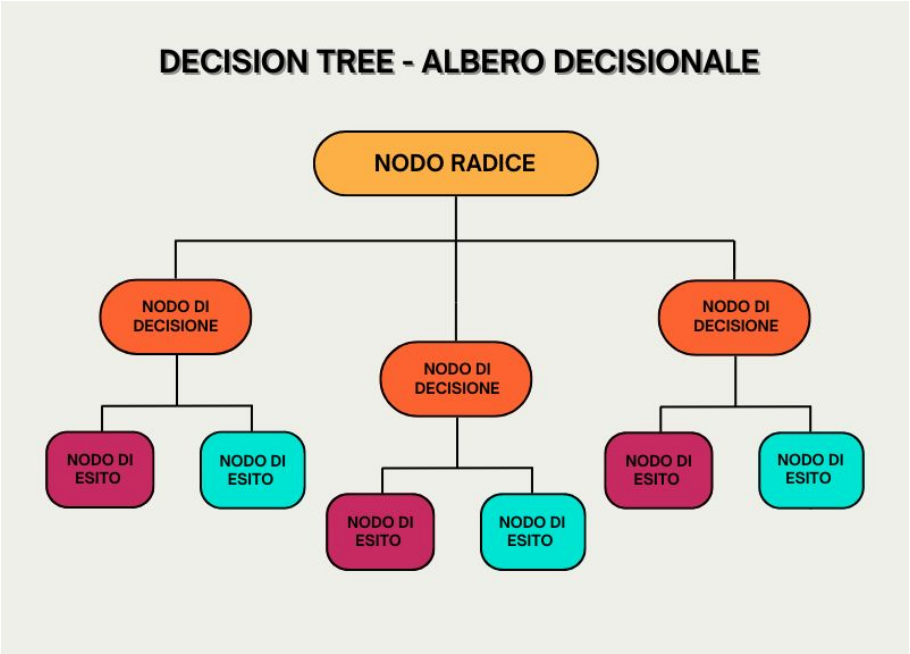

Figura 1.4: Struttura di un albero di decisione. Fonte: [9]

Come si evidenzia in 1.4, l'architettura è composta da:

- **Nodo Radice (*Root*):** contiene l'intero *dataset* di partenza prima di subire qualsiasi divisione;
- **Nodi Interni (*Splits*):** rappresentano le variabili di input (es. "L'ora attuale è maggiore delle 18:00?"). Da questi nodi si diramano le biforcazioni logiche derivanti dalle condizioni imposte;
- **Nodi Foglia (*Leaves*):** costituiscono i punti terminali dell'albero e contengono il valore numerico della previsione finale.

Dal punto di vista puramente geometrico e matematico, l'addestramento di un albero di regressione consiste nel partizionare ricorsivamente lo spazio

iper-dimensionale delle *feature* $\mathcal{X}$ in M regioni distinte e non sovrapposte $(R_1, R_2, ..., R_M)$. Per ogni osservazione che cade all'interno di una specifica regione R_m, la previsione $\hat{y}$ restituita dal modello è calcolata come la media aritmetica di tutti i valori *target* dei campioni di addestramento appartenenti a quella regione:

$$\hat{y}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i \tag{1.1}$$

dove N_m è il numero di osservazioni nella regione R_m.

Il processo di ramificazione (*splitting*) segue un approccio di tipo *greedy* (basato sul fare sempre la scelta che appare migliore (ottimo locale) al momento, con la speranza che questa porti alla soluzione globale ottimale) e *top-down* (consiste nel partire da un problema generale (la "radice") e suddividerlo ricorsivamente in sottoproblemi sempre più semplici, fino ad arrivare a istruzioni elementari e facilmente implementabili). Ad ogni nodo, l'algoritmo valuta tutte le possibili variabili e tutti i possibili punti di taglio numerici, selezionando la combinazione che minimizza maggiormente la funzione di costo nei due nodi figli risultanti. Per la regressione, il criterio di purezza adottato è solitamente la minimizzazione della somma dei quadrati dei residui (SSR - *Sum of Squared Residuals*) o la varianza all'interno dei nodi.

1.3 Paradigmi Architetturali per il Deployment dell'AI

Nella fase di progettazione di un sistema di Intelligenza Artificiale, l'addestramento del modello rappresenta solamente una frazione dell'intero ciclo di vita del software. Una volta validato, il modello predittivo deve essere integrato in un ambiente di produzione affinché possa ricevere dati reali e restituire previsioni fruibili. Questa fase prende il nome di *deployment* [10].

Nel contesto specifico degli ITS, dove le decisioni sulla viabilità devono essere prese in tempo reale o con larghissimo anticipo per prevenire conge-

stioni, la scelta dell'architettura di *deployment* diventa il fattore critico di successo dell'intera infrastruttura. Come evidenziato in Figura 1.5, esistono due paradigmi architetturali diametralmente opposti per il rilascio di modelli di *Machine Learning*: l'approccio centralizzato (Cloud AI) e l'approccio decentralizzato (Edge AI).

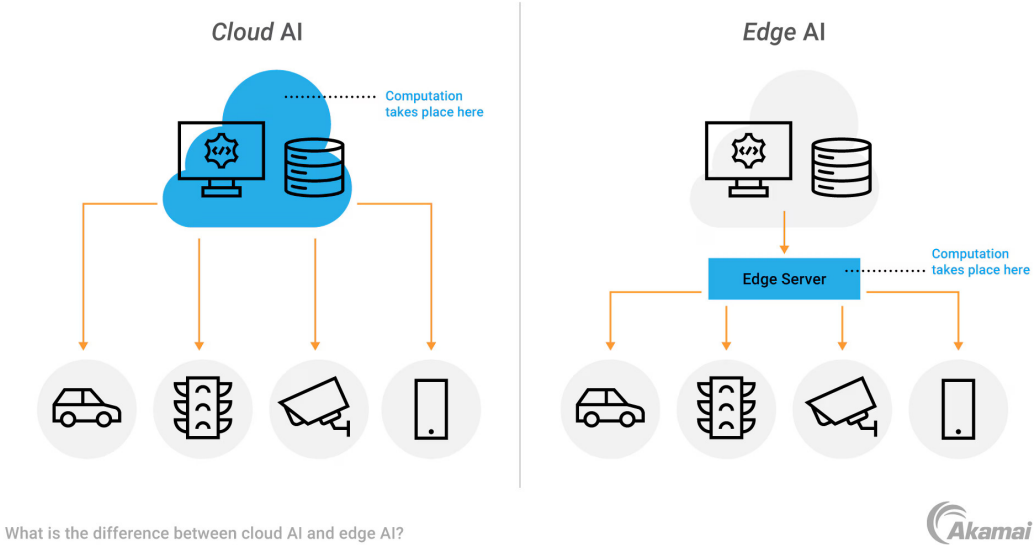

Figura 1.5: Rappresentazione schematica dei flussi di dati: a sinistra il paradigma Cloud-Centric, a destra il paradigma Edge Computing. Fonte: [11]

1.3.1 Architetture Centralizzate (Server-Side e Cloud AI)

Storicamente, l'approccio predominante nell'industria informatica consiste nel mantenere il modello addestrato confinato all'interno di server centralizzati o infrastrutture Cloud ad alte prestazioni. In questo scenario, i dispositivi periferici (come i semafori intelligenti, le telecamere agli incroci o gli smartphone degli utenti) agiscono come *thin client*, ovvero nodi che si limitano a raccogliere i dati e a inviarli al server tramite chiamate di rete

(es. API RESTful o gRPC). Il server riceve il payload, esegue l'inferenza matematica, e rispedisce il risultato al client.

Se da un lato questa architettura semplifica enormemente gli aggiornamenti del software, poiché il modello risiede in un unico punto, dall'altro introduce limitazioni infrastrutturali severe quando applicata su scala urbana:

- **Latenza di Rete:** Ogni singola previsione richiede un *round-trip time* (RTT), ovvero il tempo fisico affinché il pacchetto dati viaggi dal dispositivo al server e viceversa. Se la rete cellulare è congestionata, l'attesa può degradare l'esperienza utente o ritardare operazioni critiche;

- **Colli di Bottiglia e Scalabilità:** Se migliaia di incroci o veicoli inviano richieste simultanee di calcolo del traffico, il server centrale subisce picchi di carico (collo di bottiglia computazionale) che richiedono un costoso ridimensionamento orizzontale delle macchine in Cloud per evitare disservizi (*downtime*);

- **Dipendenza dalla Connettività:** In caso di caduta della rete internet, l'intero sistema ITS cessa di funzionare, in quanto i sensori perdono il collegamento con il proprio server centrale.

1.3.2 Architetture decentralizzate (Client-Side e Edge AI)

Per superare i limiti strutturali del Cloud Computing, l'attuale stato dell'arte si sta orientando verso il paradigma dell'**Edge Computing** [12]. Invece di portare i dati verso il modello matematico, questo paradigma inverte il flusso: è il modello matematico a essere pacchettizzato e inviato direttamente sui dispositivi periferici (*Edge Node*). Come sintetizzato nella Tabella 1.2, il paradigma Edge risolve i problemi prestazionali, pur introducendo nuove sfide tecnologiche, prima fra tutte la necessità di ottimizzare l'impronta di

memoria (*Memory Footprint*) del modello affinché possa essere eseguito su dispositivi con hardware limitato.

Metrica	**Cloud AI (Server-Side)**	**Edge AI (Client-Side)**
Latenza	Alta (dipende dalla rete)	Bassissima (tempo CPU locale)
Requisiti Hardware	Alti (lato server)	Adattivi (lato client)
Costi Operativi	Elevati (costi Cloud fissi)	Trascurabili (distribuiti)
Dipendenza Rete	Totale (sempre *online*)	Parziale (funziona *offline*)

Tabella 1.2: Sintesi dei vantaggi e svantaggi tra architetture Cloud e Edge Computing.

1.4 Privacy-Preserving AI e Sicurezza del Dato

Nel moderno panorama della progettazione dei sistemi informatici, l'efficienza prestazionale e l'accuratezza predittiva non costituiscono più gli unici parametri di valutazione per il successo di un'architettura software. La massiccia digitalizzazione dei servizi urbani e l'introduzione di normative stringenti in materia di tutela della riservatezza — prima fra tutte il Regolamento Generale sulla Protezione dei Dati (GDPR) in ambito europeo — hanno elevato la sicurezza del dato a requisito stringente e non negoziabile. Nel contesto degli ITS, questa tematica assume una rilevanza critica: i dati legati alla mobilità stradale, e in particolar modo i flussi di geolocalizzazione spaziale e temporale, portano con sé informazioni intrinsecamente sensibili relative agli spostamenti, alle abitudini e alla routine quotidiana dei cittadini.

Un'architettura basata su Cloud AI impone la trasmissione continua di pacchetti informativi dai nodi periferici verso un'infrastruttura remota. Per calcolare la previsione del flusso veicolare di uno specifico incrocio, l'ap-

plicazione client deve inviare al server un *payload* contenente coordinate geografiche precise, identificativi spaziali e marcatori temporali.

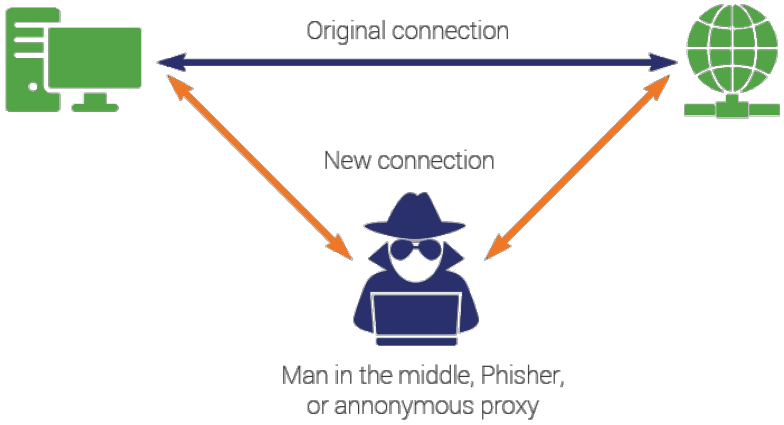

Figura 1.6: Phisher posizionato tra client e server centrale per eseguire un attacco. Fonte: [13]

Come illustrato in Figura 1.6, l'invio costante di tali vettori di *feature* attraverso canali di comunicazione pubblici espone il sistema a molteplici vulnerabilità informatiche, che abbiamo sintetizzato e classificato in tre macro-categorie:

1. **Intercettazione durante il transito (*Man-in-the-Middle*):** nei paradigmi basati su elaborazione in Cloud, i vettori contenenti le funzionalità spaziali e temporali devono necessariamente attraversare molteplici nodi di instradamento prima di raggiungere il server di destinazione. Sebbene l'infrastruttura adotti tipicamente protocolli di crittografia asimmetrica per la sicurezza del livello di trasporto (come TLS/HTTPS), le reti radiomobili (4G/5G) e le reti wireless cittadine presentano intrinseche vulnerabilità strutturali. Un attore malevolo può posizionarsi logicamente tra il dispositivo client e il server centrale, orchestrando un attacco *Man-in-the-Middle* (MitM). Sfruttando tecniche

avanzate l'attaccante può bypassare la cifratura ed eseguire lo *sniffing* dei pacchetti.

2. **Creazione di *Honeypot* centralizzati:** L'accumulo di record storici di posizioni geografiche all'interno di un unico database centralizzato in Cloud crea un bersaglio estremamente appetibile per attacchi informatici mirati. Un eventuale accesso non autorizzato (*Data Breach*) comprometterebbe istantaneamente la riservatezza delle traiettorie di un'intera popolazione urbana [14];

3. **Tracciamento e profilazione non autorizzata:** Anche in presenza di tecniche di pseudonimizzazione (sostituzione dei nomi utente con identificativi casuali), la letteratura scientifica dimostra che le traiettorie spazio-temporali sono impronte digitali uniche; incrociando pochi punti di sosta ricorrenti (es. l'incrocio di partenza mattutino e quello di arrivo serale), è matematicamente possibile risalire all'identità reale del soggetto connesso [15].

Mantenere l'intelligenza predittiva confinata sul dispositivo trasforma il *browser* web da mero visualizzatore passivo a cassaforte computazionale autonoma. Questa sovranità sul dato geografico non garantisce solamente una blindatura impenetrabile contro le violazioni della privacy, ma si riflette anche sulla robustezza operativa del sistema: un ITS decentralizzato è strutturalmente immune alle interruzioni di connettività di rete, continuando a garantire previsioni viarie accurate anche in scenari di totale isolamento o *black-out* delle comunicazioni.

1.5 Impatto Socio-Economico e Sostenibilità Infrastrutturale

L'implementazione degli ITS e di modelli predittivi non risponde esclusivamente a un'esigenza di ottimizzazione, ma si configura come un intervento

cruciale per la salvaguardia del tessuto urbano. La congestione del traffico è infatti un fenomeno in costante aggravamento a livello globale, spinto dalla rapida espansione demografica e dall'invecchiamento delle infrastrutture viarie.

Gli impatti socio-economici e ambientali derivanti da questa saturazione sono profondi e facilmente quantificabili. Secondo le analisi condotte da INRIX, società specializzata in telematica e *analytics* per la viabilità, nel 2017 la congestione stradale ha generato un costo indiretto per i pendolari statunitensi stimato in 305 miliardi di dollari [16]. Questa colossale perdita economica è direttamente imputabile allo spreco di carburante, al crollo della produttività legato alle enormi quantità di tempo perso nei ritardi e all'incremento vertiginoso dei costi logistici per il trasporto delle merci attraverso le aree congestionate.

In questo scenario, la capacità di anticipare la formazione del traffico attraverso algoritmi di *Machine Learning* permette alle amministrazioni pubbliche di adottare misure proattive (come la rimodulazione dinamica dei semafori o il reindirizzamento dei flussi), abbattendo drasticamente questi costi e rendendo la Smart City economicamente più vitale e sostenibile.

Tuttavia, affinché una soluzione tecnologica sia concretamente adottabile su scala metropolitana, essa non deve generare costi di gestione superiori ai benefici che intende portare. È proprio sotto questo profilo di sostenibilità finanziaria che la nostra scelta del paradigma architetturale Edge AI si rivela determinante.

1.5.1 L'Abbattimento dei Costi Infrastrutturali tramite Edge AI

Nelle architetture tradizionali basate su Cloud AI, l'intelligenza del sistema è centralizzata. Questo modello impone all'azienda fornitrice del servizio oneri finanziari insostenibili su larga scala. Per gestire migliaia di richieste di inferenza al secondo provenienti da tutti i veicoli o semafori della città, è necessario affittare, manutenere e raffreddare immense *sale server*. Ogni nuo-

vo utente che si connette al sistema richiede un incremento della potenza di calcolo centrale, costringendo l'erogatore a continui aggiornamenti hardware (incremento della spesa in conto capitale e della spesa operativa).

Attraverso il paradigma Edge Computing e l'adozione di WebAssembly, il nostro progetto ribalta completamente questa logica economica. Il *server* centrale perde la sua funzione di calcolo intensivo, declassandosi a semplice distributore di file statici. Il vero e proprio motore di inferenza (il modello addestrato) viene scaricato ed eseguito fisicamente all'interno del dispositivo del cliente (che sia lo *smartphone* dell'utente finale, il *computer* di bordo di un'automobile connessa o un nodo semaforico).

Dal punto di vista economico, questa architettura decentralizzata garantisce una **scalabilità orizzontale a costo marginale nullo**. I dispositivi si autogestiscono e il costo hardware e computazionale dell'elaborazione (l'energia elettrica e i cicli di CPU necessari per calcolare le previsioni) viene di fatto assorbito dall'hardware già di proprietà del cliente. L'infrastruttura centrale non subisce alcun sovraccarico computazionale, permettendo al sistema di scalare da cento a milioni di utenti senza richiedere l'acquisto di un singolo *server* addizionale.

1.5.2 Sinergia tra Privacy (GDPR) ed Economia Amministrativa

Come ampiamente discusso nella Sezione 1.4, l'elaborazione Client-Side garantisce il principio di *Privacy by Design*, in quanto le coordinate di localizzazione non lasciano mai il perimetro fisico del dispositivo dell'utente. Oltre all'evidente vantaggio etico e di sicurezza informatica, questa architettura si traduce in un ulteriore, drastico taglio dei costi amministrativi e legali.

Il Regolamento Generale sulla Protezione dei Dati (GDPR) impone sanzioni severissime (fino al 4% del fatturato globale) per la gestione impropria dei dati sensibili, costringendo le aziende che centralizzano i dati di geolocalizzazione a sostenere enormi spese per la *compliance* normativa. Tali spese includono l'assunzione di *Data Protection Officer* (DPO), l'esecuzio-

ne di costosi *audit* di sicurezza continui, l'implementazione di protocolli di crittografia dei database (*Data at Rest*) e la stipula di assicurazioni contro il rischio di *Data Breach*.

Eliminando alla radice la trasmissione e l'archiviazione centralizzata del dato spaziale, il sistema proposto si sottrae intrinsecamente a gran parte di queste gravose incombenze burocratiche. Non raccogliendo i dati personali, l'erogatore del servizio annulla il rischio di responsabilità legale connesso a potenziali furti di dati, dimostrando come la tutela della privacy (GDPR) e l'ottimizzazione economica possano coesistere sinergicamente all'interno della medesima architettura software.

Capitolo 2

Tecnologie di Sviluppo

Il presente capitolo si focalizza sulla descrizione dettagliata dell'ambiente operativo e dello *stack* tecnologico impiegato per la realizzazione dell'applicativo. L'obiettivo è delineare il percorso ingegneristico dei dati, partendo dalla loro acquisizione e strutturazione iniziale, passando per l'addestramento dei modelli predittivi, fino ad arrivare alla toolchain di compilazione avanzata necessaria per il *deployment* in ambiente Edge.

