Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

HUMAN ACTIVITY RECOGNITION SU DISPOSITIVI EMBEDDED: UN'ANALISI COMPARATIVA DI METODI DI APPRENDIMENTO AUTOMATICO

Relatore:
Chiar.mo Prof.
Angelo Trotta

Presentata da:
Giulio Billi

Correlatore:
Federico Montori

Seconda sessione - secondo appello
2022/2023

Abstract

La Human Activity Recognition è un campo di ricerca in grande espansione e ha come obbiettivo quello di riconoscere le attività svolte da un utente tra quelle presenti in un insieme di attività riconosciute. I modelli di deep learning utilizzati in questo campo utilizzano una grande quantità di dati provenienti dai sensori per ottenere un'alta precisione. Tuttavia, addestrare tali modelli con dati raccolti da dispositivi embedded comporta elevati costi di comunicazione e possibili violazioni della privacy. Inoltre un altro fattore da tenere in considerazione sono i limiti posti da tali dispositivi infatti a causa delle scarse risorse di cui dispongono, eseguire l'addestramento o l'etichettatura delle feature porta a un considerevole consumo di risorse. Queste problematiche portano all'esplorazione delle tecniche unsupervised e di metodi in grado di fornire modelli generalizzati da cui partire. Nel seguente elaborato vengono confrontati tre possibili sistemi basati su Self Organizing Maps: il sistema centralizzato, federato e singolo. Inoltre vengono tenute in considerazione le limitazioni poste dal dispositivo embedded e simulata, tramite Leave One Out Cross Validation, una situazione reale in cui un soggetto esterno vuole unirsi al sistema.

Keywords: *Human Activity Recognition, Federated Learning, Privacy, Self Organizing Maps, Machine Learning*

Introduzione

La **Human Activity Recognition** è un campo di ricerca in grande espansione che ha come obbiettivo quello di riconoscere le attività svolte da un utente tra quelle presenti in un insieme di attività riconosciute (descritte dalle label). I modelli di deep learning utilizzati in questo campo utilizzano una grande quantità di dati provenienti dai sensori per ottenere un'alta precisione. Tuttavia, addestrare tali modelli con dati raccolti da dispositivi embedded comporta elevati costi di comunicazione e possibili violazioni della privacy. Inoltre un altro fattore da tenere in considerazione sono i limiti posti da tali dispositivi, a causa dalle scarse risorse di cui dispongono. Infatti a prescindere dal modello di deep learning, per poterlo utilizzare è necessario effettuare l'addestramento sui dati raccolti ed è un processo molto costoso dal punto di vista delle risorse. Inoltre nel contesto dei dispositivi embedded i dati dei sensori tipicamente sono privi di label(spesso sono rappresentati come time series) e di conseguenza le più popolari tecniche supervised necessitano di un etichettatura manuale (non sempre possibile con flussi di dati continui) e spesso anche della creazione delle features, cosa che ha un impatto considerevole sul consumo di energia. In risposta a queste ultime problematiche lo studio condotto in [6] ha esplorato le tecniche unsupervised K-Means e Self Organizing Maps(SOM) confrontandole con i più comuni modelli supervised utilizzati per la HAR. Dallo studio è emerso che la SOM è una tecnica valida per la classificazione delle attività, in quanto ottiene un'accuratezza molto vicino alle tecniche supervised con un numero abbastanza limitato di features. Detto ciò, anche utilizzando una SOM,

il training rimane un'operazione dispendiosa nei dispositivi embedded e di conseguenza la possibilità di evitare l'addestramento grazie all'utilizzo di un modello generalizzato è molto allettante. Una possibile soluzione (utilizzata fino ad adesso) consiste in un sistema centralizzato, in cui tutti i dati raccolti dai dispositivi convergono in un server su cui viene addestrato il modello. Tuttavia come già accennato questo sistema può essere soggetto a problemi di privacy e di costi di comunicazione. Un'altra soluzione esente dai problemi di privacy consiste nell'utilizzare il federated learning per addestrare un modello generico combinando diversi modelli locali addestrati sui dati di client diversi [5]. Il seguente elaborato riprende i risultati ottenuti in [6] e li estende confrontando tre possibili sistemi basati su Self Organizing Maps: il sistema centralizzato, federato e singolo, tenendo in considerazione le limitazioni poste dall'M5Stack Gray(dispositivo utilizzato all'interno di [6]) e simulando, tramite Leave One Out Cross Validation, una situazione reale in cui un soggetto esterno vuole unirsi al sistema.

Indice

Elenco delle figure

Listings

Capitolo 1

Stato dell'arte

All'interno di questo capitolo verrà descritto lo stato dell'arte per le tecniche e gli argomenti presentati all'interno dell'elaborato

1.1 Human Activity Recognition

La Human Activity Recognition (HAR) è un ampio campo di ricerca il cui obiettivo è apprendere quale attività viene svolta da una determinata persona in un periodo di tempo specifico. Le attività possono essere di diversi tipi, ad esempio: seduta, in piedi, camminata, corsa, ciclismo o guida di un veicolo. Per riconoscere le attività, un modello di machine learning viene addestrato partendo dai dati grezzi di tali attività, raccolti mediante i sensori di diversi dispositivi (come smartphone, telecamere a circuito chiuso e dispositivi indossabili). L'uso di fonti diverse rende l'HAR importante per molteplici ambiti applicativi come il riconoscimento di attività quotidiane, sorveglianza, human-robot interaction e assistenza sanitaria a distanza. Di solito i sistemi per la HAR sono basati sia su tecniche di supervised learning sia su tecniche di unsupervised learning. Da una parte le tecniche che utilizzano supervised learning, come le Convolutional Neural Network forniscono una precisione maggiore nella classificazione delle attività, tuttavia richiedono dataset etichettati(spesso manualmente). Dall'altra parte le tecni-

che unsupervised permettono di utilizzare dataset non etichettati, scoprendo pattern ricorrenti al loro interno. Tuttavia rispetto alle tecniche supervised presentano una minore precisione. Nonostante i primi studi riguardanti la HAR risalgano alla fine degli anni 90, ci sono ancora numerosi problemi da affrontare, i quali motivano lo sviluppo di nuove tecniche sotto condizioni più realistiche. Alcune delle principali sfide sono la selezione degli attributi, la raccolta di dati sotto condizioni realistiche, il supporto a nuovi utenti in modo da non dover eseguire il training da zero e l'implementazione di tecniche di machine learning su dispositivi mobile con risorse limitate.[3]

1.2 HAR su dispositivi con risorse limitate

Come già accennato in precedenza una delle sfide principali della HAR è legato all'implementazione di modelli su dispositivi mobile. Come mostrato in [2] la maggior parte delle tecniche supervised, tra cui *FNN(Feedforward Neural Network)*, *SVM(Support Vector Machines)*, risultano essere molto costose per quanto riguarda il consumo di risorse (energia, potenza di calcolo e memoria) e come già detto in precedenza esse richiedono dati etichettati che in scenari realistici sono difficili da ottenere. Per rispondere al problema dell'etichettatura dei dati, tecniche unsupervised come *KNN (K-Nearest Neighbors)* sono state analizzate, tuttavia il pre-processing dei dati risulta costoso in termini di risorse e complessità.

1.3 Il problema della privacy

Di solito nel machine learning abbiamo un modello e abbiamo i dati che utilizziamo per addestrare il modello a svolgere un determinato compito, come riconoscere gli oggetti in delle immagini. I dati utilizzati per addestrare il modello sono generati da diversi dispositivi dislocati in diverse parti del globo e quindi per essere utilizzati devono essere raggruppati da qualche parte. L'approccio utilizzato fin'ora prevede che i dati siano raggruppati

su un server, ma per alcuni casi questo approccio non può essere applicato, soprattutto per casi d'uso legati al mondo reale.I principali ostacoli sono rappresentati da:

- Regolamenti: molti regolamenti come il GDPR in Europa, il CCPA in California il PIPEDA in Canada e molti altri, proteggono i dati sensibili degli utenti dall'essere inviati o dall'essere aggregati dalle organizzazioni, con lo scopo di essere utilizzati come dataset per l'addestramento dell'intelligenza artificiale.
- Preferenze degli utenti: spesso gli utenti non vogliono che alcuni dati particolarmente sensibili vengano inviati alle aziende produttrici dei dispositivi da loro posseduti.

