UNIVERSITÀ DEGLI STUDI DI GENOVA

SCUOLA POLITECNICA

DITEN

**Dipartimento di Ingegneria Navale, Elettrica,
Elettronica e delle Telecomunicazioni**

TESI DI LAUREA MAGISTRALE
IN
INGEGNERIA ELETTRONICA

Progetto e realizzazione di un sistema per lo sviluppo di modelli di Machine Learning su dati IoT

Relatori:

Prof. Riccardo Berta
Prof. Francesco Bellotti

Candidato:

Niccolò Monti

Dicembre 2020

Mi è doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito, con il loro instancabile supporto, alla realizzazione dello stesso.

In primis, un ringraziamento speciale ai miei relatori, i professori Berta Riccarto e Bellotti Francesco, per la loro immensa pazienza, per i loro indispensabili consigli, ma soprattutto per le conoscenze trasmesse durante tutto il percorso di stesura dell'elaborato.

Ringrazio infinitamente la mia famiglia che mi ha sempre sostenuto, appoggiando ogni mia decisione, e che mi ha permesso di portare a termine questo percorso di studi universitari.

Ringrazio la mia fidanzata Matilde per avermi supportato e sopportato durante tutto questo periodo. Se sono riuscito ad arrivare in fondo, lo devo a lei.

Infine, ringrazio i miei compagni di corso che mi hanno aiutato in questi anni, soprattutto in quest'ultimo periodo di didattica a distanza.

INDICE

ELENCO DELLE FIGURE

1

INTRODUZIONE

1.1 CONTESTO

Internet of Things (IoT) è un neologismo utilizzato in telecomunicazioni, un termine di nuovo conio nato dall'esigenza di dare un nome agli oggetti reali connessi ad internet. Il significato e definizione di IoT si esprime bene con degli esempi: IoT è ad esempio un frigorifero che ordina il latte quando "si accorge" che è finito. IoT è una casa che accende i riscaldamenti appena ti sente arrivare. Questi sono esempi di IoT, ovvero di oggetti che, collegati alla rete, permettono di unire mondo reale e virtuale.

Il termine IoT ("Internet of Things", o letteralmente "internet delle cose") viene utilizzato la prima volta da *Kevin Ashton*, ricercatore presso il MIT, Massachussets Institute of Technology. Ma anche se il termine è nuovo, si parla di questi concetti già da molto tempo, in sostanza dalla nascita di internet e del web semantico (un web fatto di "cose", non di righe di codice: "things, not strings").

Ma cosa significa IoT nel concreto? Con Internet of Things si indicano un insieme di tecnologie che permettono di collegare a Internet qualunque tipo di apparato. Lo scopo di questo tipo di soluzioni è sostanzialmente quello di monitorare e controllare e trasferire informazioni per poi svolgere azioni conseguenti.

La raccolta dei dati sempre più ampia dai dispositivi IoT sottolinea la necessità di uno sviluppo efficiente di applicazioni efficienti. I servizi cloud IoT all'avanguardia sono potenti, ma le migliori soluzioni sono proprietarie e c'è una crescente domanda di interoperabilità e standardizzazione.

Nell'ecosistema IoT, i dati raccolti dal campo alimentano una varietà di applicazioni in domini industriali (ad esempio, monitoraggio, previsione, manutenzione, sorveglianza, ecc.), ma sempre più anche in domini domestici. Secondo una previsione di Gartner, 25 miliardi di cose connesse saranno in uso entro il 2021.

Ma come fare a rendere i dispositivi IoT in grado di prendere decisioni? Stanno aumentando sempre di più gli impieghi dell'*Intelligenza Artificiale* (AI) all'interno delle soluzioni di IoT. In particolare, gli algoritmi di *Machine Learning* usano metodi matematico-computazionali per apprendere informazioni direttamente dai dati, senza modelli matematici ed equazioni predeterminate, migliorando le loro prestazioni in modo "adattivo" man a mano che gli "esempi" da cui apprendere aumentano.

Il Machine Learning premette quindi ai computer (in senso generico) di imparare dall'esperienza (da intendersi non nell'accezione "umana" ma pur sempre riferito ad un programma informatico). C'è apprendimento quando le prestazioni del programma migliorano dopo lo svolgimento di un compito o il completamento di un'azione (anche errata, partendo dall'assunto che anche per l'uomo vale il principio "sbagliando si impara").

1.2 OBIETTIVO

Nel mondo IoT abbiamo una sempre maggiore quantità di sensori che consentono di raccogliere una vastità di informazioni. Possiamo pensare all'esempio di un'automobile che (ad esempio), è in grado di accendere le luci in modo automatico rilevando, tramite un sensore, la quantità di luce ambientale.

Questo concetto possiamo pensare di ampliarlo, considerando di raccogliere dati da più sensori, e poter prendere decisioni, sempre più complesse, basate sulle esperienze passate.

L'obiettivo di questo progetto, quindi, è quello di permettere di utilizzare algoritmi noti di Machine Learning a grandi quantità di dati raccolti da dispositivi IoT.

Per questo motivo, la Tesi, è parte di due progetti più grandi:

- *Measurify*, che è un'Application Programming Interface (API) per la gestione di cose intelligenti negli ecosistemi IoT.
- *Edge Learning Machine*, che è un framework per i dispositivi "Edge", che gli consente di fare inferenze utilizzando algoritmi noti di Machine Learning.

Lo scopo finale del progetto, quindi, è quello di progettare e realizzare un supporto Machine Learning (utilizzando il framework ELM) per il progetto Measurify, e consentirgli di fare predizioni per le sue collezioni, nel contesto in cui viene utilizzato.

2

STATO DELL'ARTE

La letteratura offre un'ampia copertura dei framework IoT. Cheruvu et al. [4] forniscono un sondaggio classificando tali strumenti in base a un obiettivo di consumo, industriale o gestibilità. Atamani et al. [5] forniscono una revisione di diversi framework e piattaforme IoT disponibili, analizzando criteri quali: sicurezza, analisi dei dati e supporto della visualizzazione.

Al momento sono disponibili sul mercato potenti soluzioni per diversi tipi di servizi IoT.

Il servizio cloud Amazon Web Services per Internet of Things (AWS IoT) presenta moduli di architettura per la gestione dei dati, connettività e controllo dei dispositivi, analisi e rilevamento degli eventi. Sul back-end è disponibile uno stack di funzioni: DynamoDB, Kinesis, Lambda, S3, SNS, SQS, ecc. Poiché AWS è un provider di servizi cloud, sono già integrati più servizi di elaborazione dati. L'integrazione dell'applicazione nella piattaforma è supportata tramite vari servizi di messaggistica e accodamento. AWS IoT non distingue tra sensori, attuatori e dispositivi, poiché si concentra sul concetto di cose. Tuttavia, ci sono attributi personalizzabili limitati per le cose, il che mette alla prova la gestibilità e aumenta la latenza.

La piattaforma cloud Microsoft Azure consente agli sviluppatori di creare programmi basati su cloud utilizzando una piattaforma commerciale SaaS (Software as a Service). Azure Sphere contiene strumenti per Edge SDK per abilitare la connettività edge-to-cloud sicura.

Altri framework commerciali, come Google Cloud e Bluemix, hanno una varietà di servizi cloud IoT. Nonostante le loro differenze in termini di prestazioni ed efficienza nella pratica, Zúñiga - Prieto et al. [6] indicano i lunghi tempi di implementazione come una questione condivisa tra tali framework. Sosteniamo che una struttura dei contenuti più focalizzata potrebbe facilitare gli sviluppatori, specialmente per le applicazioni che si occupano di misurazioni.

In letteratura sono stati presentati diversi framework di dati IoT per trattare specifici domini di applicazioni IoT. Esempi recenti riguardano la medicina legale, le case intelligenti e le città intelligenti.

In un approccio più generale, Jiang et al. [7] ha proposto un framework che si occupi delle tipiche sfide dell'IoT (grande volume di dati, diversi tipi di dati, rapida generazione di dati, requisiti complicati, ecc.). Per i dati strutturati, propongono un modello di gestione del database che combina ed estende più database e fornisce interfacce di programmazione di applicazioni (API) ad accesso unificato. Per i dati non strutturati, il framework avvolge ed estende l'Hadoop Distributed File System (HDFS) basato sul modello di repository di file per implementare la gestione delle versioni e l'isolamento dei dati multi-tenant. Un modulo di configurazione delle risorse supporta la gestione dei dati statici e dinamici in termini di meta-modello predefinito. Pertanto, le risorse dati e i servizi correlati possono essere configurati in base ai requisiti del tenant.

Più recentemente, [8] ha presentato un framework funzionale che identifica le aree di acquisizione, gestione, elaborazione e mining dei big data IoT e diversi moduli tecnici associati sono definiti e descritti in termini di caratteristiche e capacità chiave.

Fu et al. [9] si occupano in particolare dell'efficienza e della sicurezza dell'archiviazione dei dati IoT, proponendo un framework che mantiene i dati sensibili al fattore tempo (ad es. Informazioni di controllo) sull'edge e invia gli altri (ad es. Dati di monitoraggio) al cloud.

Analogamente a Measurify, [10] propone un framework, che supporta gli sviluppatori nella modellazione di cose intelligenti come risorse web. Il framework supporta la definizione e la progettazione del tipo di risorsa, software generico per operazioni su risorse Web, una mappatura tra risorse Web e origini dati e strumenti di programmazione e pubblicazione. La valutazione degli utenti ha riguardato l'uso di IDN-Studio (un'applicazione web con interfaccia grafica per la progettazione e la gestione delle risorse Web) per personalizzare e migliorare la rappresentazione di un Punto di interesse all'interno dell'applicazione mySmartCity e valutarne il rendering eseguito da IDN-Viewer.

Sharm e Wang, [11], individuano quattro caratteristiche principali dei dati IoT nelle piattaforme cloud: alta eterogeneità multisorgente, dinamica su vasta scala, basso livello con semantica debole, imprecisione. Queste caratteristiche sono importanti, in quanto evidenziano le caratteristiche chiave che dovrebbero essere fornite da un efficace framework di dati IoT (ad esempio, caratterizzazione della sorgente, varietà di configurazioni/aggregazioni dei dati sorgente, calcolo dei valori anomali).

3

MEASURIFY

3.1 INDRODUZIONE A MEASURIFY

Measurify è un API cloud-based, astratta e orientata alla misurazione per la gestione di cose intelligenti negli ecosistemi IoT. Measurify si concentra sul concetto di misurazione, poiché questo tipo di dati è molto comune nell'IoT, il che rende più facile ed efficace il processo di astrazione necessario per indirizzare diversi domini e contesti operativi.

3.2 CONCETTI PRINCIPALI

Measurify è stato progettato per rappresentare il contesto dell'applicazione e i suoi elementi come oggetti software correlati, su cui costruire applicazioni. Questi oggetti sono modellati come risorse, con modelli e funzionalità propri, accessibili tramite una serie di interfacce API RESTful.

Al centro di queste risorse ci sono gli elementi essenziali comuni nell'ambiente IoT: *Thing* (Cosa), *Feature* (Caratteristica), *Device* (Dispositivo) e *Measurement* (Misurazione).