2.1 Inquadramento del Problema e Analisi del Dataset

Come ampiamente discusso nel capitolo precedente, la risoluzione delle criticità legate alla congestione stradale richiede un cambio di paradigma: dal semplice monitoraggio reattivo all'analisi predittiva e proattiva. Considerate le stringenti limitazioni fisiche e finanziarie che impediscono la costruzione indefinita di nuove arterie stradali, l'ottimizzazione dell'infrastruttura esistente passa necessariamente attraverso l'utilizzo e l'analisi di modelli di *Machine Learning* capaci di interpretare i flussi di dati in tempo reale.

È in questo esatto dominio applicativo che si inserisce l'analisi del nostro caso di studio, focalizzato sulla previsione dei flussi veicolari all'interno di nodi stradali critici. Per addestrare algoritmi supervisionati in grado di ap-

prendere pattern di traffico complessi, è stato necessario acquisire un *dataset* storico rappresentativo della mobilità urbana, caratterizzato da un elevato grado di granularità temporale, ospitato sulla piattaforma **Kaggle**. Kaggle rappresenta uno dei principali punti di riferimento a livello mondiale per la comunità accademica e industriale di *Data Science*, in quanto offre archivi validati e standardizzati per la sperimentazione algoritmica.

Il *dataset* originario, fornito sotto forma di file testuale strutturato (CSV), raccoglie misurazioni orarie continue registrate presso molteplici incroci stradali. Trattandosi di un problema di regressione su serie storiche, l'analisi esplorativa dei dati (*Exploratory Data Analysis* - EDA) si è rivelata fondamentale per comprendere la distribuzione del traffico e per estrarre vettori di *feature* significativi. La struttura tabulare del *dataset* si articola nelle seguenti variabili operative:

	DateTime	Junction	Vehicles	ID
0	2015-11-01 00:00:00	1	15	20151101001
1	2015-11-01 01:00:00	1	13	20151101011
2	2015-11-01 02:00:00	1	10	20151101021
3	2015-11-01 03:00:00	1	7	20151101031
4	2015-11-01 04:00:00	1	9	20151101041
5	2015-11-01 05:00:00	1	6	20151101051
6	2015-11-01 06:00:00	1	9	20151101061
7	2015-11-01 07:00:00	1	8	20151101071
8	2015-11-01 08:00:00	1	11	20151101081
9	2015-11-01 09:00:00	1	12	20151101091

Figura 2.1: Estratto iniziale del dataset grezzo.

2.1.1 Struttura Originaria e Feature Engineering

Come illustrato in figura 2.1, il *dataset* originario, fornito sotto forma di CSV, si presentava in una forma grezza ed essenziale. L'archivio era composto esclusivamente da quattro attributi fondamentali:

- `ID`: un identificativo progressivo per la singola registrazione;
- `DateTime`: una stringa di testo indicante la marca temporale esatta del rilevamento (formato anno-mese-giorno e ora);
- `Junction`: identificativo numerico dell'incrocio stradale monitorato;
- `Vehicles`: la variabile *target* continua che esprime il volume aggregato di veicoli transitati in quell'intervallo.

2.1.2 Strumenti di Ingestione e Manipolazione dei Dati

Per l'implementazione della *pipeline* di preparazione e ingegnerizzazione dei dati, la scelta tecnologica è ricaduta sulle due librerie fondamentali dell'ecosistema scientifico di *Python*: *Pandas* [17] e *NumPy* [18].

Pandas: Strutturazione ed Elaborazione High-Level

La libreria *Pandas* introduce nell'ambiente di sviluppo l'astrazione dei dati tabulari attraverso due strutture dati principali orientate agli oggetti: la *Series* e il *DataFrame* (matrice bidimensionale i cui vettori colonna sono costituiti da singole *Series*, condividendo un indice comune). Nel contesto della nostra ricerca, queste strutture hanno permesso di emulare il comportamento dei database relazionali direttamente all'interno della memoria centrale (*in-memory processing*).

	Junction	Vehicles	Year	Month	Day	Hour	DayOfWeek
0	1	15	2015	11	1	0	6
1	2	6	2015	11	1	0	6
2	3	9	2015	11	1	0	6
3	3	7	2015	11	1	1	6
4	1	13	2015	11	1	1	6
5	2	6	2015	11	1	1	6
6	2	5	2015	11	1	2	6
7	1	10	2015	11	1	2	6
8	3	5	2015	11	1	2	6
9	2	6	2015	11	1	3	6

Figura 2.2: Estratto finale del dataset dopo aver effettuato feature engineering.

Durante la fase di raffinamento del *dataset*, abbiamo sfruttato le funzionalità avanzate di *Pandas* per implementare molteplici operazioni cruciali di *feature engineering*:

- **Aggregazione e raggruppamento spaziale:** mediante l'utilizzo di logiche di tipo *Split-Apply-Combine* (espresse tramite l'operatore `groupby`), abbiamo isolato e analizzato in modo indipendente i flussi veicolari relativi a ciascun incrocio stradale (`Junction`), calcolando metriche statistiche aggregate descrittive;
- **Manipolazione temporale avanzata:** Attraverso i metodi di manipolazione temporale avanzata di *Pandas*, la singola colonna `DateTime` originale è stata de-strutturata e divisa in vettori di *feature* indipendenti. Questa operazione ha generato nuove colonne operative per l'algoritmo, nello specifico: `Year` (anno), `Month` (mese), `Day` (giorno) e `Hour` (ora del giorno). A queste si è aggiunta la derivazione fondamentale della variabile `DayOfWeek` (giorno della settimana), un parametro

architetturale essenziale per permettere al modello di differenziare autonomamente i pattern di mobilità feriale, dominati dai fenomeni di pendolarismo, da quelli festivi.

Pur offrendo un'interfaccia di programmazione ad alto livello estremamente intuitiva, *Pandas* risulterebbe computazionalmente inefficiente se basata esclusivamente sui tipi di dato nativi e sulle logiche di esecuzione di *Python*. Essendo quest'ultimo un linguaggio interpretato e a tipizzazione dinamica, l'iterazione nativa su milioni di registrazioni stradali (ad esempio tramite i classici cicli `for` iterando su liste standard) introdurrebbe un *overhead* insostenibile. Tale lentezza è dovuta ai continui controlli a tempo di esecuzione sul tipo di oggetto in memoria (*type checking*) e alle rigide limitazioni al parallelismo imposte dal *Global Interpreter Lock* (GIL) di *Python*.

L'eccezionale efficienza prestazionale della libreria deriva dal fatto che la sua architettura interna funge essenzialmente da *wrapper* (interfaccia di collegamento). Ogni operazione algebrica, di filtraggio condizionale o di aggregazione invocata dal programmatore in *Python* non viene materialmente calcolata dall'interprete, bensì viene delegata istantaneamente a *routine* interne pre-compilate scritte in *Cython* e in puro C. Questo approccio permette a *Pandas* di manipolare immensi vettori di dati bypassando completamente i colli di bottiglia del linguaggio originario, garantendo tempi di esecuzione nell'ordine delle frazioni di secondo e unificando l'espressività della prototipazione rapida con le prestazioni assolute del calcolo a basso livello (*bare-metal performance*).

NumPy: Calcolo Vettoriale e Strutture di Memoria Contigue

Alla base del funzionamento di *Pandas*, e di gran parte delle librerie di modellazione predittiva come *Scikit-learn*, risiede *NumPy*. Questa libreria architetturale ridefinisce il calcolo numerico in *Python* introducendo l'oggetto *ndarray* (*N-dimensional array*), un contenitore multidimensionale di elementi omogenei a dimensione fissa.

A differenza delle liste native di *Python*, che memorizzano puntatori a oggetti sparsi nella memoria dinamica (*heap*) introducendo un pesante sovraccarico computazionale (*overhead*), un array di *NumPy* alloca un blocco di memoria rigidamente contiguo. Tutti gli elementi condividono lo stesso tipo di dato primitivo (es. interi a 64 bit o floating-point a 32 bit). Questa disposizione a livello hardware sblocca due vantaggi fondamentali:

1. **Località dei dati e *Cache Looping*:** l'allocazione contigua massimizza lo sfruttamento della memoria cache del processore. Quando la CPU richiede un dato, i vettori adiacenti vengono pre-caricati nei registri veloci, riducendo drasticamente i tempi di accesso alla memoria RAM;

2. **Vettorizzazione delle operazioni (*Vectorization*):** consente di applicare funzioni matematiche ed element-wise su interi blocchi di dati bypassando i cicli iterativi espliciti del codice interpretato. Le operazioni vengono tradotte internamente in codice C compilato ed eseguite tramite istruzioni SIMD (*Single Instruction, Multiple Data*) a livello di microprocessore.

Nel nostro impianto elaborativo, abbiamo sfruttato le logiche di *broadcasting* di *NumPy* per normalizzare istantaneamente i vettori delle funzionalità e per convertire i *DataFrame* di *Pandas* in matrici dense di tipo `float32`. Questo specifico formato rappresenta lo standard di immissione richiesto dai framework basati su modelli ad albero come *XGBoost* e *LightGBM*, ottimizzando il passaggio dei dati attraverso i canali di comunicazione interni del sistema durante la fase di addestramento.

2.2 L'Ecosistema di Addestramento e Ottimizzazione

La fase di modellazione predittiva e addestramento algoritmico è stata interamente sviluppata in *Python*. Negli ultimi anni, *Python* si è consolida-

to come standard nel mondo della ricerca scientifica e dell'industria legata all'Intelligenza Artificiale e al *Machine Learning*. Tale primato non deriva unicamente dalla sua sintassi pulita ad alto livello, ma soprattutto dalla sua peculiare evoluzione storica e dal suo vastissimo ecosistema di librerie esterne.

2.2.1 Il Linguaggio Python: Evoluzione e Paradigma Architetturale

Ideato alla fine degli anni Ottanta da Guido van Rossum presso il *Centrum Wiskunde & Informatica* (CWI) nei Paesi Bassi e rilasciato pubblicamente nel 1991, *Python* [19] è nato con un obiettivo progettuale ben preciso: massimizzare la leggibilità del codice e la produttività dello sviluppatore. A differenza dei linguaggi compilati a basso livello, si tratta di un linguaggio interpretato, a tipizzazione dinamica e multi-paradigma (supporta nativamente la programmazione orientata agli oggetti, imperativa e funzionale).

Inizialmente impiegato come semplice linguaggio di *scripting* per l'automazione di sistema, *Python* ha subito una trasformazione radicale nei primi anni Duemila, quando la comunità scientifica ha iniziato a ricercare un'alternativa *open-source* ai costosi software proprietari di calcolo numerico (come MATLAB). Il vero punto di svolta è coinciso con la risoluzione del cosiddetto *two-language problem* (il problema dei due linguaggi).

Storicamente, gli ingegneri informatici e i ricercatori erano costretti a dividere il proprio flusso di lavoro: utilizzavano linguaggi flessibili ma lenti per la prototipazione e l'analisi esplorativa, per poi dover riscrivere interamente gli algoritmi in linguaggi complessi ma estremamente veloci (come C, C++ o Fortran) per la fase di esecuzione in produzione.

Python ha risolto questo problema ponendosi come **linguaggio "collante"** (*glue language*). L'implementazione di riferimento del linguaggio, denominata *CPython*, è infatti scritta in C e offre un'API (*Application Programming Interface*) nativa che permette di estendere il linguaggio integrando moduli compilati a basso livello.

Questa architettura ibrida è il segreto delle prestazioni dell'ecosistema odierno. Molti dei *framework* utilizzati in questo progetto di tesi per l'algebra lineare e il *Machine Learning*, pur esponendo interfacce semplici e intuitive in *Python*, sono in realtà scritti e compilati internamente in linguaggi a prestazioni native. *Python* si limita a gestire la logica di controllo ad alto livello e a orchestrare il flusso dei dati in memoria, delegando i pesanti calcoli matematici (come le moltiplicazioni tra matrici o l'attraversamento degli alberi decisionali) alle librerie sottostanti in C/C++. Questo paradigma unisce la rapidità di prototipazione richiesta dai *Data Scientist* alle massime prestazioni di calcolo parallelo richieste dall'hardware moderno.

Limiti del Modello Singolo e necessità degli Ensemble

Sebbene gli alberi decisionali godano di un'eccellente interpretabilità logica e non richiedano la standardizzazione dei dati di input (essendo invarianti alle trasformazioni monotone), presentano un grave difetto strutturale: un'intrinseca instabilità che li rende proni all'**overfitting**.

Se lasciato crescere senza vincoli di profondità (*unpruned tree*), un singolo albero tenderà a creare un nodo foglia per quasi ogni osservazione del *training set*, memorizzando il rumore statistico ed esibendo un'altissima varianza (*High Variance*). Una minima variazione nei dati di addestramento può generare una predizione completamente differente. Per limitare questa debolezza architetturale, la ricerca moderna si concentra sulle tecniche di aggregazione (*Ensemble Learning*), che combinano centinaia di predittori deboli per formare un unico modello robusto, sfruttando paradigmi avanzati come il *Bagging* e il *Boosting*.

Modello	**Famiglia Architetturale**	**Approccio Matematico**
Linear Regression	Parametrico Lineare	Ottimizzazione dei minimi quadrati ordinari.
k-NN	Non Parametrico	Calcolo delle distanze euclidee nello spazio delle *feature*.
SVR	*Kernel-based*	Massimizzazione del margine entro un tubo di insensibilità ϵ.
MLP	Connessionista	Approssimazione non lineare tramite strati di neuroni artificiali.
Random Forest	*Ensemble Bagging*	Media parallela di alberi decisionali indipendenti decorrelati.
XGBoost	*Ensemble Boosting*	Ottimizzazione sequenziale del gradiente con regolarizzazione.
LightGBM	*Ensemble Boosting*	Crescita dell'albero con campionamento ottimizzato.

Tabella 2.1: Classificazione e tassonomia strutturale degli algoritmi di Machine Learning investigati.

2.2.2 Tassonomia dei Modelli Predittivi Indagati

Al fine di validare scientificamente la bontà della soluzione proposta e dimostrare la superiorità prestazionale del modello di riferimento, è stata condotta una campagna di *benchmarking*. Come riassunto nella Tabella 2.1, l'analisi non si è limitata ai soli modelli ad albero, ma ha incluso paradigmi matematici radicalmente differenti. Questa eterogeneità permette di valutare l'efficienza di ciascun approccio sia in termini di accuratezza statistica, sia per la sostenibilità computazionale in vista dello sviluppo su dispositivi periferici.

I Modelli Basati su Alberi Decisionali

I modelli appartenenti a questa classe scompongono lo spazio delle funzionalità mediante un partizionamento ricorsivo. All'interno di questa famiglia, sono state investigate tre diverse filosofie di aggregazione:

- ***Random Forest* [20]:** fonda la sua architettura sul principio del *bagging*. Il modello addestra in parallelo una moltitudine di alberi su campioni casuali estratti dal *dataset* originale. Per garantire la decorrelazione tra i singoli decisori, applica il *feature bagging*, selezionando casualmente un sottoinsieme limitato di variabili a ogni diramazione nodale. La previsione finale è calcolata come media matematica degli output individuali: questa aggregazione di stimatori indipendenti permette di abbattere drasticamente la varianza intrinseca degli alberi singoli.

- ***XGBoost* (*eXtreme Gradient Boosting*) [21]:** a differenza del *bagging*, implementa una strategia sequenziale in cui ogni nuovo albero viene addestrato per approssimare il gradiente della funzione di perdita calcolata sui residui d'errore dell'intero *ensemble* precedente. L'aspetto matematicamente rivoluzionario di questo paradigma è l'uso di un'approssimazione di Taylor del secondo ordine (utilizzando gradienti ed hessiane) per l'ottimizzazione dell'obiettivo, congiunta all'inclusione di rigorose metriche di regolarizzazione ($L1$ ed $L2$) direttamente nella funzione di costo. Questo approccio penalizza i nodi foglia con pesi eccessivi, controllando la complessità strutturale dell'algoritmo e prevenendo attivamente l'*overfitting*.

- ***LightGBM* (*Light Gradient Boosting Machine*) [22]:** rappresenta un'evoluzione prestazionale del *boosting* classico. Mentre gli algoritmi tradizionali fanno crescere gli alberi simmetricamente livello per livello (*level-wise*), questo approccio espande la topologia adottando una strategia *best-first* (*leaf-wise*): viene diviso esclusivamente il nodo foglia che garantisce la massima riduzione della funzione di perdita

globale, indipendentemente dalla sua profondità. Sebbene generi alberi fortemente asimmetrici e richieda un rigido controllo sul parametro `max_depth` per evitare l'iper-adattamento, ottimizza in modo drastico la velocità di convergenza strutturale.

I Modelli Non Basati su Alberi

Per verificare oggettivamente l'idoneità della struttura ad albero, l'analisi è stata estesa a modelli operanti su presupposti geometrici, statistici e analitici eterogenei:

- ***Linear Regression* (Regressione Lineare) [5]:** costituisce la *baseline* fondamentale di confronto. Trattandosi di un modello parametrico, assume a priori una relazione strettamente lineare tra le variabili esplicative (*input*) e la variabile dipendente (*target*). L'algoritmo calcola i coefficienti ottimali minimizzando la somma dei quadrati dei residui tramite il metodo OLS (*Ordinary Least Squares*). Pur offrendo un costo computazionale pressoché nullo e una spiegabilità matematica immediata, questo modello evidenzia severi limiti nel catturare le dinamiche non lineari e i picchi anomali tipici delle serie storiche del traffico urbano.

- ***k-NN* (*k-Nearest Neighbors*) [23]:** è un algoritmo non parametrico di tipo *lazy learning*. A differenza degli altri approcci, non prevede una fase di addestramento formale volta alla creazione di un'astrazione matematica: l'algoritmo si limita a memorizzare l'intero *training set* nello spazio delle *feature*. In fase di inferenza, la predizione viene calcolata come media (eventualmente pesata) dei valori dei k vicini più prossimi, determinati tramite una specifica metrica di distanza. Questa dipendenza diretta dai dati storici comporta un'impronta di memoria e una latenza di inferenza che crescono linearmente con le dimensioni del *dataset*.

- ***SVR (Support Vector Regression)* [24]:** estende ai domini continui il solido impianto teorico delle macchine a vettori di supporto. L'obiettivo matematico è determinare una funzione (un iperpiano) che approssimi i valori reali mantenendo l'errore entro un margine di tolleranza predefinito (il tubo ϵ), penalizzando esclusivamente i residui che ricadono all'esterno di tale fascia. Attraverso l'applicazione del cosiddetto *Kernel Trick*, l'SVR proietta implicitamente i dati in uno spazio vettoriale ad alta dimensionalità, permettendo di risolvere relazioni intrinsecamente complesse, seppur al costo di un oneroso *tuning* degli iperparametri (C, γ, ϵ).

- ***MLP (Multi-Layer Perceptron)* [25]:** rappresenta la potenza dell'approccio connessionista basato sulle reti neurali artificiali. Si tratta di un'architettura *feed-forward* strutturata in un livello di *input*, uno o più *hidden layer* (strati nascosti) e un nodo di *output*. L'apprendimento dei pesi è guidato dall'algoritmo di retropropagazione dell'errore (*backpropagation*) e ottimizzato tramite la discesa del gradiente. L'utilizzo di funzioni di attivazione non lineari (come *ReLU* o *Sigmoide*) garantisce alla rete un'altissima flessibilità espressiva, ma richiede ingenti risorse hardware in fase di addestramento e un'attenta regolarizzazione per prevenire l'assimilazione del rumore statistico (*overfitting*).

2.2.3 Librerie di Modellazione Predittiva

Per tradurre in codice i paradigmi teorici appena analizzati, è essenziale affidarsi a librerie software consolidate, manutenute dalla comunità scientifica e ottimizzate per lo sfruttamento dell'hardware sottostante. Le tecnologie selezionate per lo strato applicativo sono:

- ***Scikit-learn* [26]:** rappresenta l'ecosistema *open-source* fondamentale per il *Machine Learning* in *Python*. La sua forza risiede in un'API rigorosamente unificata (*Estimator*, *Predictor*, *Transformer*). Nel contesto di questo progetto, non ha fornito solamente le implementazioni

algoritmiche dei modelli di *baseline* (LR, k-NN, SVR, Random Forest), ma ha costituito l'infrastruttura logica per l'intera *pipeline*, gestendo la normalizzazione statistica delle *feature* e il calcolo delle metriche di validazione.