Ad esempio in un caso di HAR, un nuovo utente che vuole entrare nel sistema, non vuole addestrare dall'inizio il suo modello e di conseguenza bisognerebbe avere un modello generalizzato da cui partire. Come già accennato, per renderlo possibile con l'attuale sistema, bisognerebbe raggruppare tutti i dati degli utenti su un server, ma a causa dei problemi legati alla privacy non è possibile. Di seguito sono presentati altri scenari reali in cui il modello centralizzato non è applicabile:

- Informazioni finanziare di diverse organizzazioni per rilevare frodi finanziarie.
- Dati sensibili da registrazioni sanitarie di diversi ospedali per allenare modelli per rilevare il cancro.

1.4 Federated Learning

Il federated learning [1] si pone come approccio in grado di superare i numerosi ostacoli che i casi d'uso del mondo reale, hanno posto al classico approccio utilizzato con il machine learning, in particolare il problema della privacy. Il federated learning scavalca gli ostacoli presentati in precedenza

invertendo l'approccio centralizzato, ovvero invece di spostare i dati dei vari utenti, viene spostato il modello ai vari utenti. Essenzialmente in questo approccio si ha un modello globale su un server che viene inizializzato randomicamente o da un checkpoint. Successivamente i parametri del modello globale vengono inviati ai vari client in modo da inizializzarli tutti con gli stessi parametri. Una volta inizializzati, i client iniziano l'addestramento sui loro dati locali e dopo averlo completato inviano i parametri del loro modello aggiornato al server. L'insieme dei modelli dei client viene "aggregato" (esistono diverse tecniche di aggregazione, ad esempio la media pesata dei parametri), in modo da ottenere un nuovo modello globale con i parametri aggiornati. Questo ciclo viene ripetuto più volte fino ad ottenere un modello globale che ottiene buoni risultati su tutti i client.

1.5 Obbiettivo della tesi

Dato il notevole aumento di dispositivi indossabili e mobili in grado di utilizzare sensori per raccogliere dati sulle attività svolte, la ricerca nel campo della Human Activity Recognition (HAR) sta attirando sempre più attenzione. Tuttavia, la maggior parte delle ricerche in questo ambito si avvale di una considerevole quantità di dati provenienti da diversi sensori per ottenere un'alta precisione. Come già evidenziato, questo approccio presenta problematiche sia in termini di privacy che di costi per la trasmissione dei dati. Come mostrato in [5], il federated learning è stato già considerato come soluzione a tali problemi. Tuttavia, come riscontrato nella maggior parte degli studi condotti finora, sono state valutate esclusivamente tecniche supervisionate. Di conseguenza, l'obiettivo di questa tesi si concentra sul confronto tra il modello centralizzato, quello singolo (in cui ogni utente ha il proprio modello senza uno generalizzato da cui partire) e quello federato, utilizzando una tecnica non supervisionata.

Capitolo 2

Il lavoro precedente

In questo capitolo, sarà presentato un riassunto della tesi magistrale su cui si basa il presente lavoro di ricerca.

2.1 Introduzione

I dispositivi indossabili dotati di sensori inerziali danno la possibilità di ottenere un riconoscimento personalizzato e dettagliato delle attività umane. Inoltre effettuare la classificazione su questi dispositivi (detti Extreme Edge) può ridurre la latenza e massimizzare la privacy dell'utente. Tuttavia le risorse limitate pongono sfide uniche che rendono le tecniche di Deep Learning non sempre applicabili e l'elaborazione dei dati onerosa. Nella tesi vengono affrontate le problematiche citate e proposta l'implementazione di un sistema HAR completo che tiene conto dei limiti posti dal dispositivo Extreme Edge. Il sistema implementato include un meccanismo per la selezione delle feature, in modo da ridurre la dimensionalità dei dati in ingresso e una tecnica di classificazione non supervisionata basata su Self-Organizing Maps (SOM), il tutto è stato sviluppato su una scheda IoT M5Stack. I risultati ottenuti dimostrano che la soluzione proposta è in grado di superare gli altri approcci non supervisionati e di raggiungere prestazioni vicine alle tecniche di Deep Learning utilizzate negli altri studi.[6]

2.2 Struttura del sistema

La soluzione proposta coinvolge tre fasi e un'architettura composta dal dispositivo IoT e da un server esterno. Inizialmente viene eseguita la selezione delle feature del dataset, con l'obbiettivo di ridurne la dimensionalità, in quanto solo alcune di esse possono essere correlate alle attività da rilevare. Successivamente le feature non etichettate vengono prima separate e poi classificate attraverso le Self-Organizing Maps, una tecnica di Machine Learning non supervisionata basata su reti neurali leggere. Infine il modello SOM addestrato viene implementato sull'M5Stack. Il processo di HAR è diviso in 3 fasi:

- Raccolta dati: in questa fase l'utente esegue azioni da rilevare e i dati grezzi provenienti dai sensori vengono campionati sull'M5Stack. I sensori in questione comprendono accelerometri, giroscopi e magnetometri.

- Addestramento del modello: in questa fase avviene l'addestramento della SOM dal set di dati raccolto dai dispositivi IoT. Inizialmente questi dati vengono pre-processati per rimuovere valori anomali o mancanti. Successivamente avviene la selezione delle feature attraverso la tecnica ANOVA-F e poi l'addestramento del modello che si affida alla tecnica SOM. Infine il modello addestrato, con l'elenco delle feature selezionate viene trasferito al dispositivo IoT.

- inferenza di HAR: in modalità operativa, il modello SOM addestrato viene automaticamente trasferito al dispositivo IoT. Le caratteristiche selezionate nella fase precedente vengono estratte dai campioni IMU e fornite in input alla SOM per la classificazione dell'attività umana.

2.2.1 Self-Organizing Maps

Le Self-Organizing Maps o SOM, sono un tipo di Artificial Neural Network che sfruttano l'apprendimento non supervisionato, un metodo utilizzato per

trovare pattern all'interno di un dataset, dando la possibilità di classificare dati non etichettati o di eseguire un un'analisi esplorativa per comprendere relazioni tra i pattern. In particolare le SOM permettono la riduzione di un spazio multidimensionale in uno spazio bidimensionale, ovvero un griglia. Infatti in una SOM i neuroni sono organizzati in una griglia bidimensionale e ogni neurone è completamente connesso a tutti i nodi sorgente nello strato di input. A ogni neurone è associato un vettore dei pesi della stessa dimensione dello spazio di osservazione. I pesi di ogni neurone si evolvono progressivamente seguendo l'algoritmo di apprendimento finché non si stabilizza e l'organizzazione dei neuroni riflette l'organizzazione vettoriale dello spazio di input(dati simili nello spazio di osservazione corrispondono a regioni vicine sulla mappa). Essendo una tecnica di machine learning non supervisionato, è possibile utilizzare dati non etichettati per l'addestramento e alla fine è possibile eseguire una classificazione dopo aver identificato gruppi di neuroni che rappresentano lo stesso pattern. Nello specifico, il funzionamento della SOM è il seguente: prima di tutto vengono inizializzati i pesi dei neuroni in modo casuale. Successivamente un vettore di input viene presentato alla mappa e viene calcolata la distanza di manhattan tra ogni neurone e il vettore in input. In una mappa 2D la distanza di Manhattan tra due punti p_i e p_j con coordinate (X_{p_i}, Y_{p_i}) e (X_{p_j}, Y_{p_j}) è definita dalla formula:

$$d_m(p_i, p_j) = |X_{p_j} - X_{p_i}| + |Y_{p_j} - Y_{p_i}|$$

Il neurone con la distanza minore dal vettore di input viene chiamata Best Matching Unit(BMU) e corrisponde alla rappresentazione migliore dei dati in input. Una volta trovato il neurone vincitore vengono aggiornati i suoi pesi e quelli del suo vicinato in modo da essere più vicini al vettore di input. In particolare i pesi w_i di ogni neurone i vengono aggiornati secondo la seguente formula:

$$\Delta w_i = \eta(t)h(t, i, s)(v - w_i)$$

Dove $\eta(t)$ indica il learning rate al tempo t(il rate diminuisce con l'aumentare del tempo), s il neurone vincente o BMU, v il vettore in input e h

la seguente funzione di vicinato che :

$$h(t, i, s) = e^{\frac{d_m(p_i, p_s)^2}{2\Theta(t)^2}}$$

In cui $\Theta(t)$ indica la dimensione del vicinato al tempo t (con l'aumentare del tempo Θ diminuisce).