- Una *Thing* rappresenta l'oggetto di una *Measurement*.
- Una *Feature* descrive la quantità (tipicamente fisica) misurata da un *Device*.
- Ogni elemento in una *Feature* ha un nome e un'unità.
- Un *Device* è uno strumento che fornisce *Measurement* relative a una *Thing*.
- Una *Measurement* rappresenta un valore di una *Feature* misurata da un *Device* per una *Thing* specifica.

La figura 1 mostra le relazioni tra le diverse risorse, evidenziando il ruolo centrale del concetto di Misura. La figura fornisce anche un semplice esempio nel contesto della raccolta di informazioni meteorologiche (ad es. Temperatura e velocità del vento).

Il concetto di *Measurement* astrae i valori inviati e recuperati dal cloud. La sua struttura deve corrispondere al tipo di *Feature* misurata. Ogni *Measurement* è un vettore di campioni: possono essere campioni raccolti in momenti diversi (presi ad intervalli specificati dal campo "delta"), un singolo valore o un insieme di informazioni statistiche (es. media, stdev, ecc.). Ogni campione può essere uno scalare (ad esempio una temperatura), un vettore (ad esempio l'orientamento nello spazio) o un tensore di numeri (ad esempio punti multidimensionali generali).

La risorsa *Feature* viene utilizzata per convalidare la matrice di valori di ciascuna *Measurement* ricevuta.

É inoltre definita la risorsa *Computation*, che esegue un calcolo post-elaborazione su misure, sfruttando la capacità del server cloud. Il risultato di una *Computation* è strutturato e salvato

come *Measurement*, consentendo così ulteriori lavorazioni. Attualmente sono disponibili un insieme di *Computation* (tipicamente statistiche), identificabili dall'attributo "code":

- *stat*: che include massimo, minimo, media, mediana, deviazione standard e varianza
- *quartiles*: primo quartile, terzo quartile
- *histograms*

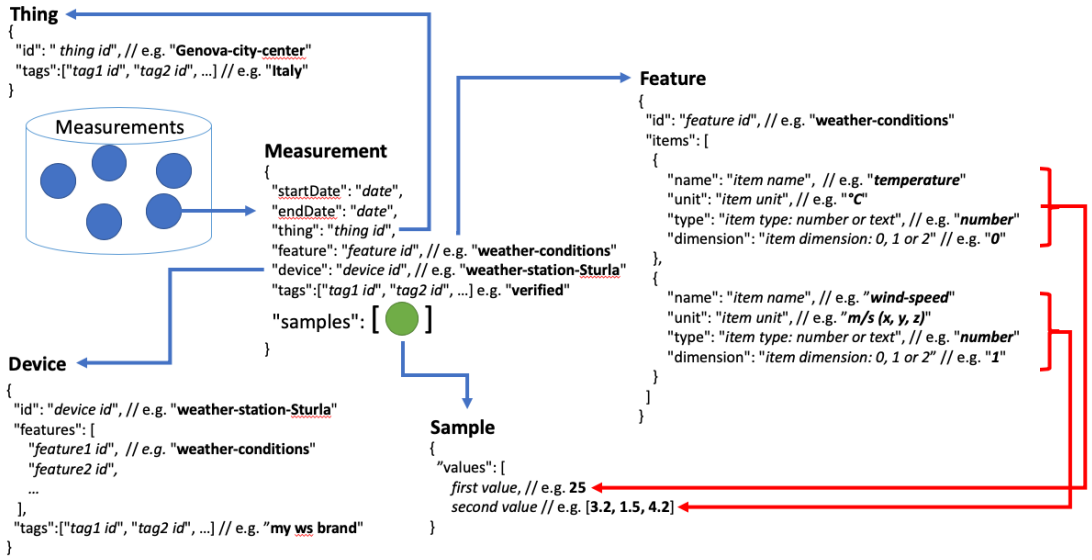


Figura 1: Struttura di Measurify con esempio

3.3 API RESTFUL

3.3.1 IMPLEMENTAZIONE

Il servizio è implementato in NodeJS (basato su JavaScript), utilizzando il framework "Express.js".

Il framework garantisce che tutte le risorse esposte possano essere manipolate tramite i metodi HTTP (GET, POST, PUT e DELETE). Codici di risposta standard sono stati definiti per ogni metodo come codici di successo (2xx) e codici di errore client (4xx) con informazioni nel messaggio di risposta adatte.

Per l'archiviazione dei dati, viene adottato un sistema di gestione di database basato su documenti (DBMS) come MongoDB. I database NoSQL forniscono una serie di funzionalità che database relazionali non possono fornire, come scalabilità orizzontale, memoria, indice distribuito e modifica dinamica dello schema dei dati.

In archivio, le risorse corrispondono alle collezioni nel database. Ogni risorsa richiede una serie di campi, alcuni dei quali obbligatori, mentre altri facoltativi. All'interno dell'API le risorse vengono implementate attraverso due fasi: schema e controller. Lo schema, necessario per il controllo dei dati, data la natura senza schema di MongoDB, definisce la struttura delle risorse,

mentre il controller implementa la funzionalità della risorsa. Uno schema di risorsa descrive i campi, compresi i loro tipi, valori predefiniti e riferimenti ad altri campi di risorsa. I campi che fanno riferimento ad altre risorse hanno funzione di convalida implementate all'interno dello schema della risorsa. Il controller definisce i metodi HTTP che sono supportati dalla risorsa (in genere: GET per il recupero della risorsa, POST per l'inserimento, PUT per l'aggiornamento e DELETE per la rimozione). Il controller di *Computation*, inoltre, esegue calcoli incrementali per evitare di esaurire le risorse del sistema (ad esempio memoria), poiché i calcoli vengono generalmente eseguiti su enormi quantità di dati.

3.3.2 SISTEMA DI LAVORO

Quando l'API viene inizializzata la prima volta, si connette all'archivio del server e crea il database "Measurify-DB" con una collezione all'interno. Quella collezione è la collezione degli "utenti", ed è essenziale avere un utente amministratore per creare altri utenti e altre collezioni. Questa prima istanza di un utente è predefinita all'interno del codice sorgente. L'utente "admin" può crearne altri due tipi: "provider" e "analyst". Mentre l'amministratore ha pieno accesso al DB, l'utente "provider" ha accesso limitato dipendente dalla proprietà (consentito POST *Measurements* e GET solo su i record di proprietà dell'utente). L'utente "analyst" può accedere solo alle *Measurements* (metodo GET).

Il meccanismo di autorizzazione supporta anche un altro controllo, che è l'autenticazione. Si basa su un token Web JSON (JWT), un pass di sicurezza che scade dopo un timeout configurabile. Per ottenere un JWT gli utenti devono effettuare un POST alla risorsa *login* con le credenziali fornite. L'API risponderà con un JWT con il livello di autorizzazione specifico dell'utente richiedente.

3.3.3 FLUSSO DI LAVORO

Il primo passo consiste nella modellazione del dominio, dove gli oggetti campo vengono mappati alle risorse dell'API. In questa fase il progettista deve definire le *Features* (ad esempio, tipi di misurazioni), i *Devices* (as esempio, strumenti di misura), i *Tags* (cioè etichette che possono essere attaccate come attributi di altre risorse) e *Constraints* (ad esempio, relazioni tra gli elementi del DB). Nella *configurazione*, le risorse del modello progettate sopra sono codificate in modo semplice in un file .json che viene POSTato alle API del framework, in modo da creare la struttura Application DB (ADB). Nella fase di *regime*, il framework gestisce l'ADB, consentendo il dinamico inserimento/aggiornamento di utenti/cose/misure sul campo e richieste di calcolo. La struttura dell'ADB può essere aggiornata anche durante la fase di regime.

4

EDGE LEARNING MACHINE

Edge Learning Machine è un framework Open-Source composto da due moduli:

- *Desk-LM*: un ambiente python per l'addestramento e il test di modelli di Machine Learning.
- *Micro-LM*: una libreria in linguaggio C, per l'inferenza su dispositivi "Edge", che utilizza i modelli creati da Desk-LM.

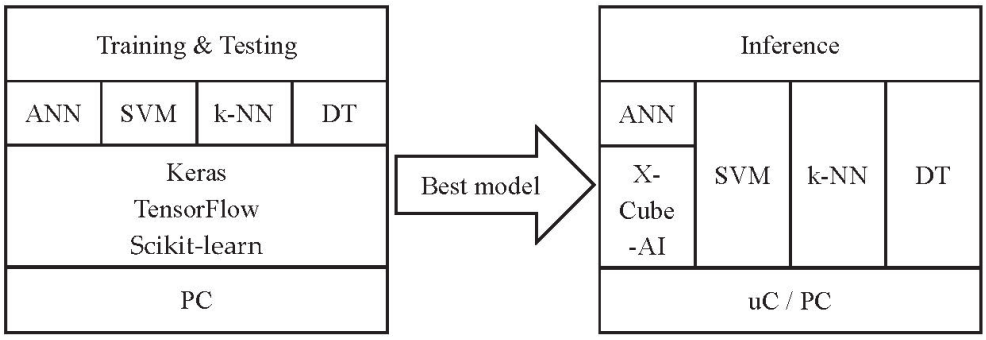


Figura 2: Struttura di Edge Learning Machine

4.1 INTRODUZIONE A DESK-LM

La funzionalità principale di *Desk-LM*, come già detto, è quella di addestrare modelli di Machine Learning a partire da un dataset (fornitogli come file .csv). Inoltre, è un grado di fare predizioni su nuovi campioni, utilizzando i modelli creati, che possono quindi essere salvati e mantenuti in memoria.

Attualmente, il software, implementa diversi algoritmi, che sono:

- *Linear Support Vector Machine* (Linear SVM)
- *Decision Tree*
- *K-Nearest Neighbors* (K-NN)
- *Artifical Neural Network* (ANN)
- *Holt-Winters Triple Esponential Smoothing* implementation for time series (TripleES)

ma la libreria è in continua espansione.

L'utente, una volta scelto il dataset, deve specificare quale algoritmo vuole implementare e, inoltre, ha la possibilità di impostare un insieme di parametri (specifici per ogni algoritmo) sia come valori puntuali, sia come array di valori sia come range di valori, in modo tale da addestrare l'algoritmo utilizzando il modello migliore tra tutte le combinazioni degli stessi.

É inoltre possibile esportare tale modello per l'utilizzo su dispositivi "Edge", sfruttando la seconda parte del software *Micro-LM*.

4.2 CONFIGURAZIONE DI DESK-LM

La configurazione di Desk-LM viene fatta in modo manuale prima dell'avvio del software, utilizzando dei file ".json" nei quali vanno inseriti i parametri di configurazione scelti.
All'avvio, nella *command line*, è possibile specificare le opzioni desiderate come i percorsi in cui si trovano i file precedentemente configurati.