- ***XGBoost* [21]:** è il *framework* di riferimento per l'implementazione pratica del *gradient boosting*. Dal punto di vista prettamente architetturale, *XGBoost* memorizza i dati in blocchi in-memory preordinati (*Block Structure*), permettendo ai diversi *thread* della CPU di ricercare i punti di taglio ottimali (*split finding*) in modo totalmente asincrono. L'integrazione di strategie di *cache-aware access* ottimizza l'allocazione della RAM, garantendo tempi di esecuzione eccellenti.

- ***LightGBM* [22]:** sviluppato dai ricercatori Microsoft per implementare l'approccio *leaf-wise*, nasce con il preciso scopo di superare i colli di bottiglia della memoria. Il *framework* integra due innovazioni: il *Gradient-based One-Side Sampling* (GOSS), che concentra la potenza di calcolo escludendo le istanze con gradienti prossimi allo zero, e l'*Exclusive Feature Bundling* (EFB), un algoritmo capace di accorpare *feature* sparse riducendo drasticamente la dimensionalità dei tensori caricati in memoria.

Un aspetto metodologico trasversale all'utilizzo di tali librerie riguarda la gestione della stocasticità. Molti degli algoritmi citati (in particolare i metodi *Ensemble*) integrano nativamente procedure pseudo-casuali, come l'inizializzazione dei pesi, il campionamento dei dati o la selezione *randomica* delle *feature* per i tagli degli alberi. Al fine di inibire questa aleatorietà intrinseca e garantire la **totale riproducibilità scientifica** degli esperimenti, il parametro di inizializzazione (*seed* o `random_state`) è stato globalmente vincolato a una costante numerica predefinita (impostata al valore 42). Questa forzatura architetturale assicura che il codice sorgente risulti strettamente deterministico, producendo i medesimi risultati statistici a ogni esecuzione.

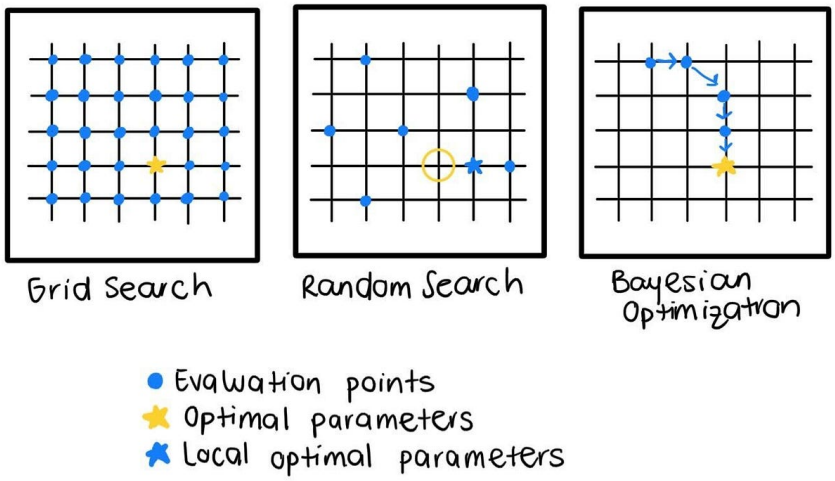

Figura 2.3: Grafici di come i metodi di ricerca arrivano all'ottimizzazione degli iperparametri Fonte:[27]

2.2.4 Analisi Comparativa: Grid Search, Random Search e Optuna

È essenziale confrontare il paradigma operativo dei metodi di *tuning* classici, le cui dinamiche di esplorazione dello spazio delle *feature* sono visualizzate in Figura 2.3.

I metodi di ricerca tradizionali integrati nei classici *framework*, come la *Grid Search* o la *Random Search*, risultano computazionalmente proibitivi per modelli complessi. Tali approcci non posseggono "memoria" degli esperimenti passati, finendo per sprecare enormi risorse di calcolo su configurazioni evidenti di scarso successo.

- ***Grid Search* (Ricerca Esaustiva)[28]:** Rappresenta l'approccio più deterministico ma computazionalmente più oneroso. L'algoritmo definisce una griglia discreta di valori per ciascun iperparametro e valuta il modello su ogni singola combinazione possibile (prodotto carte-

siano). Pur garantendo l'individuazione dell'ottimo globale all'interno dello spazio discretizzato, questo metodo soffre all'aggiunta di ogni nuovo parametro, in quanto il numero di addestramenti necessari cresce in modo esponenziale, rendendo l'approccio inapplicabile su modelli complessi e multi-parametro.

- ***Random Search* (Ricerca Casuale)[29]:** Per superare la rigidità esplorativa della griglia fissa, questo approccio campiona configurazioni in modo stocastico all'interno dei domini definiti. La letteratura scientifica dimostra che la ricerca casuale è intrinsecamente più efficiente di quella esaustiva: esplorando valori continui e non pre-spaziati, ha una probabilità nettamente maggiore di intercettare combinazioni ad alte prestazioni con un numero inferiore di iterazioni. Tuttavia, essendo un processo "senza memoria", l'algoritmo non impara dai propri errori, rischiando di sprecare risorse computazionali su aree dello spazio chiaramente sub-ottimali.

- ***Ottimizzazione Bayesiana (Optuna)*[30]:** Per superare questo ostacolo prestazionale, il nostro ecosistema di sviluppo ha integrato ***Optuna***, un *framework* di nuova generazione per l'ottimizzazione automatizzata, basato sul paradigma *Define-by-Run*. A differenza dei metodi di forza bruta, *Optuna* implementa tecniche di **Ottimizzazione Bayesiana**. L'architettura interna utilizza algoritmi di campionamento avanzati, specificamente il *Tree-structured Parzen Estimator* (TPE). Invece di procedere casualmente, il *framework* costruisce dinamicamente un modello probabilistico delle prestazioni passate; ad ogni iterazione, stima matematicamente quali regioni dello spazio degli iperparametri abbiano la maggiore probabilità di ridurre l'errore, concentrando lì le misurazioni successive.

  Un'ulteriore eccellenza architetturale di *Optuna* è il meccanismo di *pruning*. Attraverso algoritmi di valutazione intermedia (come la *Median Stopping Rule*), la libreria monitora le curve di apprendimento del mo-

dello in tempo reale: se durante i primi step di addestramento una determinata configurazione mostra performance nettamente inferiori alla media storica, *Optuna* interrompe forzatamente l'esecuzione del *thread* (*early stopping*), scartando il percorso sub-ottimale e liberando immediatamente le risorse computazionali per l'esplorazione di set di iperparametri più promettenti. Ciò ha consentito di estrarre le massime prestazioni predittive dal nostro *dataset* in una frazione del tempo operativo necessario.

Fondamenti Matematici: Ottimizzazione Bayesiana e TPE

Per comprendere appieno l'efficienza del *framework*, è opportuno formalizzare il problema matematico sottostante. L'obiettivo dell'ottimizzazione degli iperparametri è trovare un set di parametri θ^* che minimizzi la funzione obiettivo (o funzione di costo) $f(\theta)$, la quale nel nostro caso rappresenta l'errore di validazione del modello sul traffico veicolare:

$$\theta^* = \text{argmin}_{\theta \in \Theta} f(\theta) \tag{2.1}$$

dove Θ rappresenta lo spazio di ricerca multidimensionale definito a priori. Poiché la valutazione di $f(\theta)$ richiede l'addestramento completo del modello, essa è considerata una funzione *black-box* estremamente costosa in termini computazionali.

L'Ottimizzazione Bayesiana affronta questo problema applicando il Teorema di Bayes per aggiornare costantemente la probabilità a posteriori (*posterior probability*) di una certa configurazione di iperparametri, data la storia delle misurazioni precedenti. La formula fondamentale si esprime come:

$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)} \tag{2.2}$$

dove $y = f(\theta)$ è il risultato osservato (l'errore), $P(\theta)$ è la distribuzione a priori (*prior*) degli iperparametri e $P(y|\theta)$ è la verosimiglianza (*likelihood*).

Invece di modellare direttamente la probabilità a posteriori $P(y|\theta)$ tramite Processi Gaussiani (GP), il TPE modella la densità di probabilità $P(\theta|y)$

suddividendo le osservazioni storiche in due distribuzioni distinte, basandosi su un valore di soglia y^* (tipicamente un quantile delle misurazioni passate):

$$P(\theta|y) = \begin{cases} l(\theta) & \text{se } y < y^* \\ g(\theta) & \text{se } y \geq y^* \end{cases} \tag{2.3}$$

In questa formulazione:

- $l(\theta)$ è la densità di probabilità stimata degli iperparametri che hanno prodotto risultati eccellenti (errore inferiore alla soglia y^*);
- $g(\theta)$ è la densità di probabilità associata ai risultati sub-ottimali.

Ad ogni nuova iterazione, l'algoritmo non compie una scelta casuale, ma massimizza matematicamente la funzione di *Expected Improvement* (EI), che nel caso del TPE si riduce alla massimizzazione del rapporto $\frac{l(\theta)}{g(\theta)}$. In termini pratici, il *framework* seleziona campioni che hanno un'alta probabilità di trovarsi sotto la distribuzione "buona" $l(\theta)$ e una bassa probabilità di trovarsi sotto quella "cattiva" $g(\theta)$.

Questa rigorosa base probabilistica è ciò che consente di esplorare lo spazio iper-dimensionale in modo mirato, abbattendo drasticamente i tempi di calcolo per il *forecasting* del traffico.

2.3 Tecnologie di Compilazione per l'Edge Computing

Una volta addestrato e validato il modello ottimale all'interno dell'ambiente *Python*, è sorta la necessità di svincolarlo da tale ecosistema. *Python* è un linguaggio interpretato, strutturalmente inadatto per l'esecuzione diretta ad alte prestazioni all'interno di *browser* web o dispositivi periferici a risorse limitate. Per concretizzare l'architettura decentralizzata, abbiamo strutturato una *pipeline* di compilazione *Ahead-of-Time* (AOT) basata sulle seguenti tecnologie.

2.3.1 Razionale Architetturale: L'Adozione del Paradigma Client-Side

La transizione dall'ambiente di sviluppo alla messa in produzione (*deployment*) impone una scelta architetturale fondamentale. L'approccio industriale standard prevede un paradigma *Client-Server*: il modello di *Machine Learning* viene ospitato su un server Cloud centrale, esposto tramite API (RESTful o gRPC), e il client si limita a inviare i dati di input attendendo la risposta via rete.

Nel contesto specifico del nostro applicativo per il *forecasting* del traffico urbano, abbiamo deliberatamente scartato questo approccio centralizzato in favore di un paradigma puramente *Client-Side* (Edge AI). Questa scelta è motivata da tre vantaggi strutturali:

1. **Abbattimento della Latenza di Rete:** in un'architettura Cloud, il tempo di inferenza è dominato dal *Round-Trip Time* (RTT) della rete pubblica, cioè il tempo totale impiegato da un pacchetto di dati per viaggiare dal computer di origine a un dispositivo di destinazione (come un server) e per farvi ritorno. Spostando il motore di inferenza direttamente sul dispositivo dell'utente finale, eliminiamo del tutto il collo di bottiglia del trasporto dati, garantendo risposte in tempo reale indipendenti dalla qualità della connessione;

2. **Privacy by Design e Isolamento del Dato:** richiedere previsioni sul traffico implica la gestione di coordinate spaziali e marcatori temporali. Elaborando l'inferenza localmente all'interno della *sandbox* del browser, il dato sensibile dell'utente non lascia mai il dispositivo, azzerando i rischi di intercettazione (*Man-in-the-Middle*) e sollevando l'infrastruttura da complessi oneri normativi (GDPR);

3. **Scalabilità Orizzontale a Costo Zero:** nel modello *Client-Server*, migliaia di richieste simultanee richiedono costosi bilanciatori di carico e server ad alte prestazioni. Nel nostro paradigma Edge, il costo computazionale è interamente distribuito: ogni nuovo utente che accede

all'applicativo fornisce il proprio hardware (la CPU del proprio smartphone o PC) per eseguire il calcolo, rendendo il sistema intrinsecamente scalabile senza costi infrastrutturali aggiuntivi.

4. **Resilienza Offline:** Una volta che l'applicativo ha scaricato il motore di inferenza al primo avvio, l'utente può continuare a generare previsioni del traffico anche in totale assenza di connessione internet;

Attraverso il caricamento del motore di inferenza direttamente all'interno dell'ambiente *sandbox* locale del dispositivo dell'utente, la barriera di protezione si sposta sui confini fisici del client stesso. Nel nostro applicativo l'identificativo fisico dell'incrocio stradale esaminato (*Junction*) e i relativi descrittori temporali vengono inseriti direttamente nella memoria protetta del modulo in esecuzione locale.

I vantaggi derivanti dal mantenimento della posizione all'interno dell'hardware locale sono strutturali:

- **Anonimato Architetturale Nativo:** Non vi è alcuna necessità di cifrare, mascherare o anonimizzare il dato di localizzazione prima dell'invio, per il semplice motivo che il dato **non viene mai trasmesso**. La rete non viene utilizzata per ricevere le *feature* di input, azzerando di fatto il rischio di intercettazione di rete;

- **Scomparsa del Database Centralizzato:** Mancando la necessità di raccogliere flussi storici in tempo reale in Cloud per scopi di calcolo estemporaneo, si elimina alla radice la creazione di archivi massivi vulnerabili a infiltrazioni esterne;

- **Conformità Normativa Semplificata:** Dal punto di vista legale e amministrativo, l'elaborazione locale solleva l'infrastruttura centrale dalle complesse responsabilità connesse al trattamento e alla conservazione dei dati personali identificabili su server di terze parti.

2.3.2 Il Paradigma delle API

Il termine **API** ricorre in contesti operativi differenti, rendendo opportuna una sua formalizzazione architetturale. In informatica, un'API rappresenta un set di definizioni, protocolli e strumenti che permette a due applicazioni software distinte di comunicare tra loro. Funge da collegamento di interfaccia e da livello di astrazione: espone le funzionalità di un sistema nascondendone la complessità implementativa sottostante.

All'interno del nostro ecosistema di sviluppo, le API si declinano in due categorie fondamentali, le quali definiscono l'intera architettura del progetto:

- **API di Libreria (Interne):** Operano all'interno della medesima macchina e del medesimo processo di memoria. Sono i punti di accesso esposti dal codice sorgente per permettere a moduli diversi di interagire. Ne è un esempio l'API unificata di *Scikit-learn* (basata sui metodi `fit` e `predict`), o l'API nativa di *CPython*, che permette al codice interpretato ad alto livello di invocare istantaneamente le routine matematiche pre-compilate in linguaggio C. In questo caso, la comunicazione è istantanea e non subisce alcun ritardo di latenza esterna.

- **Web API (Di Rete):** Operano su macchine fisicamente distinte, sfruttando la rete internet per la comunicazione [31]. In un'architettura tradizionale di *Machine Learning* in Cloud, il modello addestrato viene "esposto" tramite Web API (tipicamente secondo lo standard **RESTful** o tramite protocolli binari come **gRPC**). Affinché avvenga l'inferenza, il dispositivo client deve impacchettare i dati di input in un formato strutturato (es. JSON), inviarli tramite protocollo HTTP al server remoto, e attendere che il server risponda con la medesima formattazione.

La transizione verso il paradigma *Edge AI* e l'utilizzo di *WebAssembly* consiste esattamente nell'eliminare la dipendenza dalle lente e vulnerabili Web API di rete, trasformando il modello da un servizio remoto a una libreria interrogabile localmente tramite API interne al *browser*.

2.3.3 Interoperabilità e Serializzazione: Lo Standard ONNX

Il primo ostacolo tecnico verso la distribuzione Edge è la frammentazione dei *framework*. Un modello esportato nativamente da un modello ad albero non è compatibile con gli interpreti web. Per risolvere questo problema, abbiamo analizzato lo standard **ONNX** (*Open Neural Network Exchange*) [32].

Sviluppato originariamente da Microsoft e Facebook, ONNX non è un semplice formato di salvataggio, ma una vera e propria **Rappresentazione Intermedia (IR)** standardizzata. Dal punto di vista architetturale, ONNX si basa su *Protocol Buffers* (Protobuf), un meccanismo di serializzazione binaria dei dati altamente efficiente e indipendente dalla piattaforma.

Quando convertiamo il nostro modello in formato ONNX, la struttura dell'algoritmo viene tradotta in un Grafo Computazionale Diretto Aciclico (DAG). All'interno di questo grafo:

- I **Nodi** rappresentano gli operatori matematici standardizzati (nel caso degli alberi decisionali: operatori di diramazione logica, confronti tra soglie, e operatori di aggregazione delle foglie);
- Gli **Archi** (*Edges*) rappresentano il flusso dei tensori di dati multidimensionali che viaggiano da un operatore all'altro.

Questa profonda astrazione matematica disaccoppia definitivamente la logica predittiva del nostro modello dal codice *Python* che lo ha generato. Ottenendo un grafo computazionale universalmente leggibile, abbiamo gettato le basi per poter instradare l'algoritmo verso i successivi compilatori a basso livello.

2.3.4 Compilazione AOT dei Modelli: Treelite e tl2cgen

Se da un lato la struttura matematica di un singolo albero decisionale è facilmente interpretabile e computazionalmente leggera, dall'altro l'attraver-

samento sequenziale di foreste *Ensemble* composte da migliaia di alberi (come nel caso di XGBoost o LightGBM) comporta sfide prestazionali non banali. Quando viene eseguito tramite un interprete standard o all'interno del *runtime* nativo di *Python*, il calcolo dell'inferenza introduce un *overhead* di memoria e ritardi di latenza che risultano inaccettabili in scenari di *real-time forecasting*, ovvero aggiornamenti istantanei su dispositivi periferici.

Per svincolare la fase di esecuzione dalla pesante infrastruttura software necessaria per l'addestramento, il nostro ecosistema ha integrato ***Treelite*** [33]. Sviluppata originariamente dalla *Distributed (Deep) Machine Learning Community* (DMLC), lo stesso consorzio di ricerca open-source che ha dato il via a XGBoost, Treelite nasce per risolvere in modo specifico il cosiddetto "collo di bottiglia del *deployment*".

Tradizionalmente, per eseguire una previsione, il server deve caricare in memoria l'intero motore del *framework* di *Machine Learning*, comprese le librerie di ottimizzazione dei gradienti che in fase di inferenza sono totalmente inutili. Treelite stravolge questo paradigma introducendo il concetto di **compilazione del modello** (*Model Compilation*). Invece di trattare il modello come un file di dati da far leggere a un interprete, Treelite lo tratta come se fosse un programma da compilare.

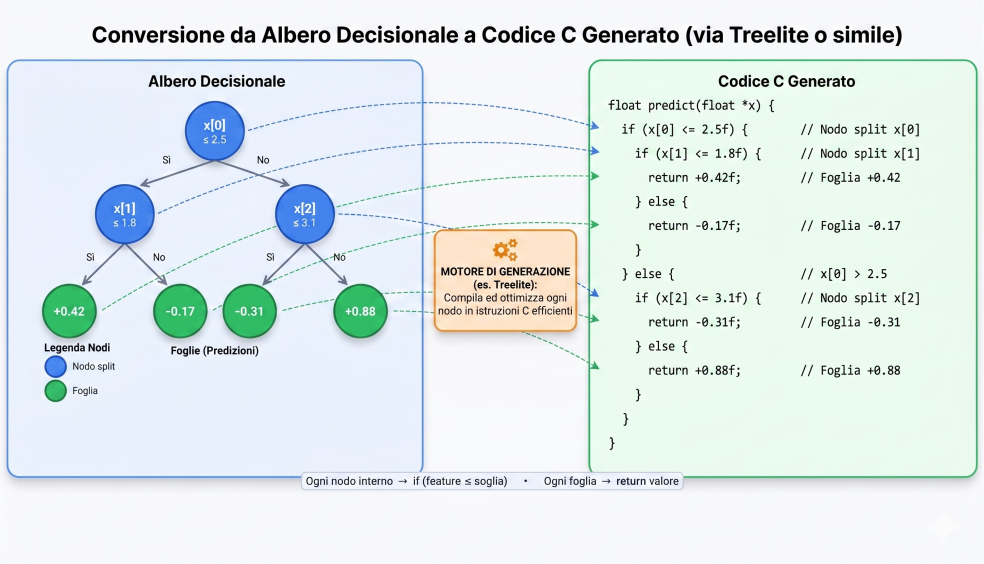

Figura 2.4: Trasformazione di un modello ad albero in codice C tramite Treelite.