Quindi dopo il processo di addestramento, viene creato un sistema di coordinate significativo per le caratteristiche di input sulla griglia e se consideriamo le coordinate del neurone vincente associato a ciascun pattern, la SOM forma una mappa topografica dei pattern di input.

2.2.2 ANOVA-F Feature Reduction

Quando si tratta di creare modelli di Machine Learning efficienti per la HAR la riduzione delle feature è essenziale. Di solito la maggior parte degli algoritmi è in grado di ignorare internamente le caratteristiche che hanno un'influenza minima. Tuttavia nel caso di dispositivi con risorse limitate è importantissimo ridurre il numero di caratteristiche senza alterare l'accuratezza. La tecnica ANOVA-F è un metodo statistico utilizzato per la selezione delle caratteristiche in grado di identificare le caratteristiche più significative per il modello. Per calcolare il valore ANOVA-F di una caratteristica f, calcoliamo innanzitutto, per ciascuna delle K classi, la varianza di tale caratteristica all'interno di tale classe, quindi la dividiamo per la varianza di tale caratteristica sull'intero set di dati. Più formalmente, il valore ANOVA-F della caratteristica f per la classe $c \in K$ è dato da:

$$\mathrm{An}_c(f) = \frac{\sigma^2(f)_c}{\sigma^2(f)}$$

Idealmente, un valore basso di $An_c(f)$ significa che la feature f è discriminante per la classe c, perché i valori di f non cambiano significativamente tra i membri della classe c rispetto a quanto cambiano nel set di dati. Invece se i valori di $An_c(f)$ si avvicinano a 1.0 o più indicano che la caratteristica f è probabile che aggiunga solo rumore per la classe c. Per ciascuna feature f

dobbiamo trovare un unico valore e di conseguenza è necessario eseguire una funzione di aggregazione. Nell'esperimento sono stati presi in considerazione due metodi i quali, scelgono rispettivamente la media e il minimo come funzione di aggregazione. Una volta calcolati si introduce la soglia ANOVA-F che il valore al di sotto del quale vengono selezionate le feature.

2.3 Risultati

I risultati sperimentali, ottenuti su due set di dati HAR, il set di dati UCI HAR e un set di dati personalizzato costruito attraverso l'M5Stack, hanno dimostrato che la soluzione SOM proposta è in grado di superare altri approcci non supervisionati come K-Means. Inoltre, il sistema ha raggiunto prestazioni vicine alle tecniche di Deep Learning supervisionate (LSTM, CNN, CNN-LSTM, CONVLSTM2D), pur generando un modello sufficientemente piccolo da adattarsi alle limitate capacità di memoria dei dispositivi Extreme Edge.

Capitolo 3

Struttura e evoluzione del progetto

In questo capitolo verrà presentata la struttura del progetto e la sua evoluzione.

3.1 UCI HAR Dataset

Il Dataset utilizzato per l'esperimento è un dataset ospitato dalla University of California Irvine (UCI) [4], per la human activity recognition ed è stato utilizzato per numerose pubblicazioni in questo ambito. Il dataset contiene informazioni di movimento campionate da accelerometro e giroscopio di uno smartphone portato da 30 volontari che svolgevano sei diverse attività: camminare, salire le scale, scendere le scale, sedersi, stare in piedi e sdraiarsi. Il dataset contiene due tipi di dati:

1. Dati temporali grezzi
2. Features pre-calcolate, ovvero metriche statistiche calcolate sui dati grezzi, a loro volta divise in due parti:
    - statistiche calcolate sulle sequenze temprali

- statistiche calcolate attraverso l'applicazione della Fast Fourier Transform (FFT).

Più precisamente, ogni record nel dataset è fornito di un vettore di 561 caratteristiche con variabili di dominio temporale e frequenziale. Le feature del dataset sono state ottenute dai segnali grezzi tridimensionali dell'accelerometro e del giroscopio e sono state acquisite a una frequenza costante di 50Hz. Per eliminare il rumore, è stato utilizzato un median filter e un low pass Butterworth filter di terzo ordine con una frequenza angolare di taglio di 20Hz. Un altro low pass Butterworth filter con una frequenza angolare di taglio di 0.3Hz è stato utilizzato per separare il segnale di accelerazione in segnali di accelerazione corporea e gravitazionale. Successivamente, l'accelerazione lineare e la velocità angolare del corpo sono state utilizzate per derivare i segnali di Jerk nel tempo. La magnitudine di questi segnali tridimensionali è stata determinata utilizzando la norma euclidea. Infine su alcuni di questi risultati è stata applicata la Fast Fourier Transform (FFT). Spesso vengono utilizzate le features pre-calcolate per la valutazione dei metodi in quanto forniscono una maggiore accuratezza ma è necessario tenere a mente che molte di esse sono state generate utilizzando Fast Fourier Transforms (FFT) computazionalmente molto costosa. Infatti per il nostro esperimento, sull'intero set di 561 features è stato considerato un sottoinsieme di 265 features che esclude quelle calcolate attraverso FFT,in quanto, come dimostrato in [6], garantiva un'accuratezza molto simile a quella calcolata utilizzando tutte le features. Come già anticipato il dataset contiene dati raccolti da 30 volontari, di conseguenza, per simulare una situazione realistica, lo abbiamo diviso in 30 sottoinsiemi, uno per ogni soggetto. In particolare abbiamo raggruppato le misurazioni di ogni soggetto e le abbiamo divise in train e test con un rapporto 70 - 30, mantenendo il rapporto tra classi originale attraverso una tecnica chiamata *Stratified train-test split.*

3.2 Struttura del progetto

La configurazione del progetto ha attraversato variazioni nel corso del suo sviluppo, adattandosi in risposta ai risultati ottenuti. Possiamo identificare due strutture principali, le quali rispecchiano i principali test che sono stati eseguiti.

3.2.1 Prima fase: confronto tra sistema centralizzato, singolo e federato

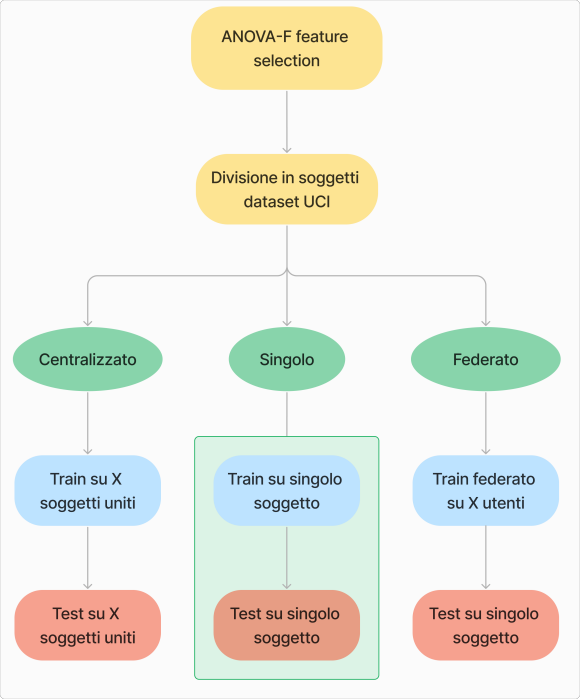


Figura 3.1: Struttura 1

Come mostrato in 3.1 l'obbiettivo principale di questa prima fase è stato quello di ricreare la situazione finale della tesi magistrale, modificando la pipeline in modo da spostare il focus sul confronto tra i tre sistemi: centralizzato, singolo e federato. Seguendo l'immagine, vediamo come inizialmente avviene la creazione dei dataset di ogni soggetto, ognuno diviso in test e

train, ottenendo così 60 file tra train e test con le feature già selezionate dall'analisi ANOVA-F (per questa fase i risultati sono stati calcolate con tutte le 265 feature). Successivamente vengono eseguiti i tre sistemi su un gruppo di soggetti scelto in modo randomico. In particolare, dopo aver caricato i dataset dei soggetti scelti, il train sui tre sistemi avviene nel seguente modo:

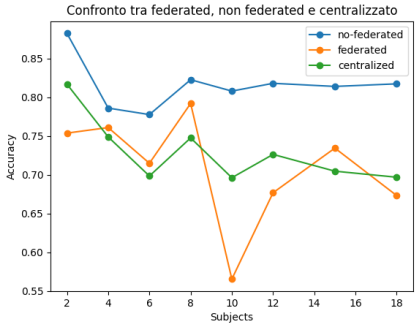
1. **Centralizzato**: in quanto sistema centralizzato, il dataset su cui viene fatto il train della SOM è uno solo, ed è composto dall'unione dei dataset dei singoli soggetti(la stessa cosa vale anche per il dataset di test). Una volta eseguito il train è possibile eseguire la classificazione, ottenendo così l'accuracy.