Nel dettaglio, si può configurare un file per ciascuno delle seguenti opzioni:

- *Dataset*
- *Estimator*
- *Preprocessing*
- *Validation*
- *Output*
- *Prediction*
- *Storage*

4.2.1 CONFIGURAZIONE DATASET

Il file di configurazione per il dataset è necessario, in quanto contiene il percorso nel quale si trova il file ".csv" contenente il dataset per l'addestramento. In questo file vengono inoltre inserite le informazioni su come leggere il dataset (specificando quali colonne considerare, quale usare come target, ecc.). I parametri consentiti sono:

- *path*: il percorso del file .csv
- *skip_rows*: numero di righe da ignorare prima dei nomi delle colonne
- *select_columns*: array di nomi delle colonne da selezionare come *Features* (tutti le altre colonne vengono scartate)
- *select_all_columns*: se "true", significa che tutte le colonne sono selezionate (ignorato se select_columns è impostato)
- *skip_columns*: array di nomi delle colonne da ignorare (ignorato se select_columns è impostato)
- *target_column*: nome della colonna di target
- *time_series_column*: nome della colonna per la serie temporale. Viene utilizzato solo per il calcolo delle "time series" ed esclude tutte le altre opzioni colonna
- *sep*: separatore stringa/carattere del file .csv

- *dec*: separatore decimale del file .csv
- *test_size*: intero (numero di campioni per il test) o float (frazione dei campioni totali da utilizzare come test)
- *categorical_multiclass*: booleano che specifica se le etichette multi classe sono "categorial" o no (cioè ordinali)

4.2.2 CONFIGURAZIONE ESTIMATOR

Anche questo file di configurazione, come il precedente, risulta essere necessario, poiché viene specificato quale algoritmo si intende addestrare. Il parametro necessario all'interno di questo file è pertanto *estimator* in cui va specificato uno dei seguenti stimatori:

- *KNeighborsClassifier*
- *KNeighborsRegressor*
- *DecisionTreeClassifier*
- *DecisionTreeRegressor*
- *LinearSVC*
- *LinearSVR*
- *ANNClassifier*
- *ANNRegressor*
- *TripleES*

Ogni stimatore ha poi i propri parametri di configurazione:

- K-NN
  - *n_neighbors* (*): default 5
- DecisionTree
  - *max_depth* (*): default *None*
  - *min_samples_split* (*): default 2
  - *min_samples_leaf* (*): default 1
  - *max_leaf_nodes* (*): default *None*
- LinearSVC/LinearSVR
  - *C_exp* (*): esponente del parametro C, default 0
- ANN
  - *epochs* (*): default 10
  - *batch_size* (*): default 32

- *dropout* (*): float[0,1], default 0
- *activation*: array di stringhe (es. "relu", "tanh", ecc.) per la funzione di attivazione dei livelli nascosti, default "relu"
- *hidden_layers*: array di array di numeri interi che rappresentano la dimensione di ogni strato nascosto, default []. Ciascun array interno rappresenta una configurazione di layer (ovvero il numero di nodi per ogni layer) tra quelli da valutare nella cross-validation.

- TripleES
  - *season_length*

I parametri seguiti da (*) sono parametri che possono essere specificati come valori singoli (sono tutti valori interi se non specificato diversamente), o nelle seguenti forme:

- <*prop*>: valore singolo
- <*prop*>*_array*: un array di valori
- <*prop*>*_upperlimit*: limite superiore per un range di valori
- <*prop*>*_lowerlimit*: limite inferiore per un range di valori
- <*prop*>*_step*: step per un range di valori

4.2.3 CONFIGURAZIONE PREPROCESSING

Questo file e i seguenti non sono più indispensabili per l'avvio del software, tuttavia offrono una serie di funzionalità molto importanti nella costruzione del modello.

In questo file è possibile specificare i parametri che servono per la preparazione dei dati prima dell'addestramento. In particolare, è possibile usare un "scaler" (scalatore) che consente di poter confrontare meglio le colonne tra di loro, ed utilizzare una tecnica, la *Principal Component Analysis PCA*, per la riduzione della dimensione dei dati:

- *scale*: un array di stringhe che specificano gli scaler (attualmente supportati: "StandardScaler", "MinMaxScaler"), default no scaler
- *pca_values*: array di numeri (interi o float[0,1]) che rappresentano le componenti principali. E' possibile specificare anche la stringa "mle" come opzione, default no PCA

(Questo file non viene considerato nel caso di stimatore TripleES)

4.2.4 CONFIGURAZIONE MODEL SELECTION

Qui vengono definiti i parametri che verranno utilizzati per la scelta del modello migliore:

- *cv*: suddivisione per la cross-validation, default *None* per usare la suddivisione più diffusa 5-fold
- *scoring*: metodo di valutazione per la cv

    - Possibili valori per il problema di regressione:
        - "mean_absolute_error"
        - "mean_squared_error", default per ANN
        - "mean_squared_log_error"
        - "root_mean_squared_error"
        - "r2", default per K-NN, LinearSVR e DecisionTree
    - Possibili valori per il problema di classificazione:
        - "accuracy", default for all
        - "balanced_accuracy"
        - "f1", "f1_micro", "f1_macro", "f1_samples", "f1_weighted"
        - "precision", "precision_micro", "precision_macro", "precision_samples", "precision_weighted"
        - "recall", "recall_micro", "recall_macro", "recall_samples", "recall_weighted"

- *verbose*: livello di verbosità, default 0

4.2.5 CONFIGURAZIONE OUTPUT

Questo file è necessario quando si vuole creare il modello per l'esportazione su microcontrollore. I parametri configurabili sono:

- *export_path*: percorso di una directory in cui esportare le cartelle di output, default *None*
- *is_dataset_test*: se devono essere preparati i file per un dispositivo, default *No dataset test* (cioè solo una stima)
- *dataset_test_size*: imposta un limite al numero di etichette di test da esportare. Può essere un intero (numero di etichette) o float[0,1] (frazione), default 1
- *training_set_cap*: per k-NN, imposta un limite al numero di campioni da esportare per la stima k-NN. Può essere intero (numero di campioni) o float[0,1] (frazione), default *None*

4.2.6 CONFIGURAZIONE PREDICTION

Configurando questo file, e quindi, specificando questa opzione all'avvio, tutte le precedenti vengono ignorate in quanto si richiede al software di effettuare una predizione di uno o più campioni utilizzando un modello precedentemente addestrato.

Le proprietà da indicare in questo file sono:

- *model_id*: l'uuid del modello che si vuole utilizzare
- *samples*: array di campioni per i quali si vuole fare una predizione

4.2.7 CONFIGURAZIONE STORAGE

Questa opzione di avvio, a differenza delle altre, non richiede un file di configurazione, ma è sufficiente specificarla nella *command line*, all'avvio del programma.

Se scelta, permette di salvare la migliore configurazione trovata per il modello, in modo da poterlo usare in modalità di predizione sul cloud.

4.3 BREVE CENNO A MICRO-LM

Come accennato in precedenza, Micro-LM è una semplice libreria in linguaggio C per il Machine Learning su dispositivi "Edge".

Attualmente implementa solo algoritmi supervisionati, quali:

- *Linera SVM*
- *Decision Tree*
- *K-NN*
- *TripleES*

I file generati dall'addestramento in Desk-LM, devono essere compilati insieme ai file disponibili in questo repository.

5 PROGETTAZIONE DEL SISTEMA

Questo capitolo presenta il processo di progettazione del sistema, a partire dai requisiti.

5.1 REQUISITI

Il primo passo per la realizzazione di una *Computation* per *Measurify*, che utilizzi gli algoritmi di Machine Learning implementati da Desk-LM, è quello di rendere facile l'interazione tra le due piattaforme.

Per questo motivo, abbiamo scelto di realizzare un Application Programming Interface (API) che faccia da framework per ELM, implementando l'architettura RESTful. Affinché un'API sia considerata RESTful, deve rispettare i criteri indicati di seguito:

- **Architettura client-server**: le interazioni sono basate sul modello request-response, dove un client invia una richiesta (request) a un server e ottiene una risposta (response). Questo massimizza il disaccoppiamento tra i componenti: si parla quindi di *Loose Coupling* (accoppiamento allentato), dove ogni componente può esistere ed evolvere indipendentemente.
- **Interfacce uniformi**: L'accoppiamento allentato può essere ottenuto solo quando si utilizza un'interfaccia uniforme che tutti i componenti del sistema rispettano.
- **Stateless**: Il contesto e lo stato del client non devono essere mantenuti sul server, ma solo sul client; ogni richiesta deve contenere lo stato del client.
- **Cacheable**: La *cache* è un elemento chiave per le prestazioni. Server, client ed intermediari possono memorizzare alcuni dati localmente per non dover recuperare i dati dal server effettivo per ogni richiesta. I server possono definire i criteri di quando i dati scadano e quando gli aggiornamenti devono essere ricaricati dal server.
- **Sistema a strati**: Diversi componenti intermedi possono essere frapposti tra server e client per migliorare scalabilità e tempi di risposta.

L'API RESTful separa l'interfaccia utente dal server e dall'archiviazione dei dati, il che migliora la portabilità, la scalabilità e lo sviluppo indipendente.

L'archiviazione verrà affidata ad un sistema di gestione di database basato su documenti (DBMS).

Per quanto riguarda le risorse accessibili tramite API, abbiamo identificato la risorsa *model* come unica risorsa, in quando il software ELM ha l'unico compito di costruire ed addestrare modelli di Machine Learning: per tale risorsa, il framework dovrà gestire tutti i metodi HTTP menzionati in precedenza.

La risposta di ogni richiesta sarà composta da un codice, che può essere di successo (2xx), o di errore (4xx). Nel caso di codice di successo, la risposta sarà accompagnata da informazioni sulla risorsa, mentre nel caso di errore, ci saranno messaggi indicativi il tipo di errore.

5.2 ARCHITETTURA

L'architettura del sistema prevede una prima parte in cui, a partire dal framework ELM, andremo a costruire un'interfaccia API che esponga delle rotte al client (Fig. 3). La costruzione del server prevede che il software ELM non venga modificato, in modo tale che possa essere ampliato in futuro senza che l'API debba essere modificata.

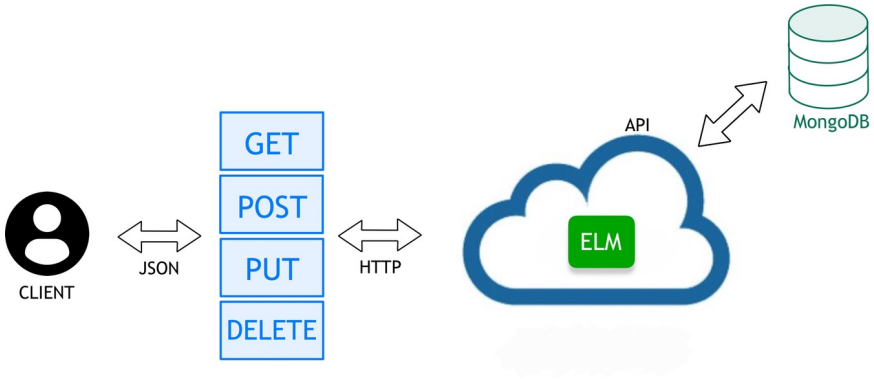


Figura 3: Realizzazione API RESTful per ELM

Una volta realizzata l'interfaccia (API) per ELM, dobbiamo consentire ai due server di poter comunicare in modo tale da poter far sfruttare le funzionalità ad un client di Measurify. Per fare questo, dovremo integrare in Measurify tutte le funzionalità di ELM, e offrire al client la possibilità di creare una *Computation* (Fig. 4). Il client non avrà un accesso diretto ad ELM, bensì specificherà i parametri direttamente nelle richieste a Measurify.