All'interno dell'ecosistema Treelite, il motore materiale della trasformazione è il modulo ***tl2cgen*** (*TreeLite 2 C Generator*)[34]. Questo componente acquisisce la rappresentazione interna del modello *Ensemble* e la converte, istruzione per istruzione, in puro codice sorgente C nativo. Le ramificazioni logiche dell'albero vengono tradotte in strutture di controllo annidate (blocchi `if-else`) come illustrato in figura 2.4. Questa transcodifica abilita l'utilizzo della compilazione AOT.

I vantaggi derivanti dalla traduzione dell'algoritmo in codice C sono molteplici:

1. **Ottimizzazione del Compilatore:** il codice generato può essere eseguito dai compilatori C standard (come GCC o Clang);

2. **Branch Prediction:** l'algoritmo sfrutta nativamente il meccanismo di predizione delle diramazioni integrato nell'hardware delle CPU moderne;

3. **Isolamento e Leggerezza:** compilando il sorgente C, si genera una libreria condivisa dinamica estremamente compatta, bypassando totalmente la necessità di un interprete durante l'esecuzione.

2.3.5 Il Target di Esecuzione: WebAssembly ed Emscripten

L'ultimo anello critico della *pipeline* architetturale consiste nel rendere il codice C, generato e ottimizzato da *tl2cgen*, eseguibile all'interno del *browser* web dell'utente finale. Storicamente, le applicazioni web ad alte prestazioni erano costrette a fare affidamento su *plug-in* di terze parti o a riscrivere algoritmi in *JavaScript*, un linguaggio interpretato non progettato per il calcolo matematico intensivo. La tecnologia abilitante che ha risolto questo paradigma è ***WebAssembly*** **(WASM)** [35].

Architettura di WebAssembly

WebAssembly non è un linguaggio di programmazione tradizionale, bensì un formato di istruzioni binario e una *Instruction Set Architecture* (ISA) virtuale basata su *stack*. Riconosciuto ufficialmente dal W3C come il quarto linguaggio nativo del web (accanto a HTML, CSS e JavaScript), WASM è progettato per essere il *target* di compilazione portabile per linguaggi a basso livello e alte prestazioni come C, C++ e Rust.

Rispetto all'esecuzione standard in JavaScript, che richiede la lettura del testo sorgente, l'analisi sintattica e la compilazione *Just-In-Time* (JIT) all'avvio dell'applicazione, il binario `.wasm` è già pre-compilato. Questo garantisce tempi di decodifica quasi istantanei, minimizzando il fenomeno del *Cold Start* (il tempo di latenza al primo avvio dell'applicazione). Inoltre, WASM viene eseguito all'interno di una *sandbox* isolata dalla memoria di sistema, garantendo i massimi standard di sicurezza richiesti dai moderni *browser*.

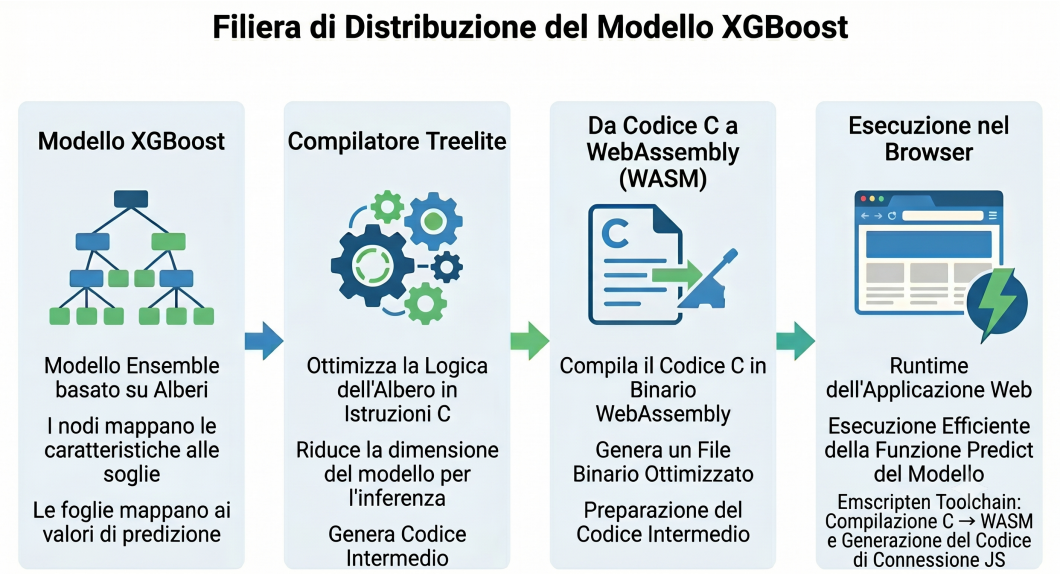

Figura 2.5: Toolchain di compilazione, partendo da un modello ad albero fino ad arrivare alla compilazione Emscripten.

La Toolchain di Compilazione: Emscripten

Per effettuare la transcodifica dal codice sorgente C generato da *Treelite* al formato binario `.wasm`, abbiamo impiegato ***Emscripten*** (*emsdk*) [36]. Emscripten non è un semplice traduttore di sintassi, ma una *toolchain* di compilazione basata sull'infrastruttura industriale **LLVM** (*Low Level Virtual Machine*).

Il processo di transcodifica si articola in due fasi: inizialmente, il *frontend* Clang analizza il codice C e lo converte nella Rappresentazione Intermedia di LLVM (LLVM IR). Successivamente, il *backend* dedicato converte questa IR nel formato binario `.wasm`. La potenza architetturale di Emscripten risiede nella sua capacità di emulare un intero strato operativo compatibile con POSIX. Esso fornisce implementazioni in *JavaScript* e WebAssembly delle librerie standard C (*libc* e *libcxx*), permettendo al codice matematico del modello di girare sul web esattamente come se stesse girando nativamente su un sistema operativo *desktop*.

Capitolo 3

Implementazione Software e Validazione Sperimentale

L'obiettivo di questo capitolo è illustrare nel dettaglio l'intero ciclo di vita dello sviluppo del software predittivo, partendo dall'ingestione e manipolazione dei dati grezzi, attraversando le fasi di addestramento e ottimizzazione degli algoritmi, fino ad arrivare alla compilazione e all'integrazione del motore di inferenza all'interno dell'applicativo web periferico.

Verranno inizialmente descritte le metodologie di strutturazione dell'ambiente di lavoro, per poi procedere con l'analisi del codice di elaborazione del *dataset* storico. Successivamente, verranno documentate le fasi di *benchmarking* dei modelli e la transcodifica in WebAssembly, terminando con la misurazione delle prestazioni del sistema operante in ambiente Edge.

3.1 Configurazione dell'Ambiente di Lavoro

La progettazione di un sistema eterogeneo, che spazia dall'analisi statistica in *Python* fino all'esecuzione di binari compilati all'interno di un *browser* tramite *JavaScript*, richiede un ecosistema di sviluppo altamente strutturato e riproducibile. Lo sviluppo dell'intero progetto è stato condotto su un ambiente operativo basato su distribuzione **Ubuntu** (Linux), una scelta ar-

chitetturale che garantisce la massima compatibilità nativa con le librerie di calcolo scientifico e con i compilatori a basso livello richiesti dalla *toolchain* di compilazione.

3.1.1 L'IDE di Sviluppo: Visual Studio Code

Per la stesura materiale del codice, la scelta è ricaduta su **Visual Studio Code** (VS Code), un *Integrated Development Environment* (IDE) sviluppato da Microsoft. A differenza dei tradizionali IDE monolitici, VS Code si distingue per la sua architettura modulare ed estremamente leggera. Tramite l'installazione mirata di estensioni ufficiali, è stato possibile configurare un *setup* multi-linguaggio perfettamente integrato.

Questo ambiente unificato ha permesso di gestire contemporaneamente, all'interno del medesimo *workspace*, l'esecuzione interattiva dei *Jupyter Notebook* (utilizzati per l'analisi esplorativa dei dati in Python), la navigazione del codice sorgente generato in C, e lo sviluppo dell'interfaccia utente web (*HTML, CSS, JavaScript*). Inoltre, l'integrazione nativa del terminale Linux all'interno dell'editor ha semplificato drasticamente l'orchestrazione degli *script* di compilazione.

3.1.2 Isolamento delle Dipendenze: Virtual Environment (venv)

Nello sviluppo di software legati al *Machine Learning*, uno dei rischi più comuni è il conflitto tra le versioni delle librerie, che può compromettere la riproducibilità degli esperimenti. Per scongiurare questa criticità, l'intero *stack* di analisi dati è stato confinato all'interno di un ambiente virtuale isolato, generato tramite il modulo *Python* nativo `venv`.

L'utilizzo di un *Virtual Environment* garantisce che le librerie installate (come *Pandas*, *Scikit-learn*, *XGBoost* e *Optuna*) non interferiscano con i pacchetti di sistema globali del sistema operativo ospitante. Al termine della configurazione, l'esatto stato dell'ambiente e le specifiche versioni dei pac-

chetti sono stati "congelati" (*freezing*) in un file `requirements.txt`. Questa *best practice* di Ingegneria del Software assicura che il motore algoritmico possa essere replicato e mandato in esecuzione su qualsiasi altra macchina in modo perfettamente deterministico, ricreando l' ecosistema computazionale con un singolo comando.

3.1.3 Gestione del Codice con Git

Considerata la complessità del progetto e la necessità di gestire molteplici transizioni di stato del modello (dalla prototipazione alla compilazione finale), il controllo di versione è stato affidato a **Git**, utilizzando **GitHub** come *repository* remoto. La cartella del progetto si trova al seguente link: `https://github.com/MatteMon03/Tesi_Monari.git`.

Il flusso di lavoro ha adottato logiche rigorose di versionamento:

- **Branching Strategico:** lo sviluppo non è avvenuto in modo lineare sul ramo principale (*main*). Sono stati creati rami di sviluppo isolati (*branch*) per affrontare compiti specifici senza destabilizzare il codice in produzione. Ad esempio, la complessa implementazione del ponte di comunicazione tra *JavaScript* e WebAssembly è stata sviluppata e testata su un ramo separato, e solo a fronte di esito positivo è stata unita (*merge*) alla base di codice stabile.

- **Commit Atomici:** ogni modifica architetturale o aggiornamento degli *script* di *benchmarking* è stata tracciata con *commit* granulari e descrittivi. Questo tracciamento puntuale ha permesso di mantenere uno storico esatto delle performance, garantendo la possibilità di regredire istantaneamente (*rollback*) a versioni precedenti del codice qualora l'ottimizzazione di un iperparametro avesse prodotto anomalie matematiche.

3.2 Ingestione, Pulizia e Ottimizzazione del Dataset

Il primo passo per la costruzione del motore predittivo ha riguardato l'importazione e la strutturazione dei dati grezzi. L'intero processo di *Data Preparation* è stato codificato in *Python* sfruttando l'efficienza vettoriale delle librerie scientifiche standard.

3.2.1 Inizializzazione dell'Ambiente e Importazione dei Dati

Il processo operativo si avvia con l'inizializzazione dell'ambiente di calcolo. Come si evince dal blocco di codice sottostante, lo script fa affidamento su un ecosistema eterogeneo di moduli: *Pandas* e *NumPy* per la manipolazione matriciale e l'algebra lineare, *Scikit-learn* per le *pipeline* di pre-elaborazione (`StandardScaler`) e il calcolo delle metriche di errore, e *Matplotlib* per il *rendering* grafico. Viene inoltre integrata la libreria `tqdm` per il monitoraggio asincrono dei cicli iterativi più onerosi.

Una volta caricati in memoria i moduli, il *dataset* grezzo del traffico (precedentemente acquisito dalla piattaforma Kaggle e archiviato localmente nel direttorio `archive`) viene esaminato all'interno di un *DataFrame* attraverso la funzione di decodifica nativa, pronto per le successive fasi di ispezione.

```
1    import pandas as pd
2    import numpy as np
3    import matplotlib.pyplot as plt
4    from scipy.stats import pearsonr
5    from sklearn.metrics import mean_absolute_error, r2_score
6    from sklearn.pipeline import make_pipeline
7    from sklearn.preprocessing import StandardScaler
8    from tqdm.notebook import tqdm
9
10   df = pd.read_csv('archive/traffic.csv')
11
```

```
12    display(df.head(10))
```

Listing 3.1: Script di inizializzazione librerie e importazione del dataset CSV

L'output di questo primo script si può osservare in figura 2.1

3.2.2 Scomposizione della Variabile DateTime

Gli algoritmi basati su Machine Learning si fondano su operatori matematici e relazionali (es. divisioni basate su soglie numeriche) e non possiedono la capacità nativa di interpretare nativamente le stringhe alfanumeriche o gli oggetti di tipo *timestamp*.

Pertanto, la fase successiva si è concentrata sul *Feature Engineering*. La colonna originale `DateTime` è stata convertita nel formato datario ottimizzato di *Pandas* e successivamente divisa nelle sue componenti intere. Come si evince dal codice seguente, è stata data particolare importanza all'estrazione del *giorno della settimana* (`DayOfWeek`), un parametro ciclico fondamentale per far apprendere all'algoritmo la drastica differenza nei volumi di traffico tra i giorni feriali (lavorativi/scolastici) e i fine settimana.

```
1     df['DateTime'] = pd.to_datetime(df['DateTime'])
2     df = df.sort_values('DateTime').reset_index(drop=True)
3     df['Year'] = df['DateTime'].dt.year
4     df['Month'] = df['DateTime'].dt.month
5     df['Day'] = df['DateTime'].dt.day
6     df['Hour'] = df['DateTime'].dt.hour
7     df['DayOfWeek'] = df['DateTime'].dt.dayofweek
8
9     df_final = df.drop(['DateTime', 'ID'], axis=1)
10
11    display(df_final.head(10))
```

Listing 3.2: Script di caricamento e scomposizione della variabile temporale

L'output di questo script si può osservare invece in figura 2.2.

3.2.3 EDA e Valutazione Statistica

Una volta derivate le nuove variabili indipendenti, si è passato all'Analisi Esplorativa dei Dati. Questo passaggio permette di comprendere la topologia dei dati, individuare anomalie distributive e validare l'incidenza statistica delle funzionalità sulla variabile *target* (`Vehicles`).

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme(style="whitegrid")
df_final.info()

display(df_final.describe())

display(df_final.nunique())
```

Listing 3.3: Script per l'estrazione delle statistiche descrittive generali