2. **Singolo**: questo sistema simula il caso in cui un soggetto decida di eseguire il train esclusivamente sui suoi dati, senza utilizzare nessun modello già addestrato. Quindi per ogni soggetto selezionato viene eseguito il train e la classificazione sul suo dataset e alla fine viene fatta la media delle accuracy ottenute (facendo riferimento a 3.1 i passaggi all'interno dell'area verde vengono ripetuti per ogni soggetto).

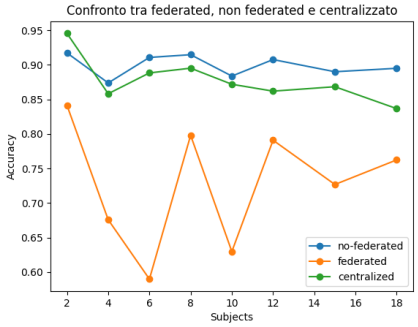
3. **Federato**: il sistema federato dopo aver inizializzato il modello generalizzato e i client dei soggetti con gli stessi pesi iniziali inizia l'addestramento. In particolare ogni client(uno per ogni soggetto) addestra il suo modello locale sul suo dataset di train e invia i pesi aggiornati al server centrale, il quale aggrega tutti i pesi ottenuti tramite la loro media pesata, in questo modo se un client ha eseguito il train su pochi dati ha un'influenza minore di uno che lo ha eseguito su un numero maggiore di dati. Dopo l'aggiornamento del modello generalizzato, i suoi pesi vengono inviati ai client che eseguono la classificazione sui loro dataset di test. Questo processo viene eseguito per un determinato numero di round, e per ogni round viene calcolata la media delle accuracy dei client.

I tre sistemi sono stati testati con gruppi di soggetti di diverse dimensioni per determinare la bontà di ogni sistema e in che modo un numero maggiore di

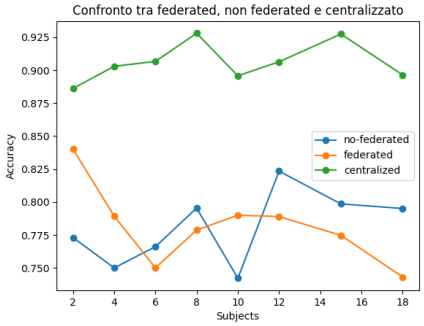
soggetti potesse influenzarne l'accuratezza. Inoltre le misurazioni sono state eseguite per diverse dimensioni della SOM, per osservare come la dimensione ne influenzasse il risultato. Di seguito vengono presentati i risultati ottenuti con i 3 sistemi.



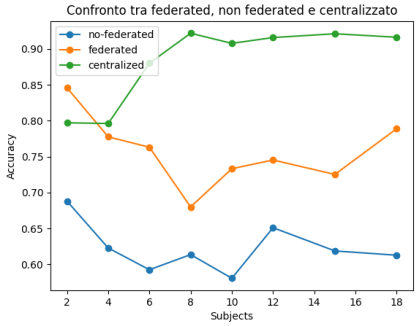
(a) SOM 10x10



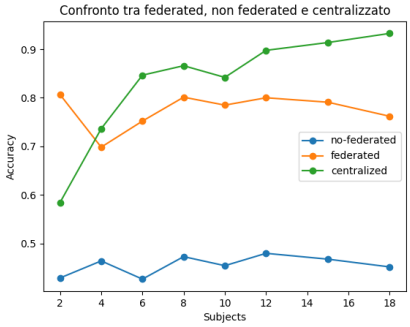
(b) SOM 20x20



(c) SOM 30x30



(d) SOM 40x40

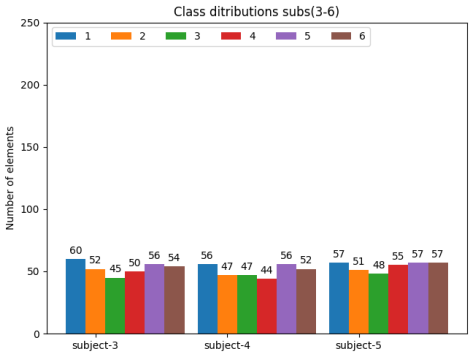


(e) SOM 50x50

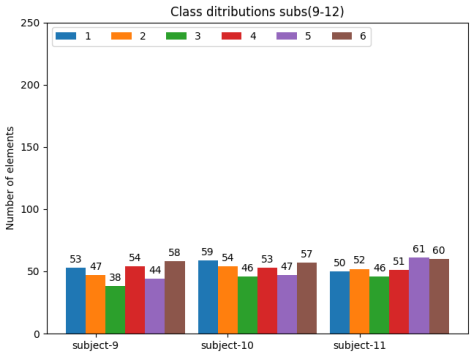
Figura 3.2: Confronto accuracies prima fase

Come possiamo notare la SOM tende a funzionare meglio nel caso singolo (nei grafici indicato come no-federated), se non nel caso di SOM molto grandi in cui il numero maggiore di neuroni e il numero ridotto di dati di train provoca un calo nell'accuratezza del sistema. Tuttavia SOM di dimensioni molto grandi, come 40x40 e 50x50 per essere implementate sul dispositivo sono soggette ad un limite massimo di feature. Questo limite è necessario in quanto per l'implementazione su dispositivi embedded il modello deve essere "leggero" e quindi maggiore è la dimensione della SOM minore deve essere il numero massimo di features.

Una volta analizzati i risultati, per essere sicuri della bontà del sistema federato è stata controllata la distribuzione delle classi nei dataset di ogni soggetto in modo da escludere la possibilità che fossero *Non i.i.d. (Independent and Identically Distributed)*, ovvero soggetti con classi predominanti. I test ottenuti hanno confermato che i dataset dei soggetti hanno una distribuzione di classi simile, confermando così la correttezza dei risultati. Di seguito alcuni esempi di distribuzione delle classi.



(a) Distrbuzione soggetti 3-5



(b) Distrbuzione soggetti 9-11

Figura 3.3: Distribuzione classi

3.2.2 Seconda fase: confronto sistemi con Leave One Out Cross Validation

Dai risultati della fase precedente è chiaro come la SOM funzioni meglio nel caso singolo, soprattutto meglio del sistema federato. Tuttavia, nel caso singolo, ogni volta che un nuovo client vuole entrare nel sistema, dovrà necessariamente effettuare un training sulle proprie attività per avere un modello funzionante. Alternativamente potrebbe prendere il modello addestrato da qualcun altro, ma sarebbe specializzato su un altro soggetto e di conseguenza funzionerebbe male. Una possibile soluzione è quella di avere un modello addestrato da più soggetti e quindi più generalizzato e meno specializzato su un unico soggetto. Come già anticipato questa soluzione è percorribile soltanto tramite il sistema federato, in quanto con quello centralizzato ci sarebbe il problema della privacy. La seconda fase si basa sul ragionamento appena presentato e di conseguenza è stato necessario modificare leggermente la struttura precedente.

Osservando la figura 3.4 è possibile notare che il focus di questa fase è sempre il confronto tra i tre tipi di sistema, ma simulando una situazione differente della precedente. Infatti le modifiche apportate si concentrano sul riprodurre il caso in cui un utente vuole entrare nel sistema, per ciascuno dei tre sistemi. Per il caso Centralizzato e Federato è stata eseguita una Leave One Out Cross Validation (LOOCV) la quale consiste nell'eseguire l'addestramento su 29 soggetti e poi fare il testing del modello sul dataset del trentesimo soggetto escluso. Per quanto riguarda il caso singolo si considera una coppia di soggetti (A, B), si esegue l'addestramento su B e il test su A. In particolare questa fase presenta la seguente organizzazione:

- **Centralizzato**: in questo caso viene eseguita la LOOCV su tutte le combinazioni possibili con i trenta soggetti (trenta combinazioni). In particolare l'addestramento avviene sui dataset uniti dei 29 soggetti e il test sul trentesimo lasciato fuori. Per ogni combinazione viene salvata l'accuracy ottenuta e infine si considera la loro media.