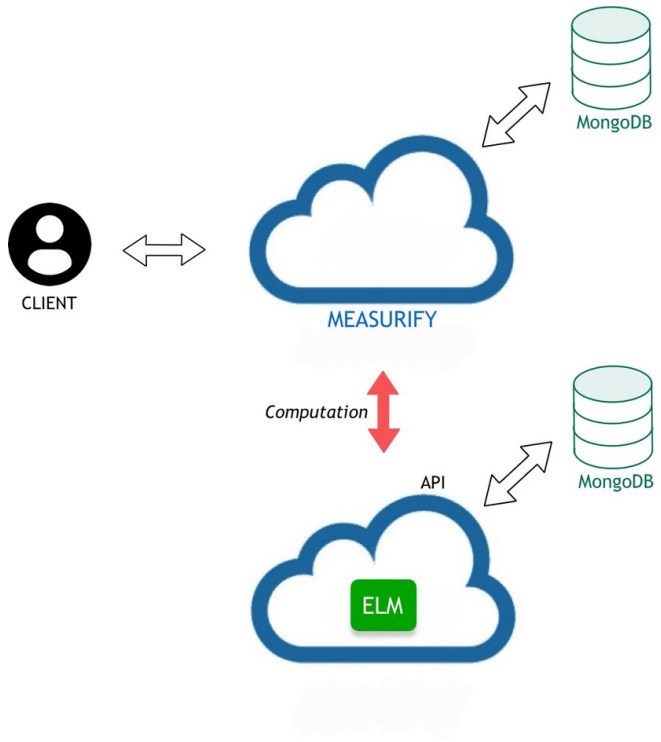


Figura 4: Integrazione di ELM in Measurify

In Fig. 5 vediamo rappresentale le varie fasi per la creazione di un modello. Abbiamo una prima fase, dove andremo a fare una POST sulla risorsa *model* in cui andremo a specificare, dentro al *body* della richiesta, tutti i parametri per la configurazione di ELM. I parametri sono quelli necessari per l'avvio di ELM e che sono descritti nel paragrafo 4.2. Inoltre, per l'integrazione con altri server (in particolare con Measurify), all'interno della richiesta, aggiungiamo un campo "webhook", dove il client può specificare una URL che le API di ELM potranno usare per informare il client sullo stato del *model* richiesto.

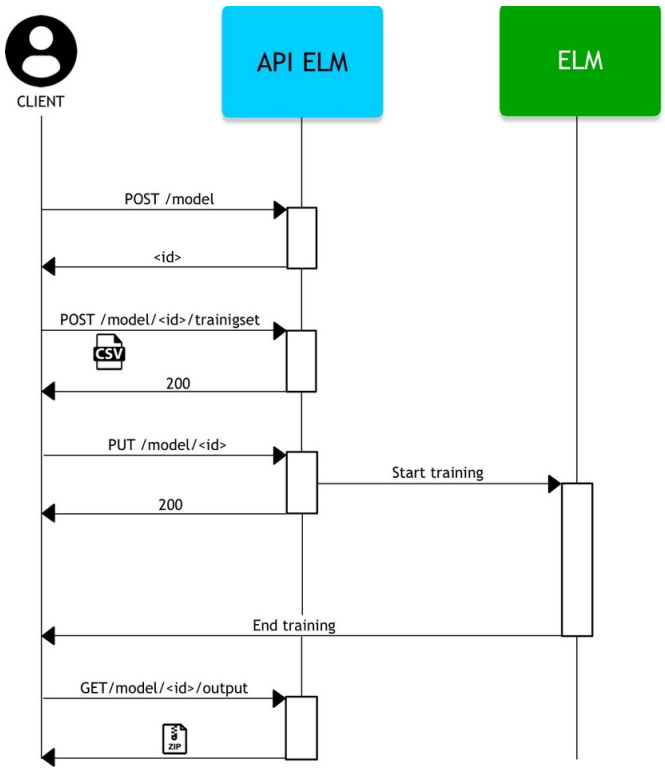


Figura 5: Diagramma UML temporale per la creazione di un modello

La richiesta POST dovrà rispondere con l'ID di una nuova risorsa di tipo *model* a cui si potrà fare riferimento successivamente. Questa risorsa avrà tutti i campi passati con il body più i seguenti:

- **_id**: identificativo univoco della risorsa *model* creata, che deve essere un GUID
- **status**: che indica a che punto si trova la costruzione del *model*
- **output**: il nome del file ZIP che possa essere scaricato
- **timestamp**: con la data di creazione della risorsa

Dopo la creazione del modello, è necessaria una rotta che permetta il caricamento di un file .csv che costituisce il training set su cui il modello verrà costruito. La rotta, indicata in Fig. 5, aggiornerà la risorsa *model* identificata dall'id nella URL. Il file CSV sarà caricato in un folder apposito nel server.

A questo punto, è possibile dare il comando poiché l'elaborazione del modello inizi: in Fig. 5 è rappresentata dalla PUT specificando nella URL l'id del modello. In questa richiesta andremo

anche ad inserire un *body*, in cui specificheremo la modalità in cui vogliamo avviare ELM (addestramento o predizione). Utilizzeremo pertanto il campo **mode** in modalità "*evaluate*", per specificare la modalità addestramento.

Dovremmo, poi, predisporre il server per gestire anche due richieste di tipo GET: la prima, che sarà nella forma

```
GET /model/<id>
```

dovrà restituire il JSON della risorsa *model* dell'id specificato. La seconda:

```
GET /model/<id>/output
```

che, una volta completato l'addestramento, restituirà il file ZIP creato da ELM.

Infine, creiamo anche una rotta DELETE per l'eliminazione di uno specifico modello, identificato, anche in questo caso, dall'id passato nella URL.

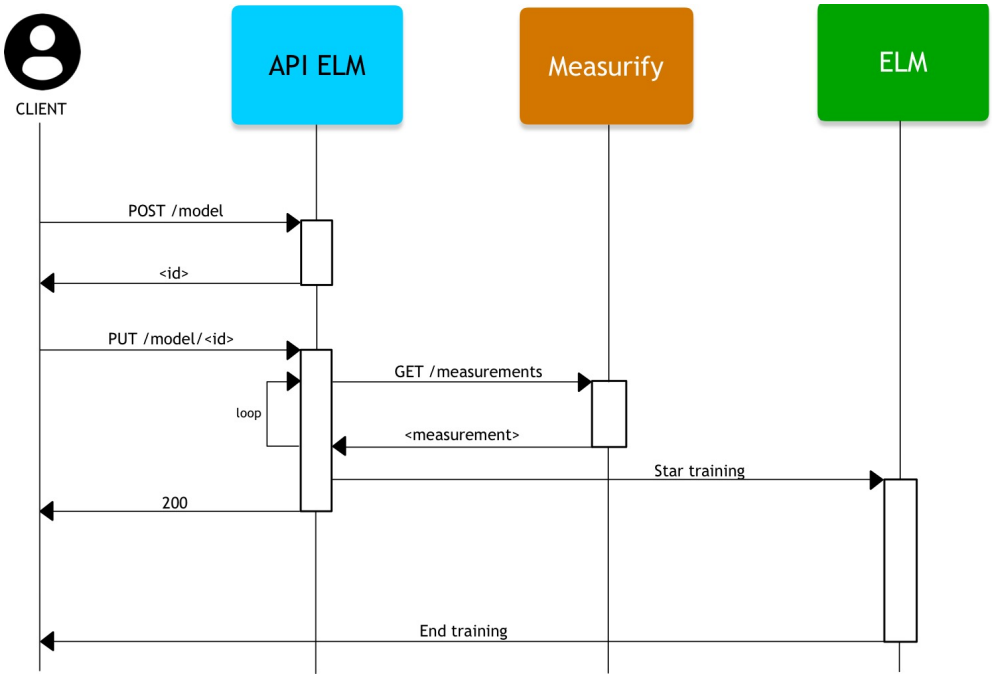


Figura 6: Diagramma UML temporale per la creazione di un modello utilizzando come dataset Measurify

Il passo successivo è quello di configurare l'API in modo che possa, invece che ricevere un file di dataset, costruirselo utilizzando il server Measurify.

Come rappresentato in Fig. 6, dovremmo configurare il server in modo tale che, dopo la POST inziale di creazione del modello, venga eseguita subito la PUT per l'addestramento, senza che gli venga passato nessun dataset. Il *body* di questa richiesta dovrà quindi contenere, oltre ad una modalità diversa, che sarà nominata "*measurify*", anche tutti i campi necessari per la richiesta di misurazioni su *Measurify*.

Una volta costruita l'API per ELM, dovremo passare all'integrazione con *Measurify*: iniziamo trattando più nel dettaglio i concetti di *Feature* e *Measurement* in *Measurify*.

In Measurify sono memorizzate le *Measurement* che sono "valori" misurati sul "campo". Il significato di questa frase è molto ampio e alcuni esempi di *Measurement* sono: il valore di temperatura misurato da un termometro in una stanza, il valore di peso di una persona misurato da una bilancia, la posizione di un'automobile misurata da un GPS, ecc.

Il valore contenuto in una *Measurement* può essere scalare o vettoriale e il suo significato viene interpretato grazie alla "feature" cui la misura fa riferimento.

La *Feature* descrive come sono fatti i valori all'interno delle *Measurement*. Ad esempio, la "feature" "temperatura" può essere definita nel modo seguente:

```
{
    "_id": "temperature",
    "items": [
        "dimension": 0,
        "type": "number",
        "name": "value",
        "unit": "celsius degree"
    ]
}
```

da cui si vede che le *Measurement* di tipo "temperatura" avranno un solo valore (infatti la dimensione del vettore *items* è pari a 1) e tale valore sarà di tipo numerico (*type*: number), scalare (*dimensios*: 0) con unità di misura (*unit*) gradi celsius.

Un esempio non scalare, è la *feature* "position":

```
{
    "_id": "position",
    "items": [
        {
            "name": "longitude",
            "unit": "degree",
            "type": "number",
            "dimension": 0
        },
        {
            "name": "latitude",
            "unit": "degree",
            "type": "number",
            "dimension": 0
        }
    ]
}
```

come si vede, in questo caso la *feature* è vettoriale, dato che l'array "items" ha due valori, entrambi scalari e numerici, che rappresentano la latitudine e la longitudine.
Si faccia attenzione a due caratteristiche:

- le *measurement* possono avere più di una singola "misura", infatti posseggono il campo "samples" che è un vettore. Questo permette alla singola *measurement* di temperatura

o posizione di memorizzare più di un valore di "temperatura" o di "posizione". Questo può essere utile per *measurement* che rappresentano delle time-series. Ad esempio, le temperature di una stanza in una giornata oppure il percorso di un'automobile. In questo caso, ogni "value" è accompagnato anche da un istante temporale relativo che determina il momento temporale di tale "value" rispetto alla "startDate" della *measurement*;

- le *measurement* hanno anche un campo "tags" che può essere usato per aggiungere delle etichette che descrivono quella misura.

Supponiamo ora, di trovarci nel contesto in cui l'utente di Measurify sia un dispositivo "Edge", e che faccia delle misure di qualche tipo tramite i suoi sensori, e supponiamo inoltre di avere una feature con N dimensioni (items) che comprende tutti i valori misurati dai sensori (Fig. 7). Infine, immaginiamo che queste misure vengano taggate con valori diversi utilizzando il campo *tags*.