```
INFORMAZIONI GENERALI:
<class 'pandas.DataFrame'>
RangeIndex: 48120 entries, 0 to 48119
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Junction   48120 non-null  int64
 1   Vehicles   48120 non-null  int64
 2   Year       48120 non-null  int32
 3   Month      48120 non-null  int32
 4   Day        48120 non-null  int32
 5   Hour       48120 non-null  int32
 6   DayOfWeek  48120 non-null  int32
dtypes: int32(5), int64(2)
memory usage: 1.7 MB
```

Figura 3.1: Output a terminale mostrante le informazioni generali del dataset.

Come illustrato in Figura 3.1, l'invocazione del metodo `info()` di Pandas fornisce una panoramica strutturale completa del *dataset* al termine della fase di analisi e modellazione delle *feature*. L'analisi dell'output di validazione restituisce tre evidenze fondamentali per garantire la robustezza della successiva fase di addestramento:

- **Integrità e Completezza (Assenza di valori nulli):** Il *DataFrame* si compone di 48.120 registrazioni. L'indicatore *Non-Null Count* conferma che tutte le colonne presentano esattamente 48.120 valori validi.
- **Tipizzazione Numerica Ottimale:** L'intero set di variabili, sia predittive che *target*, risulta correttamente codificato in formati numerici

interi (`int32` e `int64`).

STATISTICHE DESCRITTIVE:

	Junction	Vehicles	Year	Month	Day	Hour	DayOfWeek
count	48120.000000	48120.000000	48120.000000	48120.000000	48120.000000	48120.000000	48120.000000
mean	2.180549	22.791334	2016.269825	5.884289	15.700748	11.500000	2.996010
std	0.966955	20.750063	0.616093	3.569872	8.784073	6.922258	2.000017
min	1.000000	1.000000	2015.000000	1.000000	1.000000	0.000000	0.000000
25%	1.000000	9.000000	2016.000000	3.000000	8.000000	5.750000	1.000000
50%	2.000000	15.000000	2016.000000	5.000000	16.000000	11.500000	3.000000
75%	3.000000	29.000000	2017.000000	9.000000	23.000000	17.250000	5.000000
max	4.000000	180.000000	2017.000000	12.000000	31.000000	23.000000	6.000000

Figura 3.2: Output a terminale della funzione `describe()`, mostrante la sintesi statistica e la distribuzione della variabile *target* e delle *feature*.

Dall'osservazione dei risultati in figura 3.2 emergono due considerazioni strutturali:

- **Variabili Spazio-Temporali:** Le metriche confermano i limiti operativi del dominio di riferimento. Il *dataset* monitora esattamente 4 intersezioni stradali distinte (valori di `Junction` compresi tra 1 e 4), estrapolando dati lungo un orizzonte temporale pluriennale che spazia dall'anno 2015 all'anno 2017. Le variabili temporali cicliche derivate (mesi, giorni, ore) rispettano coerentemente i domini matematici attesi.

- **Asimmetria e Picchi di Congestione (`Vehicles`):** L'analisi statistica della variabile *target* rivela la natura fortemente non-lineare del traffico urbano. Il volume medio si attesta a 22.79 veicoli per intervallo, con una mediana di soli 15 veicoli. Tuttavia, si osserva una deviazione standard estremamente elevata (~ 20.75) e un divario massiccio tra il terzo quartile (29 veicoli) e il valore massimo registrato (180 veicoli). Questa asimmetria positiva (*right-skewed distribution*) indica che, sebbene il flusso di base risulti generalmente contenuto, la rete viaria è soggetta a picchi di congestione estremi (anomalie e ore di punta).

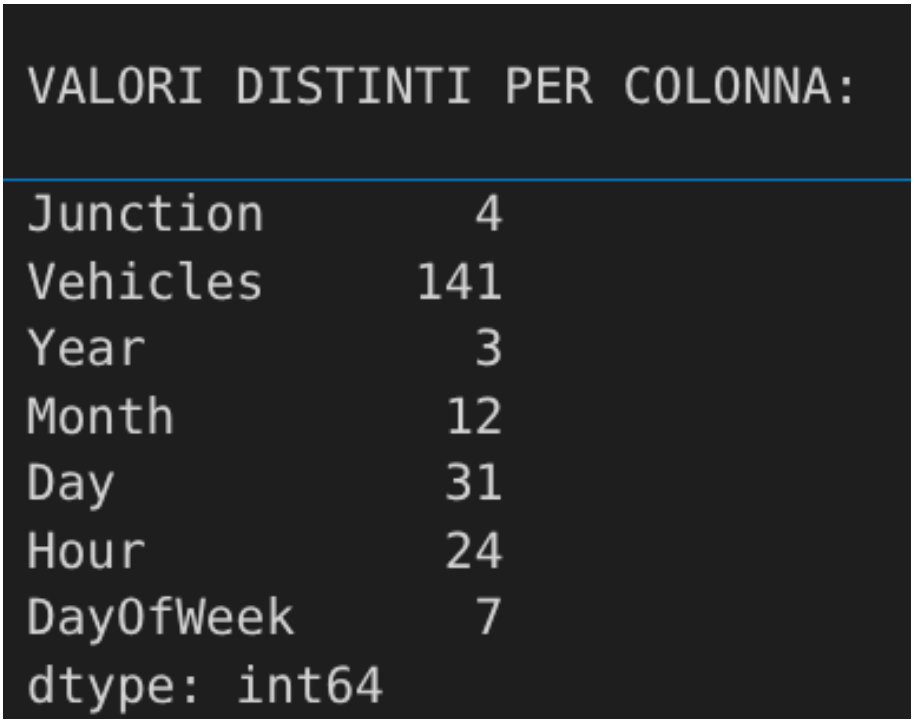

Figura 3.3: Output a terminale della funzione `nunique()`, indicante i valori distinti (cardinalità) per ciascuna colonna del *dataset*.

A completamento dell'analisi esplorativa preliminare, l'estrazione dei valori unici (cardinalità) tramite il metodo `nunique()` svolge la fondamentale funzione di validazione logica dell'analisi delle *feature*. Come evidenziato in Figura 3.3, i risultati confermano l'assoluta coerenza del *dataset* elaborato.

Distribuzione della Variabile Target

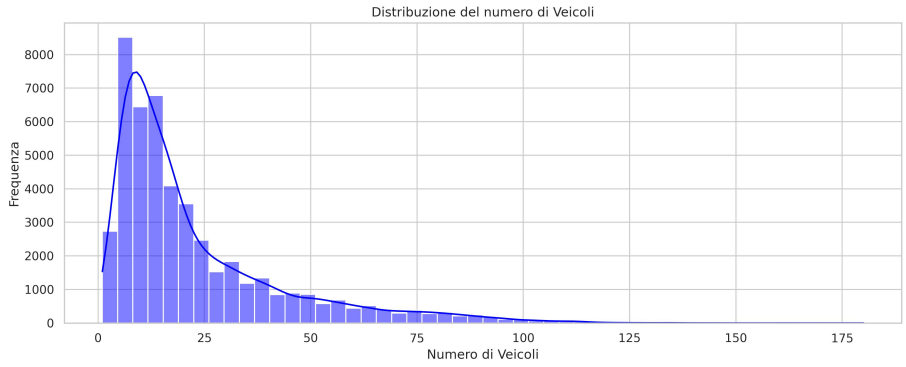

Figura 3.4: Istogramma delle frequenze relative al volume dei veicoli. L'andamento della curva evidenzia la distribuzione non uniforme del traffico urbano.

L'analisi visiva dell'istogramma combinato con la curva di densità del kernel (KDE) in Figura 3.4 conferma sperimentalmente le asimmetrie strutturali evidenziate in precedenza per via tabulare. La distribuzione della variabile *target* `Vehicles` non segue un andamento campionario di tipo Gaussiano, bensì si configura come una distribuzione asimmetrica positiva con una lunghissima coda verso destra.

Questa specifica topologia riflette la natura intrinseca della mobilità urbana all'interno dei nodi stradali analizzati:

- **Il flusso ordinario:** Il picco massimo di frequenza si concentra marcatamente nell'intervallo compreso tra 5 e 20 veicoli orari. Ciò dimostra che, per la maggior parte delle ore del giorno e della notte, l'intersezione verifica un regime di scorrimento veloce e a bassa densità;

- **I fenomeni di congestione:** Oltre la soglia dei 30 veicoli, la curva decade progressivamente ma prosegue il proprio asse orizzontale fino a raggiungere l'estremo di 180 unità. Questa sezione della curva rappresenta gli eventi a bassa frequenza ma ad alto impatto critico, ovvero

i picchi di congestione legati alle ore di punta del pendolarismo mattutino e serale o ad anomalie locali (es. incidenti stradali o blocchi viari).

Matrice di Correlazione e Mappa di Calore

Infine, per misurare in modo formale la dipendenza lineare tra le *feature* temporali, spaziali (l'incrocio di riferimento) e la variabile *target*, abbiamo calcolato la Matrice di Correlazione di Pearson. Per rendere il risultato numerico intellegibile a livello macroscopico, i coefficienti sono stati renderizzati sotto forma di *Heatmap*.

Figura 3.5: Mappa di calore (*Heatmap*) delle correlazioni lineari. Colori caldi (rosso) o freddi (blu) intensi indicano rispettivamente una forte correlazione proporzionale o inversamente proporzionale.

L'ispezione visiva della *Heatmap* in Figura 3.5 ha evidenziato come variabili quali l'ora del giorno (`Hour`) e l'identificativo dell'incrocio (`Junction`) esibiscono una correlazione significativa con la densità veicolare.

3.2.4 Strategia di Partizionamento per Serie Storiche

L'ultimo passaggio della *pipeline* di preparazione dati è stato lo *splitting* del *dataset* in segmenti di *Training*, *Validation* e *Test*. Trattandosi di misurazioni del traffico, i dati configurano una vera e propria **Serie Storica**.

A differenza dei classici problemi di regressione, in cui il partizionamento avviene in modo puramente casuale (*Random Split*), mescolare i dati stradali porterebbe al *Data Leakage* (fuga di informazioni). Se al modello venissero forniti dati casuali del futuro per prevedere il passato, le metriche di accuratezza risulterebbero artificialmente distorte. Pertanto, il partizionamento è stato condotto applicando un taglio temporale cronologico.

```
1
2     X = df_final.drop(['Vehicles'], axis=1)
3     y = df_final['Vehicles']
4
5     n = len(df_final)
6     train_index = int(n * 0.7)
7     val_index = int(n * 0.85)
8
9     X_train = X.iloc[:train_index]
10    y_train = y.iloc[:train_index]
11    X_val = X.iloc[train_index:val_index]
12    y_val = y.iloc[train_index:val_index]
13    X_test = X.iloc[val_index:]
14    y_test = y.iloc[val_index:]
```

Listing 3.4: Suddivisione cronologica in set di Training, Validation e Test

3.3 Implementazione e Benchmarking dei Modelli di Baseline

Completata la fase di sanificazione e strutturazione spaziotemporale del *dataset*, l'impianto sperimentale si è spostato sulla fase di addestramento (*Training*) e valutazione (*Validation*). E' stata implementata una campagna di *benchmarking* che ha messo a confronto modelli parametrici, *instance-based* e architetture *Ensemble* basate su alberi decisionali.

3.3.1 La Funzione di Valutazione (model_evaluation)

Per garantire un confronto scientificamente imparziale e rispettare il principio DRY (*Don't Repeat Yourself*), è stata progettata una funzione *Python* centralizzata denominata `model_evaluation`.

Questa architettura software accetta in *input* il nome identificativo dell'algoritmo, i vettori contenenti i valori reali di collaudo (`y_true`) e le previsioni generate dal modello (`y_pred`). Internamente, la funzione quantifica le prestazioni calcolando un set di metriche statistiche rigorose: l'Errore Assoluto Medio (MAE), il coefficiente di determinazione (R^2), l'indice di correlazione di Pearson e l'errore relativo percentuale rispetto alla media. I risultati vengono istantaneamente stampati a terminale e salvati dinamicamente in una struttura dati a dizionario globale (`dizionario`), permettendo una rapida aggregazione tabulare e la successiva generazione dei grafici comparativi.

```
1     dizionario = {}
2
3     def model_evaluation(model_name, y_true, y_pred):
4         mae = mean_absolute_error(y_true, y_pred)
5         re = r2_score(y_true, y_pred)
6         coeff, p_value = pearsonr(y_true, y_pred)
7         mean_y = np.mean(y_true)
8         errore_relativo = (mae / mean_y) * 100
9
10        dizionario[model_name] = {
```

```
            'MAE': mae,
            'R2': re,
            'Pearson': coeff,
            'Errore Relativo': errore_relativo
        }

        print(f"--- Risultati {model_name} ---")
        print(f"MAE: {mae:.2f}")
        print(f"R2: {re:.2f}")
        print(f"Pearson Coeff: {coeff:.2f}")
        print(f"Errore Relativo: {errore_relativo:.2f} %")

```

Listing 3.5: Definizione della funzione centralizzata per il calcolo delle metriche di errore

3.3.2 Istanziazione e Addestramento degli Algoritmi

Sfruttando l'interfaccia unificata esposta dalla libreria *Scikit-learn* e dai *framework* nativi di *XGBoost* e *LightGBM*, i modelli analizzati sono stati istanziati in sequenza. Per garantire la totale riproducibilità scientifica dell'esperimento, il parametro di inizializzazione stocastica (`random_state`) è stato fissato a un valore costante (42) per tutti gli algoritmi.

Inoltre, per i modelli basati sul calcolo delle distanze metriche (come K-NN e SVR) e per le reti neurali (MLP), è stata implementata una *pipeline* con `StandardScaler`. Questa operazione di standardizzazione assicura che variabili con scale di grandezza differenti non sbilancino il calcolo dei gradienti o delle distanze euclidee.

Di seguito viene riportato il codice di implementazione, suddiviso per famiglie architetturali.

Modelli Lineari e Non Parametrici

```
    from sklearn.linear_model import LinearRegression
    from sklearn.neighbors import KNeighborsRegressor
```

```
    from sklearn.pipeline import make_pipeline
    from sklearn.preprocessing import StandardScaler

    # --- REGRESSIONE LINEARE ---
    lin_reg = LinearRegression()
    lin_reg.fit(X_train, y_train)
    y_pred_lin = lin_reg.predict(X_val)
    model_evaluation("Regressione Lineare", y_val,
y_pred_lin)

    # --- K-NEAREST NEIGHBORS (K-NN) ---
    knn_model = make_pipeline(
        StandardScaler(),
        KNeighborsRegressor(n_neighbors=20, n_jobs=-1)
    )
    knn_model.fit(X_train, y_train)
    y_pred_knn = knn_model.predict(X_val)
    model_evaluation("K-Nearest Neighbors", y_val,
y_pred_knn)
```

Listing 3.6: Istanziazione e addestramento della Regressione Lineare e del K-Nearest Neighbors

Modelli Kernel-Based e Connessionisti

```
    from sklearn.svm import SVR
    from sklearn.neural_network import MLPRegressor

    # --- MULTI-LAYER PERCEPTRON (Rete Neurale) ---
    mlp_model = make_pipeline(
        StandardScaler(),
        MLPRegressor(hidden_layer_sizes=(100, 50),
            activation='relu',
            solver='adam',
            max_iter=500,
            random_state=42
            )
```

```
    )
    mlp_model.fit(X_train, y_train)
    y_pred_mlp = mlp_model.predict(X_val)
    model_evaluation("Multi-Layer Perceptron", y_val,
y_pred_mlp)

    # --- SUPPORT VECTOR REGRESSION (SVR) ---
    svr_model = make_pipeline(
        StandardScaler(),
        SVR(kernel='rbf',
            C=100,
            epsilon=0.1,
            cache_size=1000
        )
    )

    svr_model.fit(X_train, y_train)
    y_pred_svr = svr_model.predict(X_val)
    model_evaluation("Support Vector Regression", y_val,
y_pred_svr)
```

Listing 3.7: Istanziazione e addestramento di Support Vector Machine e Multi-Layer Perceptron

Modelli Ensemble basati su Alberi (Tree-based)

```
    from sklearn.ensemble import RandomForestRegressor
    from xgboost import XGBRegressor
    from lightgbm import LGBMRegressor

    # --- RANDOM FOREST ---
    rf_model = RandomForestRegressor(
        n_estimators=1000,
        max_depth=15,
        min_samples_leaf=5,
        random_state=42,
        n_jobs=-1
```

```
    )
    rf_model.fit(X_train, y_train)
    y_pred_rf = rf_model.predict(X_val)
    model_evaluation("Random Forest", y_val, y_pred_rf)

    # --- EXTREME GRADIENT BOOSTING (XGBoost) ---
    xgb_model = XGBRegressor(
        n_estimators=1000,
        learning_rate=0.01,
        max_depth=8,
        subsample=0.8,
        random_state=42,
        n_jobs=-1
    )
    xgb_model.fit(X_train, y_train)
    y_pred_xgb = xgb_model.predict(X_val)
    model_evaluation("XGBoost", y_val, y_pred_xgb)

    # --- LIGHT GRADIENT BOOSTING MACHINE (LightGBM) ---
    lgb_model = LGBMRegressor(
        n_estimators=1000,
        learning_rate=0.01,
        max_depth=8,
        num_leaves=50,
        subsample=0.8,
        random_state=42,
        n_jobs=-1,
        verbose=-1
    )
    lgb_model.fit(X_train, y_train)
    y_pred_lgb = lgb_model.predict(X_val)
    model_evaluation("LightGBM", y_val, y_pred_lgb)
```

Listing 3.8: Istanziazione e addestramento delle foreste decisionali: Random Forest, XGBoost e LightGBM

3.3.3 Analisi Comparativa e Selezione del Modello

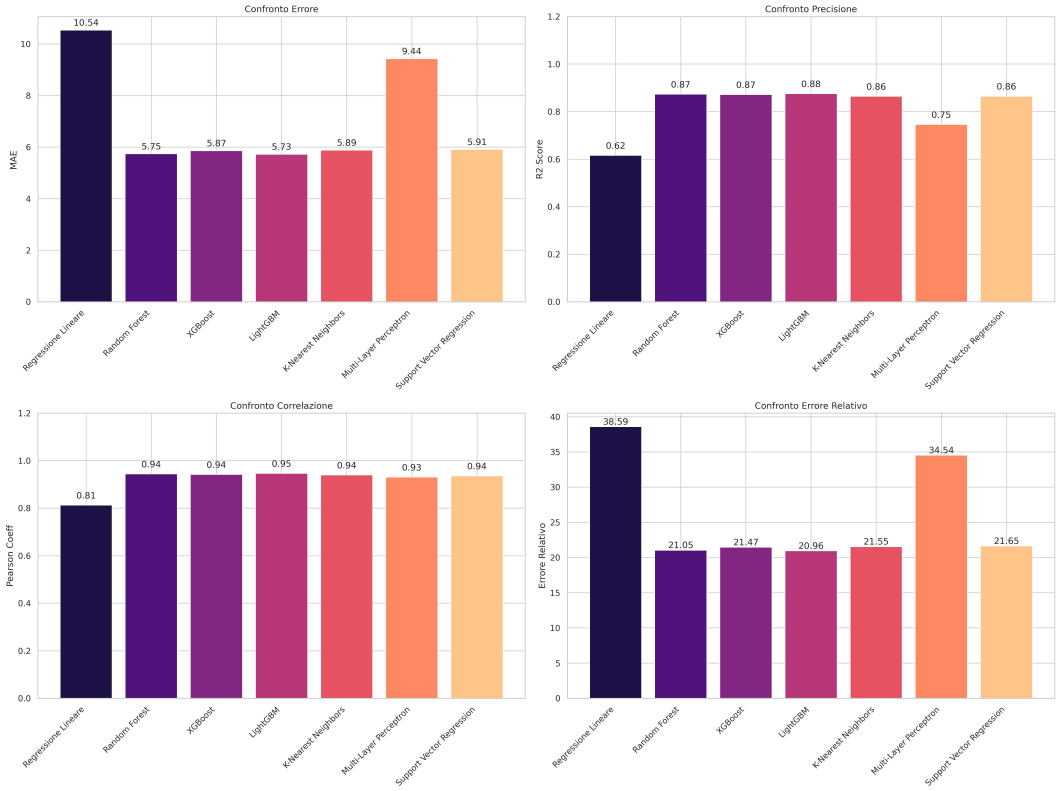

Figura 3.6: Confronto grafico delle metriche di errore e correlazione tra i diversi modelli testati nella fase di *baseline*.

I risultati estratti dalla funzione di valutazione sono stati elaborati graficamente per permettere un'analisi delle *performance*. Come si evince dai grafici in Figura 3.6, le differenze architetturali tra i vari paradigmi matematici si sono tradotte in marcate divergenze predittive. I modelli lineari (Regressione Lineare) e le reti neurali superficiali (Multi-Layer Perceptron) hanno esibito evidenti limiti strutturali, registrando gli errori più elevati (MAE rispettivamente di 10.54 e 9.44). Tali architetture si sono dimostrate incapaci di catturare i picchi di congestione repentini che caratterizzano le serie storiche del traffico (fenomeno di *underfitting*). Al contrario, le architetture *Ensemble* basate sugli alberi di decisione (Random Forest, XGBoost e LightGBM)

hanno dominato nettamente il *benchmarking*, attestandosi tutte su un coefficiente di determinazione (R^2) prossimo a $0.87 - 0.88$ e un Errore Relativo intorno al 21%.

L'analisi complessiva ha eletto **XGBoost** come algoritmo primario per il prosieguo del progetto. L'architettura di XGBoost rappresenta, in questo specifico scenario di distribuzione, il miglior compromesso tra elevata accuratezza predittiva e affidabilità durante le fasi di transcodifica.

3.4 Ottimizzazione degli Iperparametri: Optuna e XGBoost

Selezionato XGBoost come motore predittivo primario, l'architettura è stata sottoposta a una fase di taratura degli iperparametri (*Hyperparameter Tuning*). L'obiettivo di questa procedura è esplorare lo spazio delle configurazioni del modello per individuare la combinazione esatta in grado di minimizzare l'errore di generalizzazione, limitando al contempo il rischio di *overfitting*.

L'impiego di metodi di ricerca a forza bruta (*Grid e Randomized Search*) risulta computazionalmente insostenibile per modelli strutturalmente complessi. Si è pertanto proceduto all'integrazione del *framework* **Optuna**, delegando la ricerca all'algoritmo di Ottimizzazione Bayesiana basato su TPE.

3.4.1 Integrazione dello Script di Ottimizzazione Bayesiana

L'implementazione in *Python* ha richiesto la definizione di una funzione obiettivo (`objective`), la quale accetta come argomento un oggetto `trial` generato iterativamente da Optuna. All'interno di questa funzione è stato mappato lo spazio di ricerca per i sette iperparametri più critici dell'architettura XGBoost: numero di estimatori, profondità massima degli alberi, tasso di

apprendimento, frazioni di campionamento delle *feature* e delle osservazioni, peso minimo dei nodi foglia e parametro di regolarizzazione *gamma*.