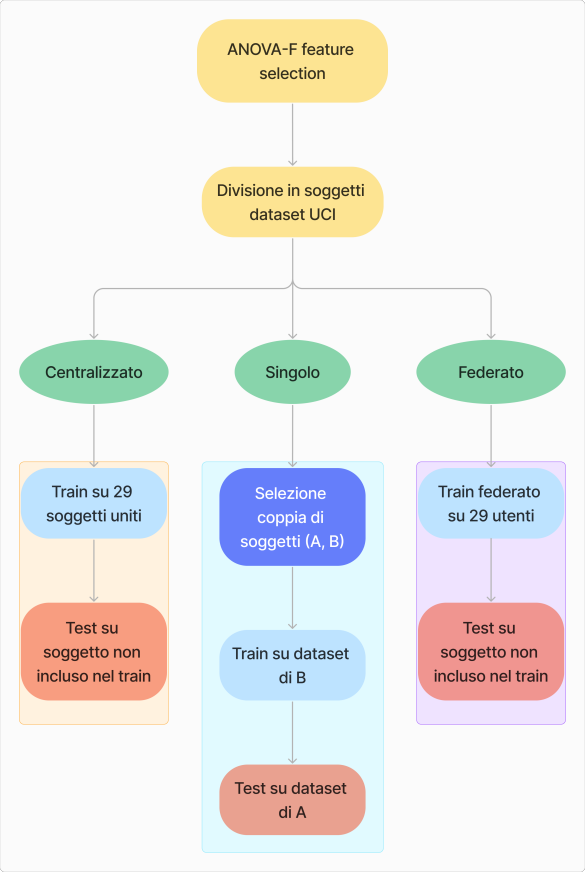


Figura 3.4: Struttura 2

- **Singolo**: nel sistema singolo vengono considerate tutte le coppie possibili considerando tutti e trenta i soggetti (435 coppie possibili). Per ogni coppia (A, B), si esegue l'addestramento su B e il test su A e alla fine si considera la media delle accuracy.

- **Federato**: nel caso federato viene eseguita la LOOCV su tutte e trenta le combinazioni possibili, effettuando l'addestramento federato sui 29 client e poi il test sul trentesimo utilizzando il modello generalizzato.

Anche in questa fase ogni sistema è stato testato con diverse dimensioni della SOM, questa volta con dimensioni concretamente implementabili sul dispositivo (10x10, 15x15, 20x20, 30x30).

3.2.3 Confronto finale

Una volta ottenuti tutti i risultati delle strutture proposte è stato effettuato un altro test, con l'obbiettivo di avere un confronto globale dei vari sistemi in tutte le configurazioni proposte. Inoltre per simulare nel modo più fedele possibile le limitazioni imposte dal dispositivo, nel confronto è stato considerato anche il caso in cui il numero di feature utilizzate varia in base alla dimensione utilizzata per la SOM. Infatti a causa delle risorse limitate del dispositivo, per mantenere il modello "leggero" è necessario diminuire il numero di feature utilizzate all'aumentare della dimensione della SOM. Più precisamente per questo ultimo test sono state considerate le seguenti configurazioni:

1. in questa configurazione viene utilizzata la prima struttura in cui nel sistema singolo viene eseguito train e test sullo stesso utente, nel sistema federato e centralizzato si fa il train con 30 soggetti e il test su uno solo. Il tutto viene ripetuto con tutti i 30 soggetti possibili.

2. la seconda configurazione è identica alla seconda struttura in cui viene effettuata la LOOCV.

3. l'ultima configurazione considerata esegue la seconda struttura ma variando il numero di features in base alla dimensione della SOM. In particolare abbiamo:

    - SOM size: 10 - features: 265 (tutte le feature)
    - SOM size: 15 - features: 158
    - SOM size: 20 - features: 89
    - SOM size: 30 - features: 39

Per ognuna di queste configurazioni i risultati vengono salvati sotto forma di JSON in un file txt, in modo da poterli riutilizzare per costruire dei grafici (nel nostro caso boxplot).

Capitolo 4

Implementazione

Nei capitoli precedenti, abbiamo esaminato l'esperimento da una prospettiva principalmente teorica, tralasciando i dettagli tecnici. Tuttavia, in questo capitolo ci concentreremo sulla presentazione degli aspetti implementativi fondamentali del progetto. La struttura dell'esperimento si articola in due macro-sezioni: la prima è dedicata alla manipolazione del dataset, mentre la seconda riguarda il test dei vari sistemi proposti.

4.1 Organizzazione dei file

Prima di mostrare effettivamente l'implementazione delle varie parti dell'esperimento, iniziamo descrivendo i vari file che lo compongono:

- **UCI HAR Dataset** contiene il dataset UCI originale. [4]
- **UCI HAR Dataset split** contiene il dataset UCI diviso in soggetti. Se è presente anche un numero indica il numero di feature utilizzate nel dataset.
- **subjects_accs_mean** contiene le accuracy raccolte dai tre sistemi durante la prima fase.
- **subjects_accs** contiene le accuracy raccolte nel confronto finale e i boxplot prodotti.

- **subjects_accs/loocv_standard** contiene le accuracy raccolte durante la seconda fase e il boxplot prodotto.
- **main.py** script Python nucleo centrale dell'esperimento. In esso viene eseguito il train e il test dei 3 sistemi, il salvataggio dei risultati e la creazione dei boxplot. Prende in input i seguenti comandi:
  - **"load"** o **"gen"** o **"a-gen"** rispettivamente per caricare il dataset, generarlo senza analisi anova o generarlo con analisi anova.
  - un numero da uno a 30 per scegliere quanti soggetti utilizzare.
  - **"centr"** o **"no-centr"** rispettivamente per includere nel test il sistema centralizzato oppure no.
  - **"sing"** o **"no-sing"** rispettivamente per includere nel test il sistema singolo oppure no.
  - **"fed"** o **"no-fed"** rispettivamente per includere nel test il sistema federato oppure no.
- **anovaf.py** contiene l'implementazione dell'analisi anova.
- **plots.py** contiene tutti i metodi dedicati alla creazione dei vari grafici.
- **ML_utilis.py** contiene alcuni metodi di supporto per il salvataggio dei risultati e la gestione delle varie combinazioni di soggetti.
- **utils.py** contiene metodi di supporto più generali come l'inizializzazione delle directory e la creazione e il caricamento del dataset.

4.2 Manipolazione dataset

Il dataset UCI contiene i dati dei vari soggetti uniti e ogni campione è associato a un soggetto indicato nei file *subject_test.txt* e *subject_train.txt*. Di conseguenza è stato necessario raggruppare i dati dei vari soggetti in modo da ottenere i dataset di train e test per ogni soggetto con un rapporto 70-30.

```
def create_subjects_datasets(anova_selection, max_n):
    # carico e unisco il dataset (considerando 265 feature)
    uci_x_train, uci_y_train = load_dataset_group("train", pathPrefix, 265)
    uci_x_test, uci_y_test = load_dataset_group("test", pathPrefix, 265)

    # selezione feature ANOVA
    a_y_train = uci_y_train - 1
    a_y_test = uci_y_test - 1

    var_avg_c, var_min_c  = get_anovaf(uci_x_train, tf.keras.utils.to_categorical(a_y_train), uci_x_test, tf.keras.utils.to_categorical(a_y_test))
    #var_avg_c, var_min_c  = get_anovaF(uci_x_train, tf.keras.utils.to_categorical(a_y_train), uci_x_test, tf.keras.utils.to_categorical(a_y_test))

    #anova_x_train, anova_x_test = feature_selection_anova(uci_x_train, uci_x_test, 1.0, var_avg_c)
    anova_x_train, anova_x_test = feature_selection_with_max(uci_x_train, uci_x_test,var_avg_c, max_n)

    anova_X = np.concatenate((anova_x_train, anova_x_test))


    X = np.concatenate((uci_x_train, uci_x_test))
    y = np.concatenate((uci_y_train, uci_y_test))

    for subject in train_subjects:
        # prendo il dataset del soggetto corrispondente all'index
        datasetX = []
        datasety = []

        if anova_selection:
            datasetX, datasety = dataset_for_subject(anova_X, y, sub_map, subject)
```

```
29          else:
30              datasetX, datasety = dataset_for_subject(X, y,
    sub_map, subject)
31
32          # split subject dataset in 70% train and 30% test
33          s_trainX, s_testX, s_trainy, s_testy =
    train_test_split(datasetX, datasety, train_size=0.70,
    random_state=42, shuffle=True, stratify=datasety)
34
35          groups = ["train", "test"]
36          # salvo il dataset in file csv
37          for group in groups:
38              if not os.path.exists("./UCI HAR Dataset split/"+
     group +"/subject-" + str(subject)):
39                  os.mkdir("./UCI HAR Dataset split/"+ group +"
    /subject-" + str(subject))
40
41          save_dataset(s_trainX, s_trainy, "./UCI HAR Dataset
    split/" + "train/" + "subject-" + str(subject) + "/",
    subject, "train")
42          save_dataset(s_testX, s_testy, "./UCI HAR Dataset
    split/" + "test/" + "subject-" + str(subject) + "/",
    subject, "test")
```