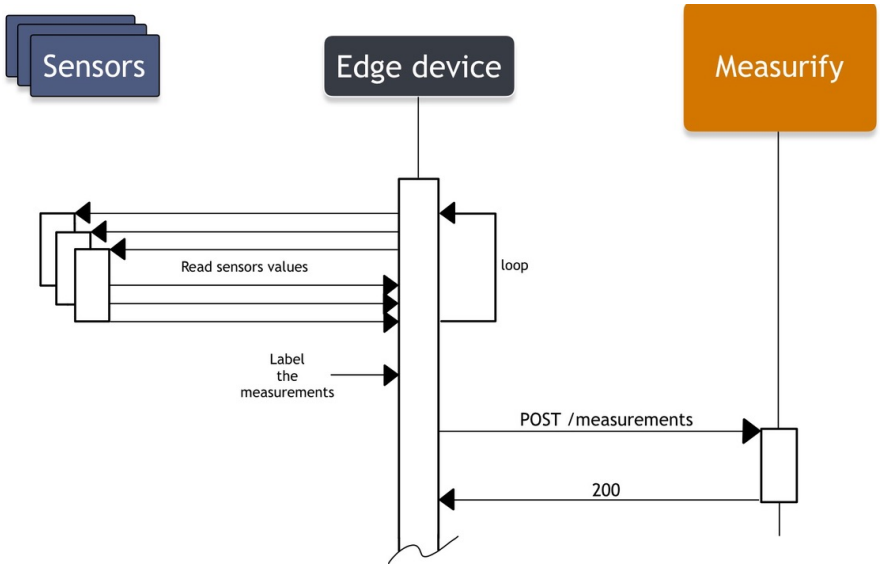


Figura 7: Dispositivo "Edge" che effettua misurazioni dai suoi sensori e le memorizza sul server Measurify

Potremmo permettere all'utente di Measurify (dispositivo "Edge") di richiedere la costruzione di un modello che sappia predire il *tag* di nuove misurazioni, sulla base degli esempi delle misure che invece sono già taggate. Per questo useremo la rotta "computation" già esistente in Measurify che dovrà essere modificata.

A questo punto, Measurify sfrutterà l'API di ELM che abbiamo descritto precedentemente: prima creerà il modello specificando i parametri, successivamente invierà a ELM tutte le informazioni necessarie affiché possa costruirsi il dataset ed addestrare il modello con tali dati (Fig. 8).

A fine addestramento, Measurify, si occuperà di memorizzare il risultato (il modello) all'interno di una risorsa e renderla disponibile al client.

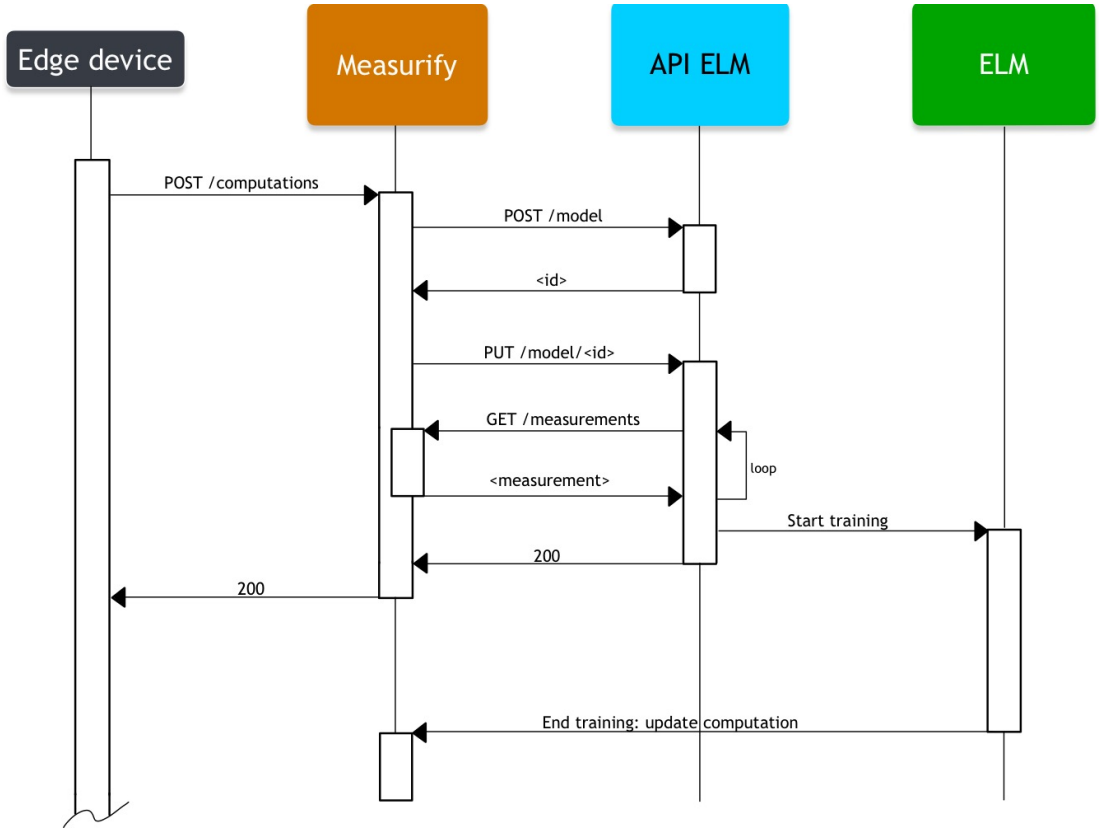


Figura 8: Creazione di una Computation da parte di un dispositivo "Edge"

6 IMPLEMENTAZIONE

6.1 APPLICATION PROGRAMMING INTERFACE PER ELM

Abbiamo implementato i servizi API RESTful in Python, in quanto ELM è stato sviluppato interamente in questo linguaggio, ed abbiamo utilizzato il micro-framework Flask, che è basato sullo strumento Werkzueg WSGI e con il motore modello Jinja2. Flask è un micro-framework perché ha un nucleo semplice ma estendibile. Non c'è uno strato di astrazione per i database, validazione dei formulari, o qualsiasi altra componente, in quanto esistono già altre librerie di terze parti che forniscono tali funzionalità.

L'utilizzo di Flask è molto semplice: per prima cosa si inizializza un nuovo oggetto

```
0 from flask import Flask
1 app = Flask(__name__)
```

successivamente si definiscono le varie rotte (con percorso relativo) specificando i metodi supportati per ciascuna:

```
0 @app.route("/", methods=["GET","POST"])
1 def main():
2     return "Response"
```

In questo modo abbiamo definito la rotta http://localhost:5000/ che risponde la stringa "Response" ad ogni richiesta GET e POST.

Per l'archiviazione, abbiamo scelto di affidarci al DBMS non razionale *MongoDB*, che come detto in precedenza, offre moltepilci vantaggi, tra cui la scalabilità e la flessibilità: data la natura senza schema di MongoDB, abbiamo bisogno di un controllo dei dati, perciò abbiamo definito uno schema per tutte le richieste.

Essendo un API che potenzialmente potremmo esporre su internet, è necessaria anche la protezione da usi malevoli. Per questo sono necessarie due cose:

- utilizzare il protocollo *HTTPS*;
- inserire in ogni richiesta un token all'interno dell'header che identifichi il client di ELM.

Per supportare HTTPS invece che HTTP è necessario modificare Flask in modo che usi i certificati: abbiamo usato *Let's Encrypt* [12] per ottenere dei certificati validi.

Per quanto riguarda la protezione tramite token, invece, abbiamo implementato una collezione su MongoDB chiamata "*clients*", in cui andranno dei documenti con i campi:

- "**_id**", "**name**", "**token**", "**IP_list**"

"IP_list" verrà usata per una maggiore protezione, pretendendo che un token possa essere usato solo da un certo insieme predefinito di numeri "IP". Se tale lista è vuota, allora le richieste con quei token possono arrivare da ogni indirizzo IP.

Ogni richiesta che arriva alle API dovrà essere accompagnata dal campo *Authorization* nell'header, valorizzato da un token. L'API controlla l'esistenza del campo e se il token presente è valido andandolo a cercare dentro la collezione su MongoDB. Se così non fosse, risponde con un codice di errore 401 (Unauthorzied).

Per semplicità, abbiamo inserito un'informazione dentro un file di configurazione delle API, chiamata "BASE_TOKEN". Allo startup, l'API, se trova la collezione dei token vuota, leggerà il file di configurazione e creerà un "clients" di default, in modo tale da avere sempre la possibilità di funzionare non appena lanciate, senza operazioni manuali di configurazione.

6.1.1 ROUTES

Di seguito vediamo l'implementazione delle rotte per l'API

Creazione del modello

La rotta di creazione del modello è la seguente:

```
POST URL/model
body:
{
    # ...
}
```

Abbiamo definito lo schema del "body" di tipo "object", nel modo seguente:

```
{
    "type": "object",
    "properties": {
        "est": { "type": "object" },
        "pp": { "type": "object" },
        "ms": { "type": "object" },
        "output": { "type": "object" },
        "webhook": { "type": "object" }
    },
    "required": ["est"],
    "additionalProperties": false
}
```

dove sono indicati tutti i campi, il loro tipo, e quelli necessari. Possiamo vedere come il solo il campo "est" sia necessario, in quanto conterrà le informazioni di configurazione dello stimatore 4.2.2. I campi "pp", "ms" e "output", in modo analogo, possono essere configurati rispettivamente come i file di configurazione di ELM di *pre-processing* (4.2.3), *model selection* (4.2.4) e *output* (4.2.5). Infine, nel campo "webhook", è possibile specificare un indirizzo URL e un metodo (GET, POST o PUT), che il server di ELM andrà a "chiamare" quando ha finito l'addestramento.

I parametri delle proprietà del body vengono definiti e convalidati mediante altri schemi interni a ELM. Questo mantiene una separazione tra API e ELM, in modo tale da poter apportare modifiche interne ad ELM senza che l'API debba essere modificata. L'unico schema che abbiamo dovuto definire è quello che riguarda il parametro "webhook" essendo una funzionalità dell'API e non di ELM:

```
{
    "type": "object",
    "properties": {
        "url": { "type": "string" },
        "method": { "type": "string" },
        "headers": { "type": "object" },
        "data": { "type": "object" }
    },
    "required": ["url","method"],
    "additionalProperties": false
}
```

Questa configurazione permette al client che userà l'API di specificare, oltre all'indirizzo URL ed al metodo (che sono necessari), anche dei possibili "headers" (ad es. se il client è anch'esso un server e necessita di un token di autorizzazione da inserire negli "headers", può essere il client stesso a fornire il token al server) e un "body" (ad es. se il client vuole come "webhook" un particolare messaggio).

Caricamento del Dataset

```
POST URL/model/<id>/trainingset
```

In questa rotta il contenuto deve essere passato come "form-data" in quanto necessita del file csv allegato. Con questa modalità (oltre al file allegato) è possibile specificare altre coppie "key-value" che possono essere usate per specificare i parametri di configurazione del dataset (4.2.1).