```
import optuna
from xgboost import XGBRegressor
from sklearn.metrics import mean_absolute_error

def objective(trial):
hyperparam = {
    'n_estimators': trial.suggest_int('n_estimators',
500, 2000),
    'max_depth': trial.suggest_int('max_depth', 3, 10),
    'learning_rate':
trial.suggest_float('learning_rate', 0.005, 0.1,
log=True),
    'subsample': trial.suggest_float('subsample', 0.5,
1.0),
    'colsample_bytree':
trial.suggest_float('colsample_bytree', 0.5, 1.0),
    'min_child_weight':
trial.suggest_int('min_child_weight', 1, 10),
    'gamma': trial.suggest_float('gamma', 1e-8, 1.0,
log=True),
    'random_state': 42,
    'n_jobs': -1
}

model = XGBRegressor(**hyperparam)

model.fit(X_train, y_train)

preds = model.predict(X_val)
mae = mean_absolute_error(y_val, preds)

return mae

study = optuna.create_study(direction='minimize')

```

```
29     study.optimize(objective, n_trials=50,
       show_progress_bar=True)
30
31     print("Migliori parametri individuati:",
       study.best_params)
```

Listing 3.9: Script di definizione dello spazio di ricerca ed esecuzione dell'Ottimizzazione Bayesiana

Il processo di *tuning* è stato impostato con la direttiva `direction='minimize'`, al fine di ricercare la minimizzazione pura dell'Errore Assoluto Medio (MAE) calcolato sul *Validation Set*. Sfruttando le scale logaritmiche (`log=True`) per l'esplorazione di parametri sensibili come il *learning rate* e il *gamma*, l'algoritmo bayesiano è riuscito a campionare lo spazio in modo estremamente efficiente.

Grazie alla memoria degli addestramenti precedenti, sono stati sufficienti 50 *trial* computazionali per far convergere l'algoritmo verso una configurazione stabilmente ottimizzata, evitando lo spreco di cicli di CPU in aree dello spazio matematicamente sub-ottimali.

Come si evince dall'andamento della curva di ottimizzazione in Figura 3.7, l'algoritmo ha esplorato lo spazio riducendo progressivamente l'errore, individuando e stabilizzandosi sul minimo globale. L'analisi delle 50 iterazioni ha eletto come architettura ottimale la seguente configurazione di iperparametri: `n_estimators: 1496`, `max_depth: 3`, `learning_rate: 0.0248`, `subsample: 0.716`, `colsample_bytree: 0.898`, `min_child_weight: 1` e `gamma: 9.11e-07`.

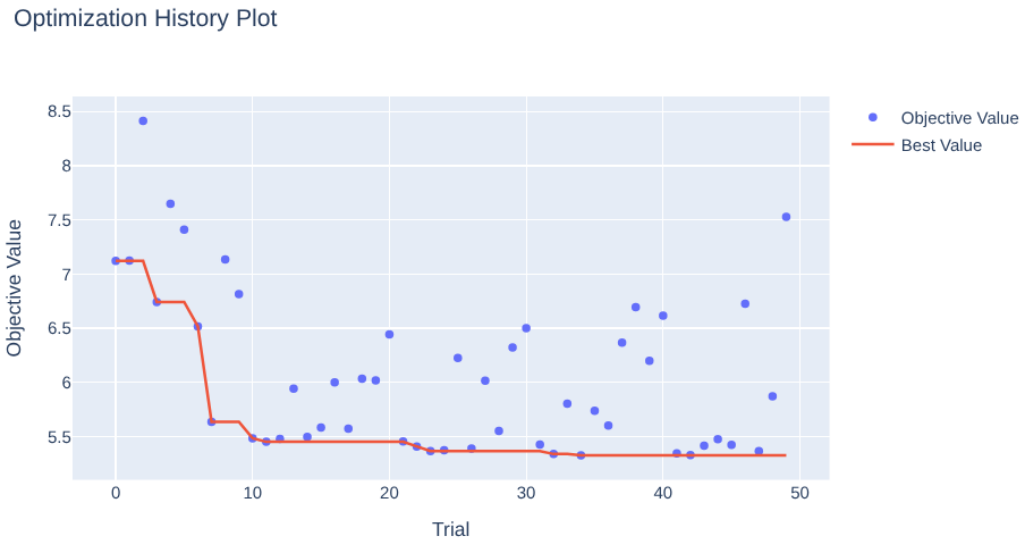

Figura 3.7: Andamento della funzione obiettivo (MAE) durante i 50 *trial* di ottimizzazione bayesiana.

Oltre al monitoraggio della convergenza globale, l'infrastruttura analitica di Optuna consente di valutare l'influenza relativa di ciascuna variabile esplorata attraverso l'algoritmo fANOVA (*Functional Analysis of Variance*). Questa analisi scompone la varianza della funzione obiettivo per quantificare quanto ogni singolo iperparametro abbia inciso sulla riduzione dell'errore.

Come illustrato nell'istogramma in Figura 3.8, l'analisi ha rivelato una netta gerarchia di importanza. La profondità massima dell'albero (`max_depth`) si configura come il parametro assolutamente dominante, determinando da solo il 62% dell'efficacia dell'ottimizzazione. Seguono, con un peso significativamente inferiore, la frazione di colonne campionate per albero (`colsample_bytree` al 18%) e il tasso di apprendimento (`learning_rate` al 15%).

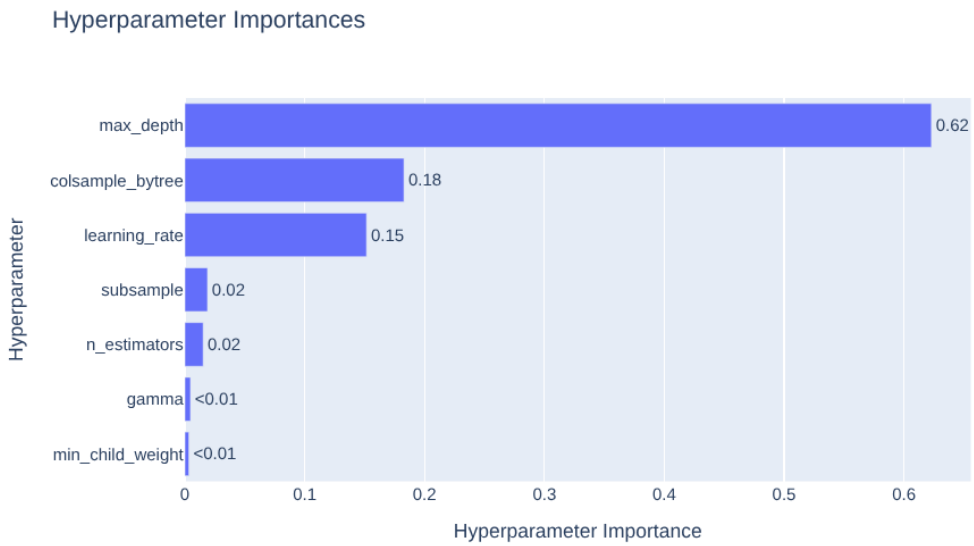

Figura 3.8: Valutazione dell'importanza degli iperparametri (calcolata da Optuna) sull'accuratezza del modello XGBoost. La profondità dell'albero (`max_depth`) rappresenta il fattore critico per la minimizzazione dell'errore.

Questo risultato risulta perfettamente coerente con la teoria matematica alla base degli alberi decisionali applicati alle serie storiche: la profondità regola in modo diretto la complessità topologica del modello e la sua capacità di intercettare le interazioni non lineari tipiche delle dinamiche di congestione stradale. Al contrario, parametri come il numero di estimatori (`n_estimators` al 2%) o la regolarizzazione (`gamma` 1%) hanno mostrato un'incidenza marginale, confermando che una foresta sufficientemente popolata si stabilizza autonomamente, lasciando alla geometria dei singoli alberi il compito di affinare la precisione predittiva.

3.4.2 Addestramento Definitivo e Serializzazione del Modello

Individuata la configurazione matematica ottimale tramite l'algoritmo bayesiano, l'ultimo passaggio della *pipeline* di modellazione ha previsto l'ad-

destramento dell'algoritmo definitivo e il suo salvataggio su disco.

Poiché il *Validation Set* aveva ormai esaurito la sua funzione di calibrazione durante il *tuning* di Optuna, esso è stato concatenato al *Training Set* originale. In questo modo, il modello XGBoost definitivo è stato addestrato su un volume di dati storici significativamente maggiore, massimizzando la sua capacità di catturare i *pattern* del traffico prima di affrontare il collaudo finale sull'intoccato *Test Set*.

Infine, per svincolare l'oggetto *Python* addestrato dalla memoria volatile (RAM) e renderlo persistente per le successive fasi di compilazione *Edge*, l'istanza del modello è stata serializzata e salvata su file fisso utilizzando la libreria standard `pickle`.

```
import pickle
import pandas as pd
from xgboost import XGBRegressor

xgb_model_ott = XGBRegressor(**study.best_params,
random_state=42, n_jobs=-1)

X_fit = pd.concat([X_train, X_val])
y_fit = pd.concat([y_train, y_val])

xgb_model_ott.fit(X_fit, y_fit)

y_pred_xgb = xgb_model_ott.predict(X_test)

model_evaluation("XGBoost_Ottimizzato", y_test,
y_pred_xgb)

model_path_xgb = "./xgb_model_ott.pkl"
with open(model_path_xgb, "wb") as f:
pickle.dump(xgb_model_ott, f)

```

Listing 3.10: Addestramento del modello XGBoost ottimizzato e serializzazione dell'oggetto in formato pickle

3.4.3 Interpretabilità del Modello: Analisi della Feature Importance

Un vantaggio strutturale intrinseco agli algoritmi basati su alberi decisionali, rispetto alle architetture *black-box* come le reti neurali profonde, risiede nella loro elevata interpretabilità. Terminata la fase di addestramento e ottimizzazione, è stato possibile interrogare il modello XGBoost per estrarre la *Feature Importance*, ovvero una metrica che quantifica il contributo relativo di ciascuna variabile indipendente nel determinare le diramazioni ottimali degli alberi.

Come evidenziato dall'istogramma in Figura 3.9, il modello ha appreso relazioni logiche e coerenti con le reali dinamiche della mobilità urbana, confermando la bontà del processo di *Feature Engineering* condotto nella fase iniziale del progetto.

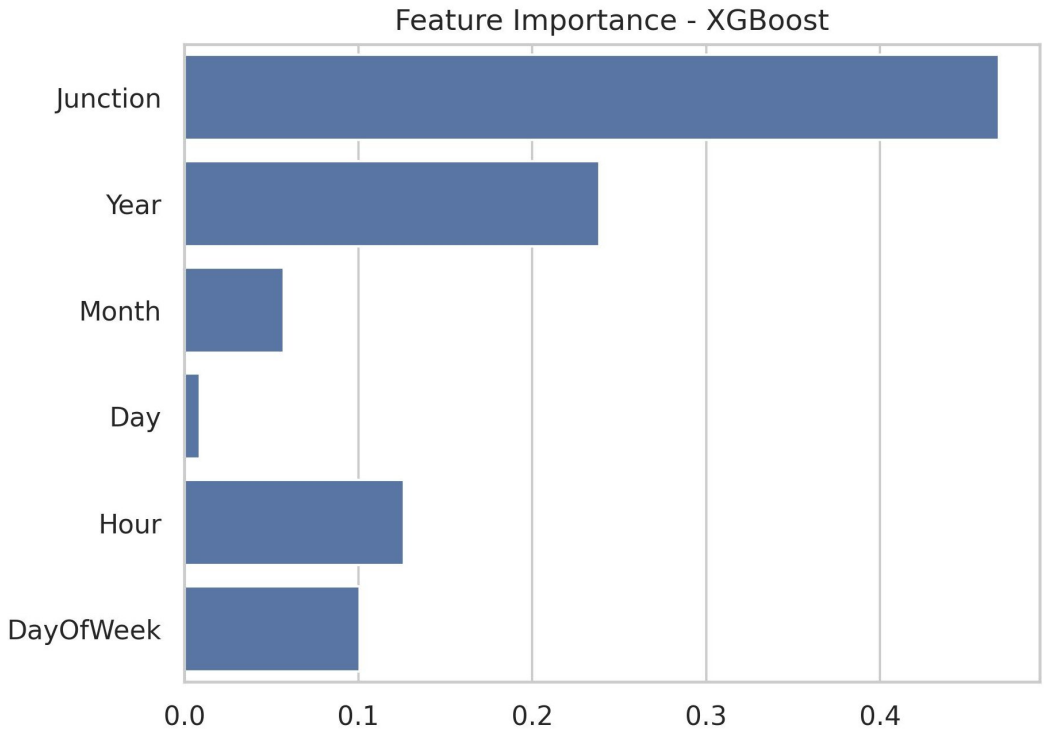

Figura 3.9: Istogramma dell'importanza delle *feature* elaborato dal modello XGBoost ottimizzato.

L'analisi del grafico evidenzia una gerarchia predittiva molto netta:

- **Dominanza della componente spaziale:** L'identificativo dell'incrocio (`Junction`) risulta la variabile in assoluto più discriminante (con un punteggio superiore a 0.45). Ciò dimostra che i volumi base di traffico sono intrinsecamente legati alla topologia della rete stradale e alla capacità strutturale del singolo incrocio.

- **Trend macroscopici:** L'anno di rilevazione (`Year`) rappresenta il secondo fattore di importanza. Questo dato è cruciale, in quanto permette all'algoritmo di intercettare l'aumento progressivo del flusso veicolare e la fisiologica espansione demografica dell'area urbana nel corso del tempo.

- **Ciclicità comportamentale:** Le variabili derivate dalla modellazione del *timestamp*, nello specifico l'ora del giorno (`Hour`) e il giorno della settimana (`DayOfWeek`), catturano perfettamente i ritmi biologici e lavorativi dei pendolari, differenziando i picchi di congestione feriali dalla flessione del traffico nei periodi festivi o notturni.

- **Rumore statistico:** Coerentemente con le aspettative, il giorno esatto del mese (`Day`) e il mese dell'anno (`Month`) apportano un contributo informativo quasi nullo, non influenzando in modo strutturale le abitudini di spostamento quotidiane a parità di giorno della settimana.

3.5 Pipeline di Conversione AOT: ONNX e Treelite

Completata la fase di ottimizzazione matematica e serializzato l'oggetto predittivo in ambiente *Python*, l'obiettivo è trasformare l'algoritmo in un artefatto binario eseguibile localmente all'interno del *browser* dell'utente finale (paradigma *Edge AI*).

Per colmare il divario strutturale tra l'ecosistema di addestramento e il *target* di esecuzione, è stata strutturata una *pipeline* di compilazione *Ahead-of-Time* multi-stadio.

3.5.1 Standardizzazione Intermedia: Esportazione in Formato ONNX

Il primo step di questa catena di conversione ha previsto la traduzione del modello *XGBoost* ottimizzato verso la rappresentazione intermedia standardizzata **ONNX**. Questo passaggio disaccoppia definitivamente la logica matematica dell'algoritmo dal codice *Python* che lo ha generato, trasformando la foresta decisionale in un Grafo Computazionale Diretto Aciclico (DAG) serializzato tramite *Protocol Buffers*.

Da un punto di vista strettamente implementativo, la conversione richiede specifici accorgimenti per garantire l'interoperabilità dei tipi di dato. Come si evince dallo script sottostante, prima di invocare il convertitore, è stato necessario accedere a basso livello all'oggetto `booster` di XGBoost per annullare i metadati testuali relativi ai nomi delle *feature*. Questa purificazione del modello è obbligatoria in quanto ONNX elabora le *feature* in ingresso sotto forma di tensori numerici anonimi, e la presenza di stringhe genererebbe un errore di validazione del grafo.

```
    import onnxmltools
    from skl2onnx.common.data_types import FloatTensorType

    xgb_model_ott.get_booster().feature_names = None

    num_features = X_train.shape[1]

    initial_type = [('float_input', FloatTensorType([None,
   num_features]))]

    onnx_model = onnxmltools.convert_xgboost(
        xgb_model_ott,
        initial_types=initial_type,
        target_opset=13
    )

    onnx_path = "./WebSite/benchmark_web/xgb_model_ott.onnx"
    with open(onnx_path, "wb") as f:
    f.write(onnx_model.SerializeToString())

```

Listing 3.11: Purificazione dei metadati ed esportazione del modello ottimizzato nello standard ONNX

Un ulteriore parametro architetturale critico definito durante la conversione è l'istruzione `target_opset=13`. Essa impone al transcodificatore di limitare l'utilizzo degli operatori matematici interni a una specifica versione dello standard ONNX. Questa restrizione a ritroso (*backward compatibility*)

garantisce che il file generato risulti perfettamente compatibile con il motore di esecuzione *ONNX Runtime Web*, che andrà, come vedremo in seguito, a processare il file in ambiente *JavaScript* e *WebGPU*.

3.5.2 Generazione del Codice C e Compilazione AOT tramite Treelite

La seconda direttrice della *pipeline* di transcodifica ha sfruttato l'ecosistema nativo sviluppato dalla DMLC (gli stessi creatori di XGBoost), impiegando le librerie **Treelite** e **tl2cgen**. A differenza di ONNX, che mappa l'algoritmo in un grafo computazionale serializzato, questo paradigma si basa sulla generazione esplicita di codice sorgente a basso livello e sulla sua successiva compilazione AOT.

Il processo implementato si articola in due macro-fasi sequenziali. Inizialmente, l'oggetto interno (`booster`) viene estratto dal modello XGBoost e tradotto nella rappresentazione astratta di *Treelite*. Da qui, il generatore `tl2cgen` traduce ogni singolo nodo della foresta in istruzioni C native (strutture di controllo `if-else` annidate), depositando i file sorgente generati in una cartella dedicata (`./model_c_src`).

```
    import treelite
    import tl2cgen
    import os
    import contextlib

    booster = xgb_model_ott.get_booster()
    tl_model = treelite.frontend.from_xgboost(booster)

    tl2cgen.generate_c_code(
        tl_model,
        dirpath="./model_c_src",
        params={'parallel_comp': 4},
        verbose=False
    )

```

```
16      with open(os.devnull, 'wb') as devnull:
17      with contextlib.redirect_stdout(devnull),
        contextlib.redirect_stderr(devnull):
18      tl2cgen.export_lib(
19          tl_model,
20          toolchain="gcc",
21          libpath="./xgb_model_ott.so",
22          params={'parallel_comp': 4},
23          verbose=False
24      )
25
```

Listing 3.12: Generazione del codice C e compilazione in libreria dinamica condivisa

Come evidenziato dallo script, la traduzione di una foresta composta da quasi 1500 alberi richiede un carico computazionale notevole. Per prevenire colli di bottiglia durante l'esportazione, il processo è stato esplicitamente parallelizzato su 4 *thread* logici della CPU (`parallel_comp: 4`).

La seconda fase dello script invoca la *toolchain* standard GNU (`gcc`) per compilare i file sorgente C appena generati in una libreria condivisa dinamica (`.so`). I file C puri salvati nella directory `model_c_src` rappresentano l'artefatto definitivo che verrà successivamente processato da **Emscripten** per la generazione del binario WebAssembly (`.wasm`), chiudendo così l'anello architetturale dell'Edge Computing.

3.5.3 Infrastruttura di Profilazione Hardware (tecno_evaluation)

Prima di procedere con la trasposizione dei modelli compilati all'interno dell'applicativo WebAssembly periferico, si è resa necessaria una validazione formale in ambiente nativo. L'obiettivo è verificare che il processo di transcodifica (verso ONNX e Treelite) non abbia introdotto penalità prestazionali o anomalie nell'allocazione della memoria rispetto all'esecuzione dell'oggetto XGBoost originario in *Python*.

A tale scopo, è stata programmata una suite di collaudo unificata, incapsulata nella funzione `tecno_evaluation`. Questa *pipeline* di testing hardware è progettata per estrarre metriche operative di basso livello in modo rigoroso e riproducibile, isolando i colli di bottiglia del sistema.

Come si evince dal riassunto del codice implementativo, la funzione garantisce la validità statistica dei risultati eseguendo un ciclo di *stress-test* da 1000 iterazioni consecutive (`n_iters = 1000`). Per evitare che l'accumulo di oggetti falsi le misurazioni della memoria, il *Garbage Collector* di Python (`gc.collect()`) viene forzatamente invocato prima di ogni caricamento. Tutte le metriche estratte vengono automaticamente aggregate e registrate sequenzialmente in un *dataset* CSV, predisponendo i dati per l'analisi grafica comparativa.