Listing 4.1: Definizione creazione dataset

Come mostrato nel codice riportato viene caricato il dataset UCI completo, sul quale viene effettuata l'analisi ANOVA e la selezione delle feature. Successivamente, per ogni soggetto, vengono presi i suoi dati, divisi in 70% train e 30% test, mantenendo il rapporto tra classi originale grazie alla tecnica *Stratified train-test split*, e alla fine vengono salvati come file csv. Una volta salvati i dataset di ogni soggetto possono essere caricati attraverso il seguente script.

```
1 def load_subjects_dataset(subjects_to_ld, output_mode,
    dataset_feature):
2     trainX, trainy = load_subjects_group("train",
    subjects_to_ld, output_mode, pathPrefix)
```

```
    testX, testy = load_subjects_group("test", subjects_to_ld
, output_mode, pathPrefix)

    # zero-offset class values to perform one-hot encode (
default values 1-6)
    if output_mode == "concat":
        trainy = trainy - 1
        testy = testy - 1

        # one hot encode y
        trainy = tf.keras.utils.to_categorical(trainy)

        testy = tf.keras.utils.to_categorical(testy)
    else:
        for idx, elem in enumerate(trainy):
            trainy[idx] -= 1
            trainy[idx] = tf.keras.utils.to_categorical(
trainy[idx])

        for idx, elem in enumerate(testy):
            testy[idx] -= 1
            testy[idx] = tf.keras.utils.to_categorical(testy[
idx])

    return  trainX, trainy, testX, testy
```

Listing 4.2: Definizione caricamento dataset

Come notiamo dal codice riportato è possibile caricare i dataset dei soggetti concatenandoli oppure mantenendoli separati e quindi sotto forma di lista di dataset. Queste due modalità sono utilizzate rispettivamente per il sistema centralizzato e per gli altri due sistemi.

4.3 Implementazione dei test

In questa sezione verrà presentato il corpo centrale dell'esperimento. Per prima cosa verranno mostrate le funzioni utilizzate per gestire l'esecuzione

dei test e poi si passerà all'implementazione effettiva dei tre sistemi: centralizzato, singolo e federato. Partiamo con la presentazione della funzione principale *run*, la quale gestisce il flusso di esecuzione dell'esperimento.

```
def run():
    if sys.argv[1] == "gen":
        create_subjects_datasets(False, 265)
    elif sys.argv[1] == "a-gen":
        create_subjects_datasets(True, 39)

    subjects_to_ld=random.sample(range(1, 31), int(
    subjects_number))
    subjects_to_ld.sort()
    print("subjects", subjects_to_ld)
    #calculate_class_distribution()


    # calcolo tutte le combinazioni possibili in cui ci sono
    29 elementi
    # e uno rimane fuori
    combinations = list(itertools.combinations(subjects_to_ld
    , len(subjects_to_ld) - 1))


    for dim in dimensions:
        current_som_dim = dim
        train_noloocv(subjects_to_ld, dim)
        train_loocv_combinations(subjects_to_ld, dim,
    combinations, "std-loocv")
        train_loocv_combinations(subjects_to_ld, dim,
    combinations, "feat-loocv")


    save_accuracies(single_accs_combinations,
    federated_accs_combinations, centr_accs_combinations,
    accs_path, dimensions)
    plot_boxplot(dimensions, accs_path)
```

Listing 4.3: Definizione funzione run

Da questo script possiamo avere una chiara visione di come si articola l'esperimento: inizialmente, avviene il caricamento/creazione del dataset; successivamente, vengono calcolate le combinazioni possibili attraverso la funzione *itertools.combinations*; poi vengono eseguiti i tre test (senza LOOCV, LOOCV senza variazione delle feature e LOOCV con variazione delle feature); infine, i risultati vengono salvati e stampati sotto forma di grafici. Ora che è stata mostrato il nucleo dell'esperimento possiamo passare ad osservare più da vicino l'implementazione dei sistemi e in particolare dei tre test eseguiti nel confronto finale.

4.3.1 Implementazione test senza LOOCV

Come spiegato nel terzo capitolo in questo test non avviene la LOOCV e quindi non si ha una vera e propria simulazione del soggetto che vuole entrare nel sistema. L'esecuzione del test è affidata alla funzione *train_noloocv* la quale gestisce l'esecuzione del sistema centralizzato, singolo e federato con la configurazione presentata nel capitolo 3.

```
1  def train_noloocv(subjects_to_ld, dim):
2      ...
3
4      trainX, trainy, testX, testy = load_subjects_dataset(
       subjects_to_ld, "concat", "full")
5      trainX_lst, trainy_lst, testX_lst, testy_lst =
       load_subjects_dataset(subjects_to_ld, "separated", "full")
6
7      # deve fare il train e test per ogni soggetto
8      if single:
9          train_single(trainX_lst, trainy_lst, testX_lst,
       testy_lst, subjects_to_ld, dim)
10         print("single accs", single_accs_combinations)
11     # deve fare il federated con train su tutti e 30 e test
       su tutti e 30
12     if federated:
13         fed_Xtrain = trainX_lst
14         fed_ytrain = trainy_lst
```

```
15          fed_Xtest = testX_lst
16          fed_ytest = testy_lst
17          federated_accs_combinations["noloocv"].setdefault(dim
    , [])
18
19          accuracies, _ = train_federated(5, len(fed_Xtrain), "
    noloocv")
20          print("fed accs", accuracies)
21          federated_accs_combinations["noloocv"][dim] =
    accuracies[-1][-1]
22
23      if centralized:
24      # deve fare il centralizzato con tutti e 30 e il test su
    ognuno
25          centr_accs_combinations["noloocv"].setdefault(dim,
    [])
26          trained_som = train_noloocv_centr(trainX, dim)
27          # sulla som trainata devo fare la classificazione su
    tutti i test
28          for subj_idx, subj in enumerate(subjects_to_ld):
29              class_report = classification_report(
    onehot_decoding(testy_lst[subj_idx]), classify_fed(
    trained_som, testX_lst[subj_idx], trainX, trainy),
    zero_division=0.0, output_dict=True)
30              centr_accs_combinations["noloocv"][dim].append(
    class_report["accuracy"])
```