Per questa rotta non abbiamo definito uno schema di convalida, in quando verrà semplicemente controllata la presenza del file allegato.
Addestramento

Questa rotta è quella che serve per l'addestramento e la predizione:

```
PUT URL/model/<id>
{
    "mode": "<tipo>",
    # ...
}
```

Il suo schema:

```
{
    "type": "object",
    "properties": {
        "mode": {
```

```
            "enum": ["evaluate", "predict", "measurify"]
        },
        "samples": {
            "type": "array",
            "items": {
                "oneOf": [
                    { "type": "array" },
                    { "type": "number" }
                ]
            }
        },
        "n_preds": { "type": "number" },
        "feature": { "type": "string" },
        "items": {
            "type": "array",
            "items": { "type": "string" }
        },
        "filter": { "type": "string" },
        "tags": {
            "type": "array",
            "items": {
                "type": "string"
            }
        }
    },
```

dove possiamo notare che la proprietà *mode* consente tre opzioni:

- **evaluate**: questa modalità necessita che, oltre al modello identificato dalla "id" presente nella URL, sia già stato caricato il dataset nel formato CSV. Se il server trova il file, e la configurazione del modello è corretta, allora avvia un thread in parallelo in cui ELM eseguirà l'addestramento. A fine addestramento, se il webhook è stato configurato, il server si occuperà di effettuare la richiesta al'URL che gli è stato indicato;
- **predict**: in questa seconda modalità, può essere richiesta una predizione su uno o più campioni ("samples"), utilizzando un modello, identificato da "id" nella URL, precedentemente addestrato;
- **measurify**: la terza e ultima modalità, è quella che ci permette di addestrare un modello senza un file di dataset, appoggiandosi al server Measurify per la costruzione di esso.

I campi necessari per ciascun tipo di richiesta sono:

```
    "allOf": [
        {
            "if": {
                "properties": { "mode" : { "const": "evaluate"
    } }
            },
            "then": {
```

```
                "required": ["mode"]
            }
        },
        {
            "if": {
                "properties": { "mode" : { "const": "predict"
} }
            },
            "then": {
                "required": ["mode", "samples"]
            }
        },
        {
            "if": {
                "properties": { "mode" : { "const": "measurify
" } }
            },
            "then": {
                "required": ["mode", "feature", "items", "
filter", "tags"]
            }
        }
    ],
    "additionalProperties": false
}
```

dove si può notare che in modalità "evaluate" abbiamo solo il campo "mode", in modalità "predict", è necessario anche il campo "samples", mentre nella terza modalità, "measurify", sono necessari i campi "feature", "items", "filter" e "tags".

Il campo "filter" serve per filtrare tra tutte le *Measurements*, quelle di una determinata *Feature*. La stessa *Feature* deve essere indicata, in quanto il dataset deve essere formato da campioni dello stesso tipo. É possibile inoltre specificare quali "items" della *Feature* dobbiamo estrarre. Il campo "tags" serve per specificare quali sono le classi su cui andremo a fare addestramento. Durante la richiesta di *Measurements* verranno filtrate in modo automatico tutte le *Measure* che non possiedono uno dei tag indicati.

Richieste GET e DELETE

Le richieste di tipo GET e DELETE, non avendo un corpo, non hanno uno schema di convalida. Pertanto, per tali richieste, l'unico controllo è quello sul token.

La richiesta

```
GET URL/model
```

ritorna un array con tutti i modelli creati.

La richiesta

```
GET URL/model/<id>
```

ritorna solo il modello identificato dall'id nella URL.

La richiesta

```
GET URL/model/<id>/output
```

ritorna il file ZIP contenente tutti i file .h e .c per l'utilizzo su MicroControllore.

Infine, la richiesta

```
DELETE URL/model/<id>
```

elimina il modello identificato dall'id (se esistente).

6.2 COMPUTATION IN MEASURIFY

La seconda parte del progetto riguarda l'estensione del framework *Measurify*. Il codice è scritto in NodeJS usando Express e Mongoose.

Una *Computation* in Measurify è una risorsa che rappresenta un'operazione che il client chiede all'API di realizzare su un insieme di misure.

Dal punto di vista dell'utilizzo, un client può costruire una nuova *Computation* nel modo seguente:

```
POST {{url}}/v1/computations
{
    "_id": "<NuovoID>",
    "code": "<code>",
    "feature": "feature",
    "items": ["<items della feature>"],
    "filter": "{\"feature\":\"feature\"}"
}
```

dove:

- "**_id**": è un identificativo univoco che individua la *Computation* che si vuole creare;
- "**code**": specifica l'operazione che si vuole eseguire con questa particolare *Computation*;
- "**feature**": identifica la *Feature* su cui si vuole eseguire la *Computation*, questo è essenziale per poter fare dei controlli su quello che si sta facendo, dato che le operazioni dovranno essere consistenti;
- **filter**: aiuta a specificare l'insieme delle misure su cui verrà effettuata la computazione richiesta. Si possono costruire filtri di qualunque tipo, seguendo le regole di MongoDB. In realtà il filtro sulla *Feature* non sarebbe necessario (dato che è stata indicata in precedenza), ma al momento l'implementazione lo richiede.

Una volta fatta la POST della computation, il calcolo parte, ma il client riceve immediatamente una risposta, senza attendere che la *Computation* sia terminata. La risposta che si ottiene è la seguente:

```
{
    "results": [],
    "items": ["<items della feature>"],
    "status": "running",
    "progress": 0,
    "visibility": "private",
    "tags": [],
    "_id": "<NuovoID>",
    "code": "<code>",
    "feature": "feature",
    "filter": "{\"feature\":\"feature\"}",
    "owner": "5f2129d46bf5331fc0fb2493",
    "startDate": "2020-07-30T13:23:20.721Z",
    "timestamp": "2020-07-30T13:23:20.721Z",
    "lastmod": "2020-07-30T13:23:20.721Z"
}
```

dove si vede che la computazione è in esecuzione ("status": "running") e che ha completato lo 0 percento ("progress":0), dato che è stata appena creata.

Il client può avere informazioni sullo stato della computation, tramite una GET che fornisce un update di "status" e "progress" e quando l'operazione è finita anche il risultato:

```
GET {{url}}/v1/computations/{{id_computation}}
{
    "results": [
        { ... },
        { ... }
    ],
    "items": ["<items della feature>"],
    "status": "concluded",
    "progress": 100,
    "tags": [],
    "_id": "<NuovoID>",
    "code": "<code>",
    "feature": "feature",
    "owner": "5f2129d46bf5331fc0fb2493"
    "filter": "{\"feature\":\"feature\"}",
    "startDate": "2020-07-30T13:23:20.721Z"
}
```

Il campo "results" è peculiare di ogni diverso tipo di *Computation*.

Il codice che gestisce le *Computations* si trova tutto all'interno della cartella "computations": abbiamo fatto uso del file "runner.js" che espone i metodi:

- **go**: che effettua un controllo sul *code* e avvia la computation corrispondente (chiamando il metodo *run*, comune di ogni computation);
- **progress**: che aggiorna il campo "progress" della *Computation* con un valore numerico passatogli come valore;

- **complete**: che aggiorna il campo "results" della *Computation*;
- **error**: chiamato quando si verifica un errore, ne interrompe l'esecuzione e mette lo stato ad "error".

Per come abbiamo configurato il server di ELM, dobbiamo estendere il "runner" con un ulteriore metodo, **update** che ci servirà per chiamare il rispettivo *update* della *Computation*.

Veniamo ora all'implementazione della *Computaion*.

Per prima cosa abbiamo realizzato un modulo, chiamato "*request.js*", che ci servirà per effettuare richieste al server di ELM. Tale modulo espone un solo metodo, *sendJson*:

```
exports.sendJson = async function(url, method, headers, json=true) {
    return new Promise((resolve, reject) => {
        request({
            url: url,
            method: method,
            headers: headers,
            json: json,
            rejectUnauthorized: false
        }, (error, response, body) => {
            if(error)
                reject(error);
            else
                resolve({response, body});
        });
    });
}
```

Tale metodo, ritorna una *Promise*, che rappresenta un proxy per un valore non necessariamente noto quando la promise è stata creata. Consente di associare degli handlers con il successo o il fallimento di un'azione asincrona (e il "valore" in caso di successo, o la motivazione in caso di fallimento). Questo in pratica consente di utilizzare dei metodi asincroni di fatto come se fossero sincroni: la funzione che compie del lavoro asincrono non ritorna il valore di completamento ma ritorna una promise, tramite la quale si potrà ottenere il valore di completamento una volta che la promise sarà terminata.

Una Promise può presentarsi in uno dei seguenti stati:

- *pending* (attesa): stato iniziale, né soddisfatto né respinto.
- *resolved* (soddisfatto): significa che l'operazione si è conclusa con successo.
- *rejected* (respinto): significa che l'operazione è fallita.

Una promise in "pending" può evolvere sia in "resolved" con un valore, sia in "rejected" con una motivazione (errore). Quando accade una di queste situazioni, vengono chiamati gli handler associati che sono stati accodati dal metodo "then" della "promise". (Se la "promise" è già stata soddisfatta o respinta quando viene agganciato l'handler, quest'ultimo verrà chiamato immediatamente, quindi non è necessario che gli handler vengano agganciati prima del completamento dell'operazione asincrona).

Questo modulo ci permetterà di effettuare richieste ad altri server, in particolare al server di ELM, specificando l'URL, il metodo (GET, POST, PUT o DELETE), gli "headers" (in cui dovremo ricordarci di inserire il token per l'autorizzazione) e il corpo (in formato JSON).

Poiché questo modulo astrae la richiesta e può essere usato indistintamente per qualsiasi server, abbiamo realizzato un modulo specifico per utilizzare l'API ELM. Tale modulo, chiamato "*elm.js*", espone un metodo per ogni rotta definita dall'API ELM tra quelle di cui abbiamo bisogno:

```
exports.postModel = async function(data) { /*...*/ }
exports.putMeasurify = async function(id, data) { /*...*/ }
exports.getModel = async function(id) { /*...*/ }
exports.getOutput = async function(id) { /*...*/ }
```

Questi metodi utilizzano tutti quanti il modulo precedentemente descritto, ognuno personalizzando i parametri a seconda della richiesta.

Infine, abbiamo realizzato il modulo vero e proprio della *Computation*, il modulo "*model.js*". Questo modulo espone due metodi:

```
exports.run = async function(computation, user, tenant) { /*...*/ }
exports.update = async function(computation, user, tenant) { /*...*/ }
```

Il primo metodo, "run" è il metodo che viene chiamato all'avvio della *Computation*.
All'interno della funzione arriva l'oggetto *Computation* che contiene i campi di cui avremo bisogno, in particolare il campo *metadata*, che è una lista di copie chiave-valore dove vengono memorizzati i dati che servono per determinare i parametri per la costruzione del modello. In questo campo devono essere specificati i parametri di configurazione per ELM (4.2). Gli altri campi che arrivano all'interno dell'oggetto *Computation* sono:

```
POST {{url}}/v1/computations
body:
{
    "_id": "Create a model",
    "code": "model",
    "metadata": {},
    "feature": "{{feature}}",
    "items": [],
    "filter": "{\"feature\":\"{{feature}}\"}",
    "tags": []
}
```

dove il "code" di valore *model* è quello che identifica il tipo di *Computation* che abbiamo creato, mentre gli altri gli abbiamo già descritti ad inizio paragrafo.