```
    import time, cProfile
    from memory_profiler import memory_usage

    def tecno_evaluation(model_path, tecno_type):

    if tecno_type == 'xgboost':
    x_input = X_test.iloc[0:1]
    predict_fn = lambda: model_obj.predict(x_input)

    elif tecno_type == 'onnx':
    x_input = X_test.iloc[0:1].values.astype(np.float32)
    input_name = model_obj.get_inputs()[0].name
    predict_fn = lambda: model_obj.run(None, {input_name:
x_input})

    elif tecno_type == 'treelite':
    x_input =
tl2cgen.DMatrix(X_test.iloc[0:1].values.astype(np.float32))
    predict_fn = lambda: model_obj.predict(x_input)

    profiler = cProfile.Profile()
    profiler.enable()
    for n in range(100): predict_fn()
```

```
22      profiler.disable()
23
24      inf_times = []
25      for n in range(n_iters):
26      t0 = time.perf_counter()
27      predict_fn()
28      inf_times.append((time.perf_counter() - t0) * 1000)
29
30      def stress_test():
31      for _ in range(n_iters): predict_fn()
32      mem_use = memory_usage(stress_test, interval=0.01)
33
```

Listing 3.13: Estratto dello script di profilazione hardware in ambiente Python

L'architettura di questo script monitora quattro domini fondamentali:

1. **Latenza di Inferenza e *Throughput*:** Viene misurato il tempo esatto (in millisecondi) richiesto per generare una singola previsione, calcolando di conseguenza il volume di richieste gestibili al secondo (*Throughput*).

2. **Latenza di Caricamento (*Cold Start*):** Viene quantificato l'overhead temporale richiesto per leggere il file binario dal disco e istanziare l'oggetto in memoria RAM.

3. **Impronta di Memoria):** Attraverso le librerie `tracemalloc` e `psutil`, viene tracciato il picco massimo di allocazione della RAM durante la deserializzazione del modello, un parametro vitale per le architetture Edge a risorse limitate.

4. **Tracciamento dei *Thread* (*CPU Profiling*):** Utilizzando la libreria nativa C `cProfile`, l'algoritmo ordina e conteggia le chiamate alle funzioni a livello di *kernel*, permettendo di individuare quale modulo interno consumi il maggior numero di cicli di processore.

3.5.4 Analisi dei Risultati: L'Impatto Architetturale della Transcodifica

Al fine di raccogliere la telemetria prestazionale, la suite `tecno_evaluation` è stata invocata sequenzialmente sui tre artefatti generati nei paragrafi precedenti: l'oggetto *Python* nativo serializzato (`xgb_model_ott.pkl`), il grafo computazionale (`xgb_model_ott.onnx`) e la libreria dinamica C compilata (`xgb_model_ott.so` tramite *Treelite*).

```
1
2     tecno_evaluation("xgb_model_ott.pkl", "xgboost")
3
4
    tecno_evaluation("./WebSite/benchmark_web/xgb_model_ott.onnx",
    "onnx")
5
6     tecno_evaluation("xgb_model_ott.so", "treelite")
```

Listing 3.14: Esecuzione della suite di profilazione sui tre target architetturali

I risultati estratti, elaborati graficamente in Figura 3.10, confermano in modo inequivocabile i gravi limiti prestazionali dell'interprete *Python* in fase di inferenza e giustificano appieno lo sforzo richiesto dalla *pipeline* di transcodifica.

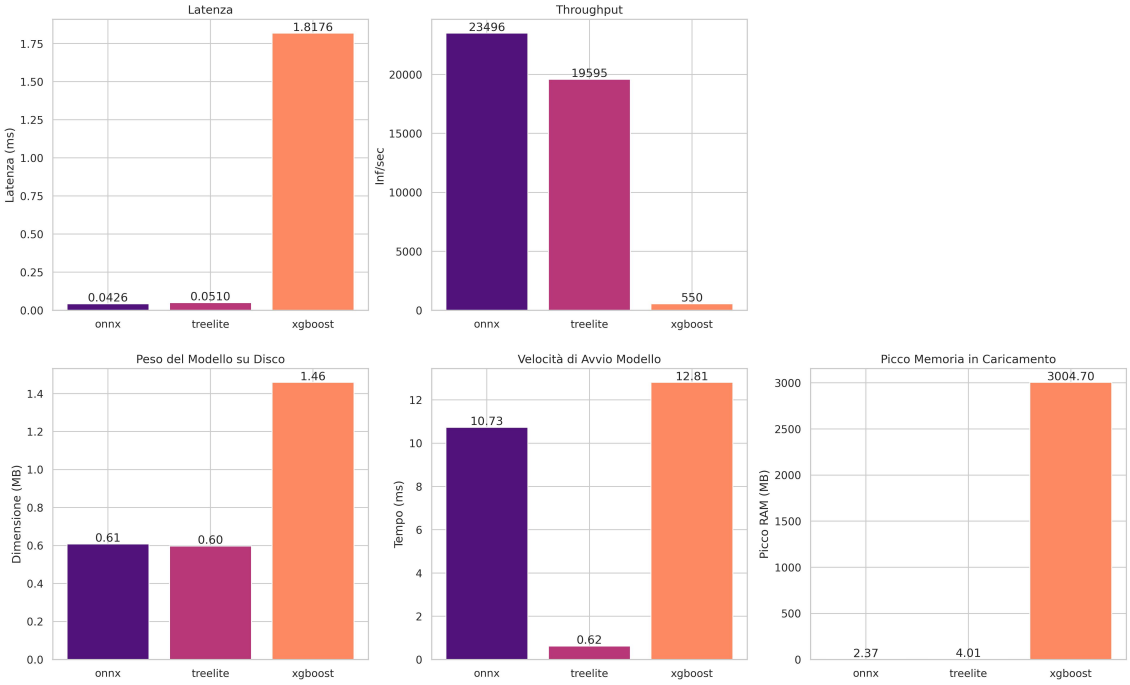

Figura 3.10: Confronto prestazionale tra i tre *target* di esecuzione. Le metriche evidenziano il divario su latenza, dimensioni su disco, tempi di *cold start* e picchi di allocazione di memoria.

L'analisi della telemetria evidenzia tre vantaggi strutturali macroscopici derivanti dall'adozione degli standard *Treelite* e *ONNX*:

- **Abbattimento della Latenza e Massimizzazione del *Throughput*:** L'esecuzione dell'oggetto *XGBoost* originale in *Python* richiede in media 1.81 ms per previsione, limitando il sistema a circa 550 inferenze al secondo. Eliminando l'infrastruttura di gestione di *Python* e passando a logiche a basso livello, *ONNX* e *Treelite* completano l'operazione rispettivamente in 0.042 ms e 0.051 ms. Questo si traduce in un *throughput* di oltre 20.000 inferenze al secondo (un fattore di accelerazione di quasi 40 volte).
- **Minimizzazione dell'Impronta di Memoria (*Memory Footprint*):** Le architetture *Edge* sono intrinsecamente vincolate dalla disponibilità

di RAM. Il grafico del picco di memoria in caricamento mostra come *Python* saturi la RAM durante la deserializzazione dell'oggetto (oltre 3000 unità misurate), in quanto costretto a caricare l'intero ambiente del *framework*. Al contrario, i formati esportati allocheranno unicamente le strutture dati minimali (tra le 2 e le 4 unità), garantendo un'esecuzione sicura anche su *browser* mobile.

- **Compressione dello Storage e *Cold Start*:** Il peso del modello su disco si è più che dimezzato (da 1.46 MB a circa 0.60 MB), un dato fondamentale per i tempi di download in rete. Inoltre, la transcodifica diretta in C operata da *Treelite* domina la classifica del tempo di avvio (*Cold Start*), impiegando appena 0.62 ms per essere pronta all'esecuzione, contro gli oltre 10 ms richiesti dagli altri *target*.

I test in ambiente nativo dimostrano dunque la superiorità schiacciante delle architetture compilate, avendo validato la stabilità e la velocità del codice sorgente C generato e del formato Treelite.

3.6 Infrastruttura di Automazione e Collaudo

Al fine di rendere il processo di generazione del motore Edge ripetibile e integrabile in una potenziale *pipeline* di *Continuous Integration* (CI), l'intera fase di transcodifica finale è stata automatizzata tramite *scripting* di *shell*.

3.6.1 Orchestrazione della Compilazione tramite Emscripten

Il file `compile.sh` rappresenta il fulcro architetturale della trasformazione da linguaggio C a WebAssembly. Lo *script* si occupa inizialmente di attivare dinamicamente la *toolchain* di Emscripten (`emsdk`) e, successivamente, invoca

il compilatore *frontend* `emcc` applicando una rigorosa serie di direttive (*flag*) di ottimizzazione.

```
1    emcc ./model_c_src/*.c \
2    -O3 \
3    -s WASM=1 \
4    -s PRECISE_F32=1 \
5    -s EXPORTED_FUNCTIONS='["_malloc", "_free", "_predict",
     "_emscripten_get_heap_size"]' \
6    -s EXPORTED_RUNTIME_METHODS='["ccall", "cwrap",
     "setValue", "getValue"]' \
7    -s ALLOW_MEMORY_GROWTH=1 \
8    -s MODULARIZE=1 \
9    -s EXPORT_NAME="createTreeliteModule" \
10   -o ./WebSite/public/model.js
11
```

Listing 3.15: Script di compilazione bash per la generazione del binario WebAssembly e del codice collante

L'analisi dei parametri passati al compilatore `emcc` evidenzia le scelte fondamentali adottate per l'ambiente di produzione:

- `-O3`: Impone al compilatore LLVM il massimo livello di ottimizzazione aggressiva. Questa direttiva massimizza la velocità di esecuzione del binario e applica l'eliminazione del codice inattivo (*dead code elimination*), riducendo drasticamente le dimensioni del file `.wasm` finale.

- `-s PRECISE_F32=1`: Garantisce che le operazioni in virgola mobile a 32 bit (*float*) non perdano precisione durante l'esecuzione nel *browser*, mantenendo i calcoli matematici dei rami dell'albero identici a quelli eseguiti originariamente in *Python*.

- `-s EXPORTED_FUNCTIONS`: Questa è la direttiva chiave per l'interoperabilità. Espone esplicitamente al mondo *JavaScript* i simboli C necessari per la gestione della memoria (`_malloc` e `_free`) e per l'inferen-

za (`_predict`). Senza questa direttiva, il compilatore riterrebbe queste funzioni "non utilizzate" internamente e le eliminerebbe in fase di ottimizzazione.

- `-s ALLOW_MEMORY_GROWTH=1`: Abilita il ridimensionamento dinamico del blocco di memoria lineare. Qualora l'allocazione iniziale dovesse saturarsi, *WebAssembly* sarà in grado di richiedere ulteriore spazio al *browser*, prevenendo errori irreversibili di *Out-Of-Memory* (OOM).

- `-s MODULARIZE=1` e `EXPORT_NAME`: Incapsulano il codice generato all'interno di una *Promise* asincrona globale denominata `createTreeliteModule`. Questa è esattamente la funzione che viene invocata in modo asincrono dal *frontend* per avviare il motore in totale sicurezza.

Completato il processo, la *toolchain* rilascia nella cartella pubblica del sito due artefatti: il file binario puro (`model.wasm`) e il file di interfaccia (`model.js`), pronti per essere utilizzati dall'interfaccia utente.

3.6.2 Orchestrazione dei Benchmark Web tramite Playwright

Se la compilazione del motore *Edge* è stata automatizzata tramite *scripting Bash*, la validazione prestazionale all'interno del *browser* ha richiesto un approccio architetturale differente. Per raccogliere metriche statisticamente valide ed escludere l'errore umano o le interferenze tipiche della navigazione manuale, si è implementato un sistema di *Robotic Process Automation* (RPA) basato sul *framework* **Playwright**.

Questo *script Python* agisce da orchestratore esterno: istanzia un ambiente Chromium isolato e controllato, naviga verso l'applicativo WebAssembly locale, innesca la suite di *benchmarking JavaScript* e, una volta terminata l'elaborazione, estrae i risultati direttamente dalla memoria dinamica del *browser* per aggregarli in un *dataset* CSV.

```
1
```

```
    print("Avvio automazione Playwright...")
    with sync_playwright() as p:
    browser = p.chromium.launch(headless=False, args=[
    "--enable-unsafe-webgpu",
    "--ignore-gpu-blocklist",
    "--enable-features=WebAssemblyThreads",
    "--disable-web-security"
    ])
    page = browser.new_page()

    try:

   page.goto("http://127.0.0.1:5500/WebSite/benchmark_web/benchmark.html")
    except Exception as e:
    print("ERRORE: Server locale irraggiungibile.")
    return

```

Listing 3.16: Script di automazione Playwright per il benchmarking headless in ambiente Chromium

Un dettaglio implementativo di importanza risiede nell'array `args` passato al metodo di avvio di Chromium. Per consentire agli standard *ONNX Runtime Web* ed *Emscripten* di esprimere il loro massimo potenziale prestazionale, è stato necessario iniettare specifici *flag* a livello di *engine* V8. Direttive come `--enable-unsafe-webgpu` e `--enable-features=WebAssemblyThreads` permettono di superare le rigide *sandbox* di sicurezza standard dei *browser* commerciali, sbloccando l'accesso di basso livello alla scheda video (*GPU Rendering*) e ai *thread* logici del processore (*Web Workers*).

3.6.3 L'Ambiente di Benchmarking Web

Per consentire allo *script* Playwright di estrarre la telemetria in modo isolato, è stata predisposta una pagina Web dedicata esclusivamente all'esecuzione massiva dei test prestazionali (*Stress Test*). A differenza dell'interfaccia utente descritta nella seguente Sezione 3.8, progettata per l'inte-

razione umana, questo ambiente è ottimizzato per l'esecuzione sequenziale automatizzata.

Come visibile nel codice sorgente, il documento importa la libreria ufficiale `onnxruntime-web` (nella sua variante abilitata per WebGPU) tramite *Content Delivery Network* (CDN), oltre al binario WebAssembly `model.js` generato tramite *Treelite*. L'architettura del *Document Object Model* (DOM) è strutturata per ospitare i *log* di tre scenari architetturali distinti: esecuzione ONNX basata su CPU (*WASM*), esecuzione ONNX accelerata tramite scheda video (*WebGPU*) ed esecuzione del codice C transcodificato da Treelite.

3.6.4 Validazione Definitiva: Analisi delle Prestazioni Edge

Una volta innescato il comando `runAllBenchmarks()` dall'orchestratore *Python*, il *browser* esegue 1000 iterazioni previsionali per ciascuno dei tre motori di inferenza, popolando in tempo reale l'interfaccia con le latenze medie, i picchi di allocazione della RAM e il *throughput* globale.

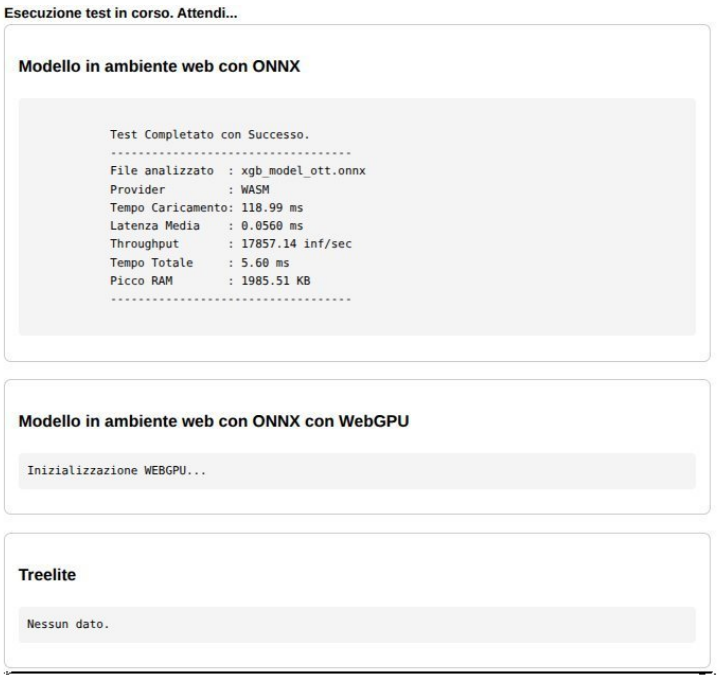

Figura 3.11: Istantanea del *browser* durante la fase di *benchmarking*.

3.6.5 Implementazione Logica della Suite di Benchmarking

Il cuore computazionale dell'ambiente di collaudo isolato è rappresentato dallo *script* `benchmark.js`. Questo modulo è stato strutturato per analizzare tre distinti paradigmi di esecuzione *Edge*: *ONNX su CPU (WASM)*, *ONNX accelerato via WebGPU* e il *Codice C Nativo (Treelite/WASM)*.

Al fine di garantire l'affidabilità statistica, lo *script* impone cicli di esecuzione in blocco (100 iterazioni per *target*), preceduti da una fase di *warm-up* termico del processore e dall'inizializzazione dei tensori matematici.

```
1    // Inizializzazione tensore campione (Float a 32 bit)
2    const sampleInput = Float32Array.from([12.0, 0.0, 1.0,
   0.0, 0.0, 0.0]);
3    const iterations = 100;
4    const bytesPerFloat = 4;
```