Listing 4.4: Implementazione test senza LOOCV

4.3.2 Implementazione test con LOOCV

Di seguito viene mostrata l'implementazione della funzione utilizzata per eseguire i test con la *Leave One Out Cross Validation*.

```
1 def train_loocv_combinations(subjects_to_ld, dim,
    combinations, loocv_type):
2     ...
3
```

```
4     if loocv_type == "std-loocv":
5         trainX_lst, trainy_lst, testX_lst, testy_lst =
  load_subjects_dataset(subjects_to_ld, "separated", "full")
6         num_feat = "full"
7     elif loocv_type == "feat-loocv":
8
9         if dim == 10:
10            trainX_lst, trainy_lst, testX_lst, testy_lst =
  load_subjects_dataset(subjects_to_ld, "separated", "full")
11            num_feat = "full"
12        elif dim == 15:
13            trainX_lst, trainy_lst, testX_lst, testy_lst =
  load_subjects_dataset(subjects_to_ld, "separated", "158")
14            num_feat = "158"
15        elif dim == 20:
16            trainX_lst, trainy_lst, testX_lst, testy_lst =
  load_subjects_dataset(subjects_to_ld, "separated", "89")
17            num_feat = "89"
18        elif dim == 30:
19            trainX_lst, trainy_lst, testX_lst, testy_lst =
  load_subjects_dataset(subjects_to_ld, "separated", "39")
20            num_feat = "39"
21
22
23    print("trainX lst shape", trainX_lst[0].shape)
24    if federated:
25        train_combinations_federated(trainX_lst, trainy_lst,
  testX_lst, testy_lst, subjects_to_ld, dim, combinations,
  loocv_type)
26
27    if single:
28        train_combinations_single(trainX_lst, trainy_lst,
  testX_lst, testy_lst, subjects_to_ld, dim, "no-centr",
  loocv_type)
29    if centralized:
30        train_combinations_centralized(dim, combinations,
  loocv_type, num_feat)
31
```

```
32 def train_combinations_federated(trainX_lst, trainy_lst,
   testX_lst, testy_lst, subjects_to_ld, dimension,
   combinations, loocv_type):
33     ...
34     for combination in filtered_combinations:
35 
36         test_subject = find_missing_element(combination, 1,
   31)
37         fed_Xtrain = filter_list_by_index(trainX_lst,
   combination)
38         fed_ytrain = filter_list_by_index(trainy_lst,
   combination)
39         fed_Xtest = filter_list_by_index(testX_lst,
   combination)
40         fed_ytest = filter_list_by_index(testy_lst,
   combination)
41 
42         loocv_Xtrain = trainX_lst[test_subject[0] - 1]
43         loocv_ytrain = trainy_lst[test_subject[0] - 1]
44         loocv_Xtest = testX_lst[test_subject[0] - 1]
45         loocv_ytest = testy_lst[test_subject[0] - 1]
46         _, weights = train_federated(5, len(fed_Xtrain))
47         accuracy = test_combination_federated(loocv_Xtrain,
   loocv_ytrain, loocv_Xtest, loocv_ytest, dimension, weights
   )
48         save_federated_combination(test_subject, accuracy,
   dimension)
49         federated_accs_combinations[loocv_type][dimension].
   append(accuracy)
50 
51 def train_combinations_single(trainX_lst, trainy_lst,
   testX_lst, testy_lst, subjects_to_ld, dimension, run_type,
    loocv_type):
52 
53     global single_accs_combinations
54     # devo ottenere tutte le possibili combinazioni di coppie
    di soggetti
```

```
55      # conviene lavorare su un array di indici e poi prendere
    il test e il train corrispondente dalle liste.
56      combinations = list(itertools.combinations(subjects_to_ld
    , 2))
57      single_accs_combinations[loocv_type].setdefault(dimension
    , [])
58      # per ogni combinazione devo eseguire il train sul
    secondo e il test sul primo
59      for combination in combinations:
60          trainX = trainX_lst[combination[1] - 1]
61          trainy = trainy_lst[combination[1] - 1]
62          testX = testX_lst[combination[0] - 1]
63          testy = testy_lst[combination[0] - 1]
64
65          accuracy = run_training_single(trainX, trainy, testX,
     testy, dimension, run_type)
66          single_accs_combinations[loocv_type][dimension].
    append(accuracy)
67
68  def train_combinations_centralized(dimension, combinations,
    loocv_type, num_feat):
69
70      centr_accs_combinations[loocv_type].setdefault(dimension,
     [])
71      for combination in combinations:
72          trainX, trainy, testX, testy = load_subjects_dataset(
    combination, "concat", num_feat)
73          print("trainx", trainX.shape)
74          test_subject = find_missing_element(combination, 1,
    31)
75          loocv_Xtrain, loocv_ytrain, loocv_Xtest, loocv_ytest
    = load_subjects_dataset(test_subject, "concat", num_feat)
76
77          # eseguire il train sul dataset concatenato
78          accuracy = run_training_centr(trainX, trainy,
    loocv_Xtest, loocv_ytest, dimension)
```

```
79          centr_accs_combinations[loocv_type][dimension].append
    (accuracy)
```

Listing 4.5: Implementazione test con LOOCV

Come possiamo notare dai metodi riportati sopra, la funzione *train_loocv_combinations* si occupa principalmente di gestire il dataset da utilizzare in base al test da eseguire. Infatti nel caso del test con la LOOCV standard il dataset conterrà tutte le features, mentre nell'altro caso il numero di features varierà in base alla dimensione della SOM. Successivamente nel caso del sistema federato e centralizzato si identifica il soggetto di test, in quanto escluso dalla combinazione, si esegue l'addestramento e poi viene eseguito il test sul dataset del soggetto escluso. Per quanto riguarda il sistema singolo per ogni coppia di soggetti viene eseguito l'addestramento sul secondo e il test sul primo.

4.3.3 Implementazione SOM

Come anticipato la tecnica di Machine Learning utilizzata nell'esperimento è una *Self Organizing Map(SOM)*. Prima di mostrare le variazioni applicate per adattarla ai vari sistemi è utile osservarne una sua implementazione più pura.

```
1      som = MiniSom(
2          n_neurons,
3          m_neurons,
4          trainX.shape[1],
5          sigma=5,
6          learning_rate=0.1,
7          neighborhood_function="gaussian",
8          activation_distance="manhattan",
9      )
10     som.random_weights_init(trainX)
11     som.train_random(trainX, 100, verbose=False)
```

Listing 4.6: Implementazione SOM

Per l'implementazione è stata utilizzata la libreria Python **minisom**[7]. MiniSom è un'implementazione minimalistica e basata su NumPy della SOM. E stata progettata per permettere a ricercatori di costruirci sopra e ' per dare agli studenti di coglierne rapidamente i dettagli. Nonostante sia definita come "minimalistica", questa libreria tuttavia offre diverse configurazioni possibili per la som il che la rende una libreria molto versatile. Per quanto riguarda l'esperimento la configurazione utilizzata è stata la seguente:

- **Sigma**: 5
- **Learning rate**: 0.1
- **Neighborhood function**: gaussian
- **Activation distance**: manhattan
- **Train iterations**: 100

Il codice mostrato riguarda soltanto la parte di training della SOM, infatti per poter eseguire la classificazione è necessario utilizzare una funzione ausiliaria denominata *classify*. Questa funzione per prima cosa calcola il label mapping sui neuroni allenati, ovvero associa a ogni neurone una label, successivamente calcola la BMU sui dati del dataset di train, e in base alle sue coordinate si risale alla label associata da cui è possibile calcolare l'accuracy attraverso il metodo **classification_report** di scikit-learn.

```
1 def classify_fed(som, data, X_train, y_train):
2     """Classifies each sample in data in one of the classes
    definited
3     using the method labels_map.
4     Returns a list of the same length of data where the i-th
    element
5     is the class assigned to data[i].
6     """
7     # winmap contiene una classificazione di campione in
    X_train
8     # con una delle classi in y (associazione neurone-label)
```

```
9
10      winmap = som.labels_map(X_train , new_y_train)
11      default_class = np.sum( list (winmap.values())).
        most_common()[0][0]
12
13      result = []
14      for d in data :
15          win_position = som.winner( d )
16          if win_position in winmap :
17              result.append( winmap [ win_position ].
        most_common()[0][0])
18          else :
19              result.append( default_class )
20      return result
```

Listing 4.7: Implementazione Classify

4.3.4 Implementazione train e test sistema centralizzato e singolo

Per quanto riguarda il sistema centralizzato e singolo l'implementazione di train e test con la SOM è sostanzialmente identica alla sua forma standard.

```
1  def run_training_centr(trainX, trainy, testX, testy, neurons)
       :
2      # eseguiamo il train e testiamo l'accuracy sul test
3      n_neurons = m_neurons = neurons
4      new_y_test = onehot_decoding(testy)
5
6      som = MiniSom(
7          n_neurons,
8          m_neurons,
9          trainX.shape[1],
10         sigma=5,
11         learning_rate=0.1,
12         neighborhood_function="gaussian",
13         activation_distance="manhattan",
14     )
```

```
    som.random_weights_init(trainX)
    som.train_random(trainX, 100, verbose=False)

    class_report = classification_report(new_y_test,
   classify_fed(som, testX, trainX, trainy), zero_division
   =0.0, output_dict=True)
    return class_report["accuracy"]
```