Come da schema temporale di Fig. 8, il metodo "run" effettua una prima richiesta all'API di ELM per costruire il modello, usando come "body" di richiesta il campo "metadata". Alla risposta, effettuerà una PUT in modalità "measurify", passandogli tutti i campi che sono necessari al server di ELM affinché possa costruirsi il dataset interrogando Measurify. L'API ELM, una volta costruito il dataset, avvia l'addestramento in modo parallelo: una volta completato, manda un "webhook" a Measurify, che a questo punto avvierà l'altro metodo del modulo "model.js",

il metodo "update". Questo metodo (che può essere chiamato anche durante l'addestramento) aggiorna il "progress" della *Computation* (utilizzando il metodo "progress" di "runner.js"), con il valore passatogli di ELM. Quando raggiunge il 100%, farà una richiesta di tipo GET sull'id del modello per recuperare il "result" dell'addestramento, e completare la *Computation* utilizzando il metodo "complete" di "runner.js".

Il modulo, inoltre, chiamerà il metodo "error" di "runner.js" in caso di errori, che possono essere:

- servizio ELM non disponibile;
- errori di "metadata" mancanti o errati;
- errori provenienti da ELM.

7 TESTING

L'obiettivo del test è quello di andare a testare il funzionamento del workflow del sistema, partendo da dei dati raccolti in uno scenario di IoT, fino ad arrivare ad ottenere un modello. Per questo motivo, non è fondamentale la raccolta di dati reali, e neanche l'accuratezza del modello creato, il test non è centrato sui modelli di Machine Learning.

7.1 SCENARIO DI TEST

Lo scenario scelto per un primo semplice test, è quello della *Smart Home* in cui sono presenti alcuni sensori di temperatura e umidità. In particolare, abbiamo la presenza di due sensori di temperatura (uno interno all'abitazione ed uno esterno), ed un sensore di umidità (interno) (Fig. 9).

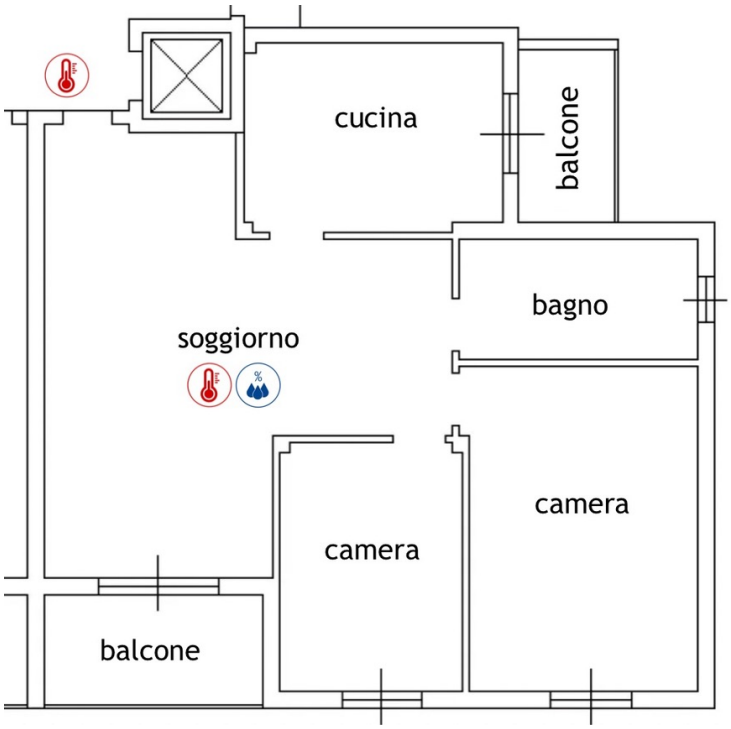


Figura 9: Esempio di scenario

La *Feature* che andiamo ad identificare, è quindi composta da tre *items*:

- *Temperatura interna*: "temp_int"
- *Umidità interna*: "hum_int"
- *Temperatura esterna*: "temp_est"

L'applicazione potrebbe essere quella di rilevare o meno, la presenza di persone all'interno dell'abitazione. Per la costruzione del dataset, immaginiamo di aver raccolto una certa quantità

di misurazioni, che siano state tutte etichettate con i tags "*empty*", in caso in cui non ci fosse nessuno dell'abitazione, e "*detection*" in caso di presenza di una o più persone.

7.2 GENERAZIONE DATI

La generazione dei dati viene fatta simulando un dispositivo "edge" avente accesso alle misurazioni effettuate dai sensori. Abbiamo perciò creato un semplice script, che si fingerà "client" di Measurify, in grado di generare misurazioni in modo randomico, ma il più possibile realistiche.

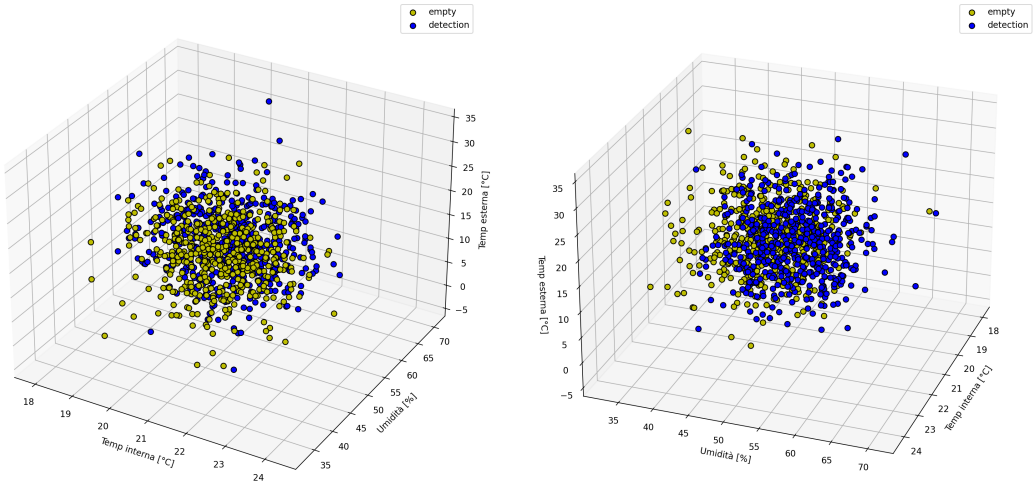
Per prima cosa, il client, effettuerà le inizializzazioni delle *resources* che serviranno per il test. In particolare, dovrà definire la *feature* (come definita del paragrafo precedente), il *device* (chiamato per semplicità "smartHomeDevice"), la *thing* (analogamente al device "smartHomeThing") e i *tags* necessari (anche questi definiti precedentemente).

Una volta inizializzato Measurify, abbiamo utilizzato la funzione *make_blobs* di *sklearn* [13], che consente di creare dei dataset multi classe, assegnando a ciascuna classe uno o più cluster di punti distribuiti normalmente, specificando centri e deviazioni standard di ciascun cluster.

I parametri di configurazione principali sono i seguenti:

- *n_sample*: il numero di campioni che si desiderano;
- *n_feature*: la dimensione dei campioni;
- *centers*: array di centri (array) dei clusters;
- *cluster_std*: array di deviazioni standard (array) per i cluster;

Per il nostro test abbiamo generato due cluster, uno per classe (le due classi corrispondono ai tags "empty" e "detection") con posizione dei centri molto vicine e deviazioni standard uguali per entrambi. Nel dettaglio, abbiamo scelto di far variare le misurazioni di temperatura interna tra 19°C e 23°C, l'umidità tra il 40% e il 60% e la temperatura esterna tra 0°C e 30°C. Nelle Fig. 10, 11 e 12, sono rappresentate diverse viste del dataset generato.



(a) (b)

Figura 10: Vista 3D del dataset

Tutte queste misurazioni vengono poi inviate al server, tramite la rotta corrispondete, per avere un dataset di partenza disponibile per essere addestrato.

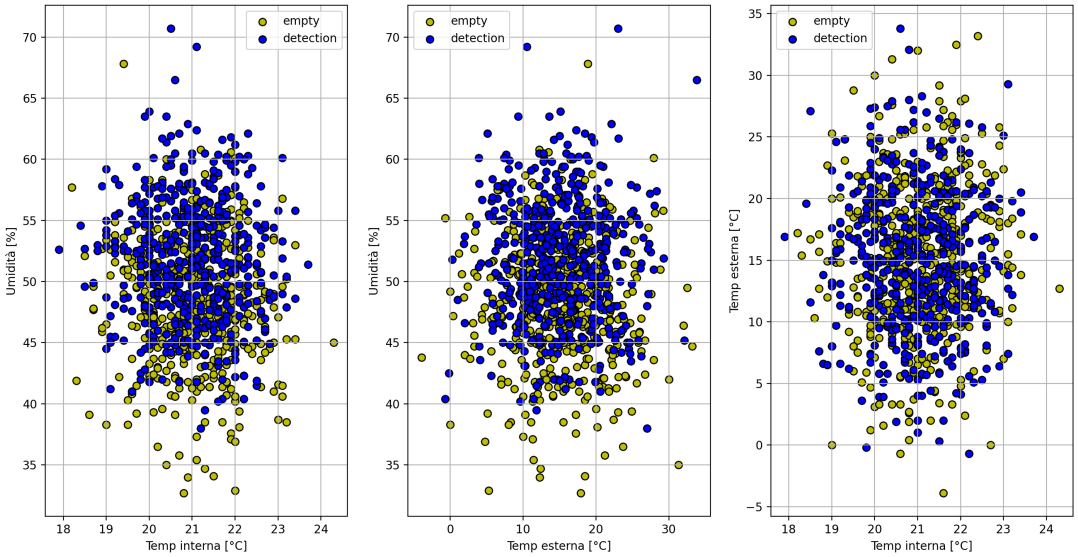


Figura 11: Viste 2D del dataset con in risalto i campioni con etichetta "detection"

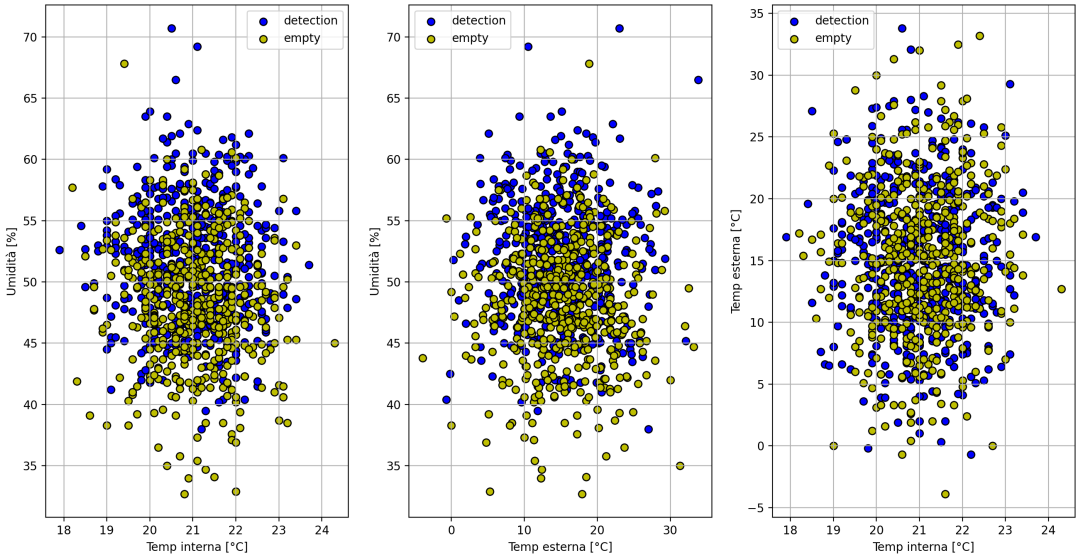


Figura 12: Viste 2D del dataset con in risalto i campioni con etichetta "empty"

Una volta generato il dataset, abbiamo simulato la richiesta, da parte del client, di una *computation* sui dati generati. Per fare questo, abbiamo simulato il client utilizzando il software POSTMAN [14], che è un'applicazione del browser Google Chrome che consente di testare un API in modo veloce e completo. Tramite Postman è possibile effettuare delle chiamate API senza dover mettere mano al codice dell'applicazione, consentendo di effettuare le chiamate tramite questo plugin che fornisce un utile interfaccia grafica. Le richieste possono essere effettuate sia verso un server locale che verso un server online impostando tutti i dati di una tipica chiamata API, dagli headers al body.