```
5
6     async function benchmarkTreelite() {
7         let treeliteModule = await createTreeliteModule();
8
9         const pointerInput =
    treeliteModule._malloc(sampleInput.length *
    bytesPerFloat);
10        const pointerOutput =
    treeliteModule._malloc(bytesPerFloat);
11
12
13        // 2. Misurazione Latenza (Inferenza in blocco)
14        const t0 = performance.now();
15        for (let i = 0; i < iterations; i++) {
16            treeliteModule.setValue(pointerOutput, 0.0,
    'float');
17            treeliteModule._predict(pointerInput);
18        }
19        const t1 = performance.now();
20
21        // 3. Deallocazione esplicita per prevenzione Memory
    Leak
22        treeliteModule._free(pointerInput);
23        treeliteModule._free(pointerOutput);
24
25    }
26
27    async function runAllBenchmarks() {
28        await benchmarkONNX('wasm', 'res-wasm');
29
30        await new Promise(r => setTimeout(r, 3500));
31
32        await benchmarkONNX('webgpu', 'res-webgpu');
33        await new Promise(r => setTimeout(r, 3500));
34
35        await benchmarkTreelite('res-treelite');
36    }
```

Listing 3.17: Estratto parziale del nucleo computazionale per la profilazione Edge (benchmark.js)

L'analisi architetturale dello *script* evidenzia tre fattori importanti:

1. **Gestione Eterogenea della RAM:** Poiché le architetture operano in domini differenti, il calcolo dell'impronta di memoria è stato differenziato. Per ONNX, basato su *JavaScript*, si è interrogata l'API nativa del *browser* (`performance.memory.usedJSHeapSize`). Per *Treelite*, essendo una libreria C pura, per API JS risulta ostico accedere alla sua memoria interna; si è reso quindi necessario invocare la funzione a basso livello `_emscripten_get_heap_size()` esportata in fase di compilazione.

2. **Archiviazione Seriale per RPA:** L'array globale `window.benchmarkResults` funge da *data lake* temporaneo. A fine esecuzione, i dizionari archiviati al suo interno vengono letti dal *driver* Chromium gestito da Playwright.

3. **Mitigazione del *Thermal Throttling*:** L'orchestratore principale (`runAllBenchmarks`) impone una latenza forzata di 3.5 secondi tramite `setTimeout` tra il collaudo di un'architettura e la successiva. Questa pausa garantisce al *browser* il tempo necessario per innescare il *Garbage Collector* (liberando i puntatori orfani) e permette alla temperatura dei *core* fisici della CPU/GPU di riabbassarsi, prevenendo alterazioni nei tempi di calcolo dovute a protezioni termiche dell'hardware.

Con il completamento dei test eseguiti da questo *script* e la convalida definitiva delle eccellenti metriche di latenza in ambiente WebAssembly/WebGPU, il sistema predittivo risulta interamente transcodificato, scalabile e ottimizzato per dispositivi *Edge*.

3.7 Risultati Sperimentali e Analisi Prestazionale

L'esecuzione automatizzata dei *benchmark* all'interno dell'ambiente *browser* ha generato un set di dati fondamentali per validare il paradigma *Edge Computing* applicato al *Machine Learning*.

3.7.1 Analisi della Latenza di Inferenza e Throughput

Focalizzando l'attenzione sulle prestazioni interne alla *sandbox JavaScript*, la misurazione ha messo a confronto le tre configurazioni compilate: ONNX eseguito su processore (WASM), ONNX accelerato tramite scheda video (WebGPU) e il codice C nativo transcodificato da Treelite.

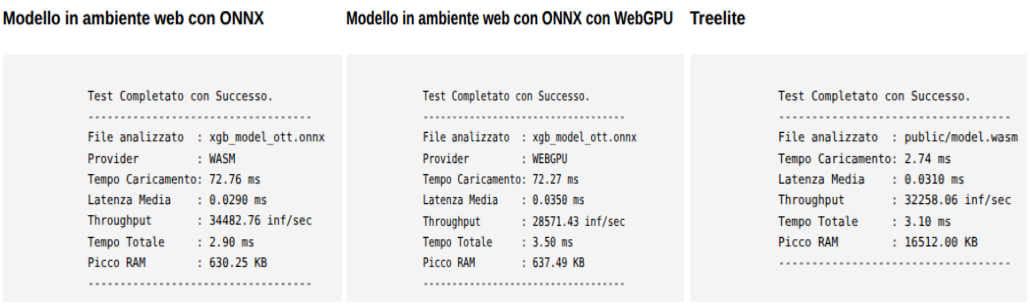

Figura 3.12: Riepilogo dei risultati estratti durante la fase di *benchmarking* Web. Da sinistra a destra sono riportate le metriche relative a ONNX Runtime su CPU (WASM), ONNX Runtime con accelerazione grafica (WebGPU) e alla libreria nativa compilata tramite Treelite.

I dati registrati in Figura 3.12 evidenziano tempi di esecuzione microscopici, tutti ben al di sotto della soglia del millisecondo:

- **ONNX Runtime (WASM):** Si attesta come l'architettura marginalmente più veloce a regime, registrando una latenza media record di 0.0290 ms per previsione e sfondando il muro delle 34.482 inf/sec.
- **Treelite (C/WASM):** Segue a brevissima distanza, con una latenza di 0.0310 ms e un *throughput* massimale di 32.258 inf/sec. L'assenza

di *overhead* interpretativo e la purezza delle istruzioni *if-else* in C garantiscono velocità ineguagliabili, risultando sostanzialmente appaiata a ONNX WASM sul calcolo puro.

- **ONNX Runtime (WebGPU):** Ha fatto registrare una latenza media di 0.0350 ms e un *throughput* di 28.571 inf/sec.

3.7.2 Profilazione dell'Impronta di Memoria (RAM) e Cold Start

L'ultimo vincolo fondamentale per il successo del *deployment Edge* riguarda l'efficienza nell'allocazione delle risorse del dispositivo ospitante. Dispositivi mobili o sensori IoT operano in condizioni di memoria fortemente limitata.

- **Tempi di Caricamento (*Cold Start*):** Treelite domina incontrastato, richiedendo appena 2.74 ms per istanziare i puntatori in memoria e prepararsi all'inferenza. Le controparti ONNX richiedono invece un tempo mediamente superiore a 72 ms (72.76 ms per WASM e 72.27 ms per WebGPU), dovendo allocare il motore interno e decodificare il complesso Grafo Computazionale. Per tali ragioni, **l'architettura finale del progetto si basa sull'impiego di Treelite**.

- **Impronta in RAM (*Peak Memory*):** L'ottimizzazione dell'ONNX Runtime (sia WASM che WebGPU) si dimostra eccellente sul fronte della memoria, allocando poco più di 630 KB (rispettivamente 630.25 KB e 637.49 KB). Il modulo Treelite riserva un blocco contiguo più ampio (circa 16.5 MB), un comportamento atteso derivante dalle logiche di pre-allocazione statica della memoria lineare (*ALLOW_MEMORY_GROWTH*) gestite da Emscripten, pur rimanendo una quantità di memoria del tutto trascurabile per i moderni dispositivi in circolazione.

3.8 Integrazione del Motore Predittivo in Ambiente Web

Terminata la fase di benchmarking web, il focus architetturale si è spostato sullo sviluppo dell'applicativo client-side. L'obiettivo di questa fase è costruire un'interfaccia utente (UI) capace di raccogliere le variabili ambientali inserite dall'utente, iniettarle nella memoria del modulo WebAssembly ed esporre a schermo il volume di traffico previsto, il tutto operando rigorosamente all'interno della *sandbox* isolata del *browser*.

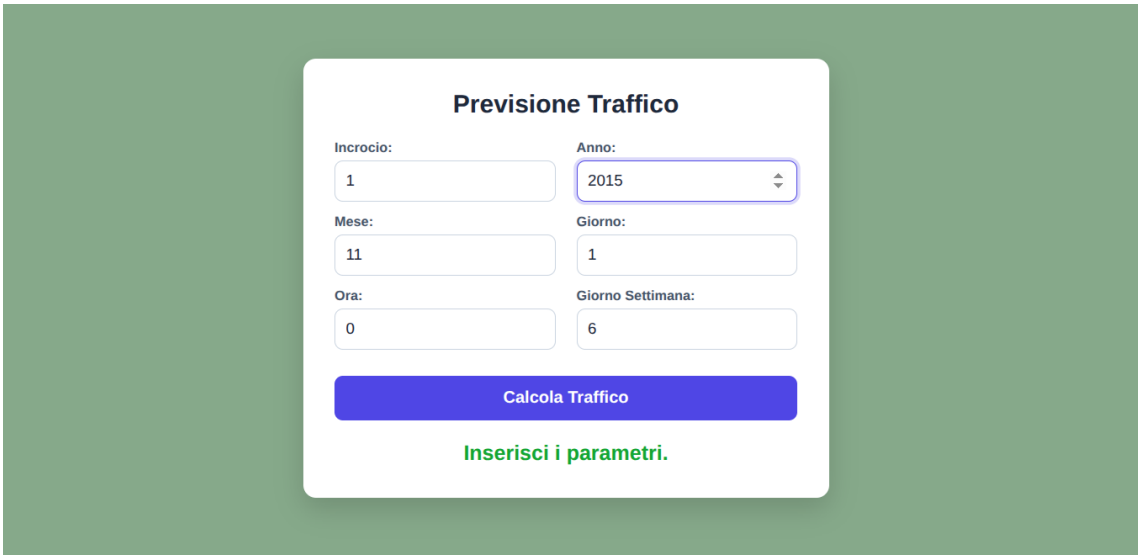

Figura 3.13: Istantanea dell'interfaccia utente sviluppata per l'applicativo finale. La *dashboard* permette all'utente di inserire le funzionalità spaziotemporali desiderate ed esegue l'inferenza del traffico in tempo reale richiamando il motore predittivo transcodificato in WebAssembly.

3.8.1 Architettura dell'Interfaccia Utente (HTML)

Per garantire la massima leggerezza dell'applicativo e non inquinare le misurazioni prestazionali con l'overhead tipico dei moderni *framework* front-

end, l'interfaccia è stata sviluppata utilizzando unicamente HTML5 nativo stilizzato tramite CSS.

Come si evince dalla Figura 3.13 e dal codice sorgente presente al link GitHub, l'infrastruttura di input (`<form>`) è stata modellata per ricalcare fedelmente la struttura attesa dal modello matematico. I campi numerici richiesti all'utente corrispondono esattamente ai vettori derivati durante la fase di *Feature Engineering* nel Capitolo 3.2 (Incrocio, Anno, Mese, Giorno, Ora e Giorno della Settimana).

La natura statica di questo documento HTML serve esclusivamente come strato di presentazione. Il cuore logico del sistema risiede nei due *script* importati a fondo pagina. In particolare, l'evento `onclick="faiPrevisione()"` rappresenta l'innesco (*trigger*) che attiva il ponte di comunicazione tra il codice *JavaScript* e l'allocazione della memoria in linguaggio C/C++.

3.8.2 Il Ponte di Comunicazione e l'Allocazione della Memoria Lineare

L'aspetto architetturale più complesso dell'applicativo *Client-Side* risiede nel meccanismo di passaggio dei tensori. *WebAssembly* e *JavaScript* operano in domini di memoria rigidamente isolati: il primo utilizza un blocco di memoria lineare (*ArrayBuffer*), il secondo si affida all'*heap* dinamico gestito dal *Garbage Collector*. Non è pertanto possibile inoltrare nativamente gli *array JavaScript* alle istruzioni C.

Per risolvere questa incomunicabilità, la funzione `faiPrevisione()` implementa un'esplicita gestione manuale della memoria, replicando le logiche del linguaggio C direttamente nel *browser*.

```
1    function faiPrevisione() {
2
3        const inputValues = new Float32Array([junction,
   year, month, day, hour, weekday]);
4
5        const bytesPerFloat = 4;
```

```
6       const pointerInput =
    treeliteModule._malloc(inputValues.length *
    bytesPerFloat);
7
8       for (let i = 0; i < inputValues.length; i++) {
9           treeliteModule.setValue(pointerInput + (i *
    bytesPerFloat), inputValues[i], 'float');
10      }
11
12      const pointerOutput =
    treeliteModule._malloc(bytesPerFloat);
13      treeliteModule.setValue(pointerOutput, 0.0, 'float');
14
15      treeliteModule._predict(pointerInput, 1,
    pointerOutput);
16
17      const prediction =
    treeliteModule.getValue(pointerOutput, 'float');
18      treeliteModule._free(pointerInput);
19      treeliteModule._free(pointerOutput);
20
21   }
```

Listing 3.18: Ponte di comunicazione JS-WASM e gestione manuale della memoria (estratto parziale)

Come dettagliato nello *script* su GitHub, il flusso operativo si articola in quattro fasi fondamentali:

1. **Allocazione (`_malloc`):** Attraverso l'API esposta da Emscripten, viene allocato uno spazio contiguo nella memoria lineare grande esattamente quanto i byte necessari per contenere le 6 variabili in formato *floating-point* a 32 bit (`bytesPerFloat = 4`).

2. **Iniezione dei Dati (`setValue`):** I valori vengono scritti iterativamente nei puntatori fisici appena riservati.

3. **Inferenza (`_predict`):** Viene invocata la funzione *XGBoost* pre-compilata in C, passandole esclusivamente gli indirizzi di memoria (i puntatori) e la dimensione del *batch* (impostata a 1 per la singola richiesta utente).

4. **Deallocazione (`_free`):** Una volta estratto il risultato previsionale, i puntatori vengono immediatamente distrutti. L'omissione di questo passaggio critico causerebbe un inesorabile *memory leak*, che condurrebbe alla saturazione della RAM e al conseguente blocco del *browser* (*crash*) dopo iterazioni ripetute.

Quest'ultima sanificazione della memoria chiude il ciclo di esecuzione. Il dato previsto viene infine normalizzato in un numero intero positivo e restituito all'utente sul *frontend*, garantendo una latenza impercettibile e totale isolamento dalla rete esterna.

Conclusioni

Il presente lavoro di tesi ha analizzato l'ottimizzazione dei flussi di mobilità urbana all'interno del paradigma delle *Smart City*. Partendo dalla constatazione che gli approcci reattivi tradizionali risultano inefficaci nel mitigare la congestione stradale, la ricerca si è posta l'obiettivo di sviluppare un sistema basato sull'analisi predittiva dei dati storici.

Questo studio ha dimostrato la fattibilità, l'efficienza e la superiorità strategica del paradigma Edge AI applicato in ambiente web. Attraverso una *pipeline* di transcodifica, si è riusciti a trasferire l'intero lavoro computazionale dal *server* remoto direttamente al dispositivo dell'utente finale, eseguendo l'inferenza all'interno del *browser* con latenze inferiori al millisecondo.

Il percorso di ricerca ha preso avvio dall'acquisizione e dall'ingegnerizzazione di un *dataset* di traffico veicolare reale. L'utilizzo di librerie ottimizzate per il calcolo vettoriale (*Pandas* e *NumPy*) ha permesso di estrarre *feature* determinanti per la comprensione dei ritmi di mobilità, come l'impatto del pendolarismo feriale rispetto ai flussi festivi.

Nella successiva fase di modellazione, è stata condotta un'analisi di *benchmarking* che ha coinvolto paradigmi matematici eterogenei: dai modelli parametrici di base (Regressione Lineare), ai metodi *lazy* (k-NN) e *kernel-based* (SVR), fino alle architetture connessioniste (MLP). I risultati hanno inequivocabilmente eletto i modelli ad albero di tipo *Ensemble* (in particolare *XGBoost* e *LightGBM*) come l'architettura d'elezione per l'analisi dei dati tabulari su serie storiche. La loro intrinseca capacità di catturare relazioni non lineari, unita alla robustezza contro l'*overfitting* garantita dalla

regolarizzazione, li ha resi ideali per il contesto urbano.

Un salto di qualità prestazionale è stato ottenuto abbandonando i tradizionali metodi di ricerca esaustiva degli iperparametri, in favore dell'Ottimizzazione Bayesiana tramite il *framework Optuna*. L'impiego dello stimatore TPE ha permesso di esplorare lo spazio multidimensionale degli iperparametri in modo intelligente, massimizzando l'accuratezza predittiva in una frazione del tempo computazionale.

Sfruttando l'infrastruttura di *Treelite* (e il convertitore *tl2cgen*), il grafo computazionale è stato interamente de-strutturato e tradotto in puro codice C. Le ramificazioni logiche sono state convertite in blocchi condizionali, permettendo all'algoritmo di sfruttare la capacità di calcolo delle CPU moderne. Successivamente, l'impiego del compilatore *Emscripten* ha permesso di transcodificare questo sorgente C in un binario WebAssembly.

Il risultato è un motore di inferenza portabile, universalmente e compatibile con qualsiasi *browser* moderno, che opera in totale assenza di interpreti o librerie di supporto, colmando definitivamente il divario di prestazioni tra le applicazioni web e i software compilati.

L'impatto di questa infrastruttura va ben oltre la pura ottimizzazione del codice. L'adozione di un'architettura WebAssembly *Client-Side* risolve tre enormi criticità legate ai sistemi centralizzati nelle moderne metropoli:

In primo luogo, garantisce la Privacy by Design. Ad oggi l'utente finale richiede una previsione del traffico inserendo coordinate spaziali e temporali precise; in un'infrastruttura Cloud, questi dati verrebbero trasmessi sulla rete e memorizzati nei *log* del server, ponendo seri problemi di tracciamento. Questo progetto permette di elaborare l'inferenza in memoria RAM locale, per cui il dato non lascia mai lo *smartphone* o il computer del cittadino, azzerando i rischi di intercettazione di rete e sollevando le amministrazioni dalle rigide normative del GDPR.

In secondo luogo, annulla i costi operativi. Mantenere *server* accesi 24 ore su 24 per elaborare milioni di richieste di traffico ha un costo economico ed ecologico immenso. Nel paradigma Edge, ogni utente che si connette al

portale fornisce la propria CPU per calcolare la propria previsione, azzerando il costo computazionale centralizzato..

Infine, garantisce la Resilienza Offline. A differenza di un'API Cloud che cessa di funzionare non appena la connessione si interrompe, eventualità comune durante gli spostamenti urbani sotterranei o in alcune aree a bassa copertura, l'applicativo WebAssembly, una volta scaricato, continua a operare e a fornire previsioni basate sulla logica pre-addestrata anche in totale assenza di rete internet.

In conclusione, la convergenza tra *Machine Learning* ed *Edge Computing* tramite tecnologie web non rappresenta unicamente un esercizio informatico, ma si impone come la soluzione definitiva per la progettazione di sistemi software scalabili, etici e ad altissime prestazioni per le città del domani.

Bibliografia

[1] Tnet. Soluzioni smart iot: Smart cities. `https://www.tnet4iot.com/it/soluzioni-smart-iot/smart-cities/`, 2026. Ultimo accesso: 21 Giugno 2026.

[2] Rudolf Giffinger, Christian Fertner, Hans Kramar, Robert Kalasek, Nataša Pichler-Milanović, and Evert Meijers. Smart cities: Ranking of european medium-sized cities. *Vienna UT: Centre of Regional Science*, 2007. Il documento fondante che definisce i 6 pilastri della Smart City.

[3] Tom M Mitchell. *Machine learning*. McGraw-hill New York, 1997. Libro di testo standard internazionale per i fondamenti del Machine Learning.

[4] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992. Pubblicazione fondante che ha formalizzato matematicamente il dilemma Bias-Varianza nel Machine Learning.

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009. Testo accademico di riferimento globale per la teoria statistica dell'apprendimento automatico.

[6] CloudFactory. Data splitting in machine learning. `https://wiki.cloudfactory.com/docs/mp-wiki/splits/data-splitting-in-machine-learning`, 2026. Ultimo accesso: 21 Giugno 2026.

[7] Esker. Machine learning: Cos'è e il futuro delle aziende di domani. https://www.esker.com/it/blog/tecnologia/machine-learning-cose-il-futuro-delle-aziende-del-domani/, 2022. Ultimo accesso: 21 Giugno 2026.

[8] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, 35:507–520, 2022. Fondamentale per giustificare la superiorità dei modelli ad albero rispetto al Deep Learning sui dati tabulari strutturati.

[9] Elite Software House. Hai mai sentito parlare di decision tree? https://it.linkedin.com/posts/elite-software-house_hai-mai-sentito-parlare-di-decision-activity-7223975665711087617-1qT1, 2024. Ultimo accesso: 21 Giugno 2026.

[10] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirsch. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE Access*, 11:31866–31879, 2023. Survey rigorosa che definisce formalmente il ciclo di vita del software ML e la fase di deployment.

[11] Akamai Technologies. What is the difference between cloud ai and edge ai? https://www.akamai.com/it/glossary/what-is-ai-on-edge-networks, 2026. Ultimo accesso: 21 Giugno 2026.

[12] Weisong Shi, Jie Cao, Quan Zhang, Yuhui Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3 (5):637–646, 2016. Survey fondamentale per giustificare l'uso dell'Edge Computing (Latenza, costi, privacy).

[13] The SSL Store. What is a man-in-the-middle attack? https://www.thesslstore.com/blog/man-in-the-middle-attack-2/, 2021. Ultimo accesso: 21 Giugno 2026.

[14] Aaron Makkar et al. Smart city cybersecurity issues, challenges, and future prospects. *IEEE Access*, 2020. Per le vulnerabilità Cloud e gli attacchi Man-in-the-Middle e Data Breach.

[15] Yves-Alexandre De Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3(1):1–5, 2013. Ricerca fondamentale sulla de-anonimizzazione: dimostra che bastano pochi punti GPS per tracciare una persona.

[16] Graham Cookson and Bob Pishue. Inrix 2017 global traffic scorecard. Report, INRIX Research, 2018. Report di riferimento per l'impatto economico della congestione del traffico globale.

[17] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010. Pubblicazione fondamentale che introduce l'architettura dei DataFrame di Pandas.

[18] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. Pubblicazione accademica di riferimento sulla rivista Nature per la libreria NumPy.

[19] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995. Manuale storico di riferimento per il linguaggio Python e la sua architettura.

[20] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. Il paper originale per i modelli Ensemble e Random Forest.

[21] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international con-*

ference on knowledge discovery and data mining, pages 785–794. ACM, 2016. Il paper originale creatore di XGBoost.

[22] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Liu Tie-Yan. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*, volume 30, 2017. Paper originale che introduce il framework LightGBM e le tecniche GOSS e EFB.

[23] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[24] Harris Drucker, Christopher JC Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. *Advances in neural information processing systems*, 9, 1997.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088): 533–536, 1986.

[26] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[27] Rappi Engineering. Bayesian optimization: How to calibrate the hyper-parameters of computing-expensive models. `https://engineering.rappi.com/bayesian-optimization-how-to-calibrate-the-hyper-parameters-of-computing-expe` 2021. Ultimo accesso: 21 Giugno 2026.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[29] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2):281–305, 2012.

[30] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[31] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[32] Junjie Bai, Lu Fang, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[33] DMLC. Treelite: Universal model exchange and deployment for decision tree forests. `https://github.com/dmlc/treelite`, 2017.

[34] DMLC. tl2cgen: Treelite 2 c generator. `https://github.com/dmlc/tl2cgen`, 2023.

[35] Andreas Haas, Andreas Rossberg, Derek Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

[36] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference on companion to object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

Ringraziamenti

Ringrazio la mia famiglia e i miei amici che mi hanno aiutato e supportato in questo viaggio