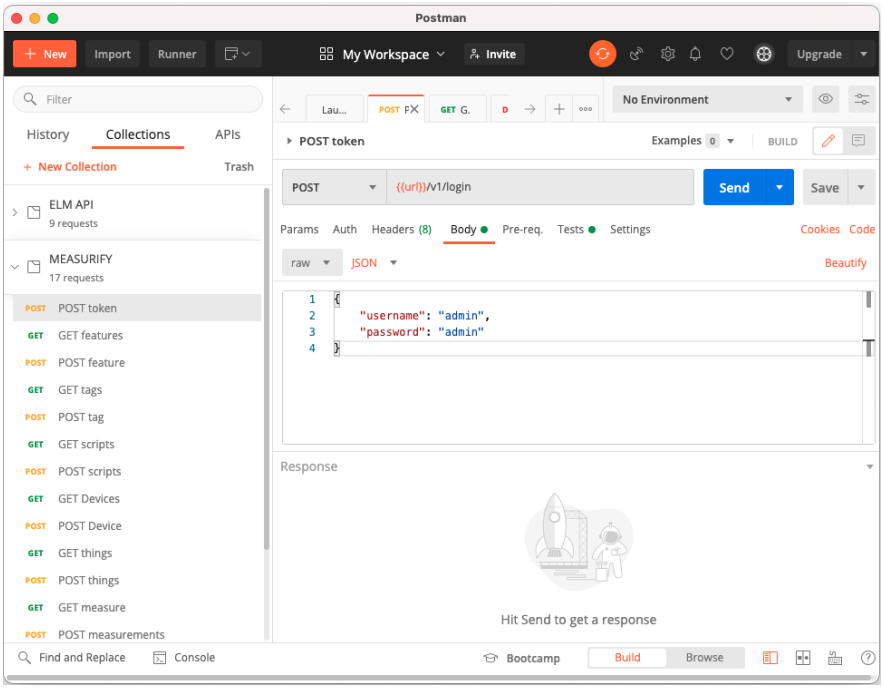


Figura 13: Interfaccia grafica di Postman

Abbiamo quindi addestrato due modelli, tra quelli disponibili in ELM, variando il campo "metadata" nella *computation*:

Primo modello

```
POST {{url}}/v1/computations
{
    "_id": "Model-1",
    "code": "model",
    "filter": "{\"feature\":\"smartHomeFeature\"}",
    "feature": "smartHomeFeature",
    "items": ["temp_int","hum_int","temp_est","target"],
    "tags": ["empty","detection"],
    "metadata": {
        "est": "{\"estimator\": \"KNeighborsClassifier\",\"
   n_neighbors_array\": [5, 10, 20]}"
    }
}
```

Secondo modello

```
POST {{url}}/v1/computations
{
    "_id": "Model-2",
    "code": "model",
    "filter": "{\"feature\":\"smartHomeFeature\"}",
    "feature": "smartHomeFeature",
```

```
    "items": ["temp_int","hum_int","temp_est","target"],
    "tags": ["empty","detection"],
    "metadata": {
        "est": "{\"estimator\": \"LinearSVC\",\"
   C_exp_lowerlimit\":-3,\"C_exp_upperlimit\":2,\"C_exp_step\"
   :2}"
    }
}
```

7.3 RISULTATI

Il workflow del sistema delle computation è quello desiderato e descritto nel diagramma UML in fase di progettazione (Fig. 8). Facendo una richiesta di tipo GET sulle *Computation* a fine addestramento, possiamo ottenere le informazioni riguardanti i risultati degli addestramenti.

Model-1

```
{
    "results": [
        {
            "best_params": "{'esti__n_neighbors': 5}",
            "metrics": "accuracy_score",
            "score": 0.7266666666666667
        }
    ],
    "items": [ "temp_int", "hum_int", "temp_est" ],
    "status": "concluded",
    "progress": 100,
    "visibility": "private",
    "tags": [ "empty", "detection" ],
    "_id": "Model-1",
    "code": "model",
    "filter": "{\"feature\":\"smartHomeFeature\"}",
    "feature": {
        "tags": [],
        "visibility": "public",
        "_id": "smartHomeFeature",
        "items": [
            {
                "dimension": 0,
                "type": "number",
                "name": "temp_int",
                "unit": "celsius degree"
            },
            {
                "dimension": 0,
                "type": "number",
```

```
                "name": "hum_int",
                "unit": "percent"
            },
            {
                "dimension": 0,
                "type": "number",
                "name": "temp_est",
                "unit": "celsius degree"
            }
        ],
        "owner": "5fcdf9c688dc7d37c3bda7b7"
    },
    "metadata": {
        "est": "{\"estimator\": \"KNeighborsClassifier\",\"
   n_neighbors_array\": [5, 10, 20]}",
        "model_id": "cd0bded0-19da-4dae-a3a5-1e117a8805ec"
    },
    "startDate": "2020-12-11T14:36:27.264Z"
}
```


Model-2


```
{
    "results": [
        {
            "best_params": "{'esti__C': 0.046415888336127795}"
   ,
            "metrics": "accuracy_score",
            "score": 0.823333333333333
        }
    ],
    "items": [ "temp_int", "hum_int", "temp_est" ],
    "status": "concluded",
    "progress": 100,
    "visibility": "private",
    "tags": [ "empty", "detection" ],
    "_id": "Model-2",
    "code": "model",
    "filter": "{\"feature\":\"smartHomeFeature\"}",
    "feature": {
        "tags": [],
        "visibility": "public",
        "_id": "smartHomeFeature",
        "items": [
            { /* ... */ }
        ],
        "owner": "5fcdf9c688dc7d37c3bda7b7"
    },
```

```
    "metadata": {
        "est": "{\"estimator\": \"LinearSVC\",\"
    C_exp_lowerlimit\":-3,\"C_exp_upperlimit\":2,\"C_exp_step\"
    :2}",
        "model_id": "f1dd1c86-317b-46ab-a6c5-a98b9b856aa9"
    },
    "startDate": "2020-12-11T14:39:09.749Z"
}
```


Dove nel campo "results", troviamo i risultati dell'addestramento del modello, con la scelta dei parametri migliori per avere uno score il più alto possibile. Come detto, questi valori non sono importati in quanto non abbiamo testato l'algoritmo di ML, ma è importante aver verificato che i due server abbiano comunicato in maniera corretta e che ELM sia riuscito ad arrivare a fine addestramento senza errori.

8 SVILUPPI FUTURI

Il test che abbiamo realizzato, per quando valido, si basa su uno scenario semplice, con una generazione di dati, sì realistici, ma del tutto arbitrari (costruiti ad hoc). Pertanto, sarebbe opportuno permettere di testare le funzionalità in situazioni reali: un progettista di applicazioni IoT o un data-scientist potrebbero realizzare dei test più complessi per testane l'utilità.

Come sviluppo futuro ci sarà sicuramente il *deployment* in un ambiente reale, quindi prendere questo oggetto e metterlo a disposizione di Measurify e sul server, e introdurlo in qualche progetto domotico o automotive reale. Questo prevedrà di andare a vedere se uno sviluppatore con a disposizione questo strumento sia in grado di usarlo e quanto questo incida sui tempi di sviluppo.

Per quanto riguarda la parte ELM, sicuramente si potrà seguire il suo sviluppo per implementare nuovi algoritmi di Machine Learning, sia supervisionati che non supervisionati, in modo da aumentare la scelta di possibili modelli.

La *Computation*, per ora consente solo una classificazione binaria. Un prossimo sviluppo sarà quello di consentire anche una classificazione multiclasse.

Attualmente la *Computation* ritorna una risposta solo dopo aver che ELM ha costruito il dataset. Questo potrebbe bloccare il server in caso di un numero molto alto di *Measurements*. Un prossimo miglioramento sarà quello di separare le operazioni, in modo tale che il server non rischi di rimanere bloccato: la *Computation* non dovrà più aspettare che il dataset sia stato creato per mandare la risposta al client.

BIBLIOGRAFIA

[1] R. Berta, A. Kobeissi, F. Bellotti, A. De Gloria. **Measurify, an Open Source Measurement-Oriented Data Framework for IoT**. IEEE Transactions on Industrial Informatics. 2020.

[2] F. Sakr, F. Bellotti, R. Berta, A. De Gloria. **Machine Learning on Mainstream Microcontrollers**. Sensor 2020.

[3] R. Berta. **Approccio Makers alla Progettazione Elettronica**. Slide corso 2019.

[4] S. Cheruvu, A. Kumar, N. Smith, D.M. Wheeler, **IoT Frameworks and Complexity," In: Demystifying Internet of Things Security**. Apress, Berkeley, CA, 2020, pp. 23-148.

[5] Atmani A., Kandrouch I., Hmina N., Chaoui H. (2020) **Big Data for Internet of Things: A Survey on IoT Frameworks and Platforms. In: Ezziyyani M. (eds) Advanced Intelligent Systems for Sustainable Development (AI2SD'2019)**. AI2SD 2019.

[6] M. Zúñiga-Prieto, J. González-Huerta, E. Insfran, and S. Abrahão. **Dynamic reconfiguration of cloud application architectures**. Journal of Software: Practice and Experience, vol. 48, pp. 327– 344, 2016.

[7] L. Jiang, L. D. Xu, H. Cai, Z. Jiang, F. Bu and B. Xu. **An IoT-Oriented Data Storage Framework in Cloud Computing Platform**. IEEE Transactions on Industrial Informatics, vol. 10, no. 2, pp. 1443-1451, May 2014.

[8] H. Cai, B. Xu, L. Jiang and A. V. Vasilakos. **IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges**. IEEE IoT Journal, vol. 4, no. 1, pp. 75-87, Feb. 2017.

[9] J. Fu, Y. Liu, H. Chao, B. K. Bhargava and Z. Zhang. **Secure Data Storage and Searching for Industrial IoT by Integrating Fog Computing and Cloud Computing**. IEEE Transactions on Industrial Informatics, vol. 14, no. 10, pp. 4519-4528, Oct. 2018.

[10] F. Paganelli, S. Turchi, and D. Giuli. **A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City**. IEEE Systems Journal, vol. 10, no. 4, pp. 1412-1423, Dec. 2016.

[11] S. K. Sharma and X. Wang. **Live Data Analytics With Collaborative Edge and Cloud Processing in Wireless IoT Networks**. IEEE Access, vol. 5, pp. 4621-4635, 2017.

[12] Lets Encrypt.

https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https.

[13] Scikit-Learn, Machine Learning in Python.

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html.

[14] Postman.

https://www.postman.com