Listing 4.8: Implementazione train e test sistema centralizzato

4.3.5 Implementazione train e test sistema federato

Per quanto riguarda il sistema federato il concetto e l'implementazione base della SOM sono gli stessi ma cambia il modo in cui sono utilizzati. Infatti nel sistema federato ogni client avrà la sua istanza della SOM che verrà inizializzata con i pesi iniziali del modello generalizzato. Successivamente ogni client eseguirà il train sul suo dataset e invierà i pesi della SOM al modello generalizzato del server. Il server aggregherà i pesi dei vari client attraverso la "strategia" scelta(nel nostro caso la media pesata) e aggiornerà il modello generalizzato e quello dei client. Una volta aggiornato il modello dei client, ognuno di essi eseguirà il test sul suo dataset con il modello aggiornato. Il numero di volte per cui viene eseguito questo procedimento è determinato dal numero di round dato in input alla simulazione (dopo un certo numero di round i risultati tendono a stabilizzarsi).

```
def train_federated(num_rounds, num_clients, loocv_type):

    strategy = None

    if loocv_type == "noloocv":
        strategy = fl.server.strategy.FedAvg(
           fraction_fit=1.0,
           fraction_evaluate=1.0,
           min_fit_clients=int(num_clients),
           min_evaluate_clients=int(num_clients),
           min_available_clients=int(num_clients),
```

```
            evaluate_metrics_aggregation_fn=append_accuracies,
        )
    else:
        strategy = fl.server.strategy.FedAvg(
            fraction_fit=1.0,
            fraction_evaluate=1.0,
            min_fit_clients=int(num_clients),
            min_evaluate_clients=int(num_clients),
            min_available_clients=int(num_clients),
            evaluate_metrics_aggregation_fn=simple_average,
        )

    client_resources = {"num_cpus": 1, "num_gpus":1}
    hist = fl.simulation.start_simulation(
        client_fn = client_fn,
        num_clients = int(num_clients),
        config = fl.server.ServerConfig(num_rounds=num_rounds)
    ,
        strategy = strategy,
        client_resources = client_resources,
    )
    #    ray_init_args = {"num_cpus": 2, "num_gpus":0.0}
     return hist.metrics_distributed["accuracy"], hist.
    metrics_distributed["weights"]
```

Listing 4.9: Implementazione train federato

Il sistema federato è stato implementato grazie al framework **Flower**[1], il quale rende possibile la simulazione di un sistema federato in pochissimo tempo e soprattutto su un unico dispositivo. Nel codice mostrato avviene il setup del sistema; qui viene definita come strategia (definisce come gestire i client durante l'addestramento) **FedAvg** che esegue la media pesata dei pesi e gli vengono passati vari parametri che definiscono il numero di client da utilizzare per il train e per il test(nel nostro caso tutti e 30 in entrambi). Inoltre al parametro **evaluate_metrics_aggregation_fn** viene passata una funzione che determina come gestire le accuracy e i vari dati che si vogliono passare da un round all'altro. Nel nostro caso se si è nel test senza

loocv la funzione passata è **append_accuracies** la quale salva le accuracy di ogni round e i pesi finali, mentre se si è nel test con loocv viene passata la funzione **simple_average** che salva la media delle accuracy e i pesi finali. Dopo aver definito la strategia è possibile far partire la simulazione attraverso **fl.simulation.start_simulation** la quale prende in input diversi parametri tra cui la funzione **client_fn** la quale definisce come creare le istanze di ogni client.

```
def client_fn(cid) -> SomClient:
    neurons = current_som_dim
    train_iter = train_iter_lst[0]
    # prendo il dataset corrispondente al cid(client id)
    Xtrain = fed_Xtrain[int(cid)]
    ytrain = fed_ytrain[int(cid)]
    Xtest = fed_Xtest[int(cid)]
    ytest = fed_ytest[int(cid)]

    som = MiniSom(
            neurons,
            neurons,
            Xtrain.shape[1],
            sigma=5,
            learning_rate=0.1,
            neighborhood_function="gaussian",
            activation_distance="manhattan",
        )

    return SomClient(som, Xtrain, ytrain, Xtest, ytest,
   train_iter, cid)
```

Listing 4.10: Implementazione client_fn

Come possiamo vedere dal codice la funzione determina come viene inizializzata ogni istanza della classe **SomClient**, la quale è definita nel seguente modo:

```
class SomClient(fl.client.NumPyClient):
    def __init__(self, som, Xtrain, ytrain, Xtest, ytest ,
   train_iter, cid):
```

```
        self.som = som
        self.Xtrain = Xtrain
        self.ytrain = ytrain
        self.train_iter = train_iter
        self.Xtest = Xtest
        self.ytest = ytest
        self.cid = cid

    # Return the current local model parameters
    def get_parameters(self, config) -> NDArrays:
        return get_parameters(self.som)

    # Receive model parameters from the server,
    # train the model parameters on the local data,
    # and return the (updated) model parameters to the server
    def fit(self, parameters, config):
        set_parameters(self.som, parameters)
        self.som.train_random(self.Xtrain, self.train_iter,
verbose=False)
        return get_parameters(self.som), len(self.Xtrain), {}

    # Receive model parameters from the server,
    # evaluate the model parameters on the local data,
    # and return the evaluation result to the server
    def evaluate(self, parameters, config) -> Tuple[float,
int, Dict[str, Scalar]]:
        new_y_test = list()
        for idx, item in enumerate(self.ytest):
            # inserisco in new_test_y gli index di ogni
classe invertendo il one-hot encode
            new_y_test.append(np.argmax(self.ytest[idx]))
        set_parameters(self.som, parameters)
        class_report = classification_report(
            new_y_test,
            classify_fed(
                self.som,
                self.Xtest,
                self.Xtrain,
```

```
                self.ytrain
            ),
            zero_division=0.0,
            output_dict=True,
        )

        return float(0), len(self.Xtest), {"accuracy": float(
class_report["accuracy"]), "weights": get_parameters(self.
som)[0]}
```

Listing 4.11: Implementazione SomClient

Questa classe definisce ogni client che partecipa al training e i suoi metodi implementano i passaggi principali del sistema federato. In particolare abbiamo:

- **get_parameters**: restituisce i pesi correnti della som.
- **fit**: è la funzione responsabile dell'addestramento della SOM del client. Essa riceve in **parameters** i pesi del modello generalizzato, attraverso **set_parameters** aggiorna i pesi del suo modello locale, esegue l'addestramento del suo modello e alla fine restituisce i pesi aggiornati.
- **evaluate**: è il metodo in cui avviene la classificazione del modello generalizzato sul dataset locale del client. Esso riceve i pesi del modello generalizzato con cui aggiorna il suo modello locale tramite la funzione **set_parameters**, successivamente esegue la classificazione e infine restituisce l'accuracy ottenuta e i pesi del modello.

Capitolo 5

Risultati

Come anticipato nel terzo capitolo l'esperimento si è concluso con un test in grado di mostrare un confronto globale delle varie configurazioni esplorate. In questo capitolo verranno mostrati e commentati i risultati di quel test.

5.1 Risultati test finale

Un boxplot, noto anche come Whisker plot è un tipo di grafico molto utile per mostrare un sommario delle principali proprietà di un set di dati. Esso è in grado di mostrare proprietà come minimo, primo quartile (o 25° percentile), terzo quartile, mediana e massimo. In particolare, ogni box va dal primo quartile al terzo quartile(o 75° percentile), la riga orizzontale all'interno della box rappresenta la mediana e le due tacche agli estremi sono minimo e massimo. Di seguito verranno mostrati i boxplot ottenuti nelle tre configurazioni testate:

- **Test senza LOOCV** in cui nel sistema singolo viene eseguito train e test sullo stesso utente, nel sistema federato e centralizzato si fa il train con 30 soggetti e il test su uno solo. Il tutto viene ripetuto con tutti i 30 soggetti possibili.
- **Test con LOOCV senza variazione feature** simula la situazione reale in cui il soggetto vuole unirsi al sistema. Nel caso singolo ven-

gono considerate tutte le coppie possibili considerando tutti e trenta i soggetti (435 coppie possibili). Per ogni coppia (A, B), si esegue l'addestramento su B e il test su A e alla fine si considera la media delle accuracy. Nel caso centralizzato l'addestramento avviene sui dataset uniti dei 29 soggetti e il test sul trentesimo lasciato fuori (per tutte le combinazioni possibili con i 30 soggetti). Infine nel caso federato, per tutte e trenta le combinazioni possibili, si effettua l'addestramento federato sui 29 client e poi il test sul trentesimo utilizzando il modello generalizzato.

- **Test con LOOCV con variazione feature** simula l'implementazione sui dispositivi embedded, effettuando le stesse operazioni del test precedente ma variando il numero di features utilizzate in base alla dimensione della SOM:
  - SOM size: 10 - features: 265 (tutte le feature)
  - SOM size: 15 - features: 158
  - SOM size: 20 - features: 89
  - SOM size: 30 - features: 39

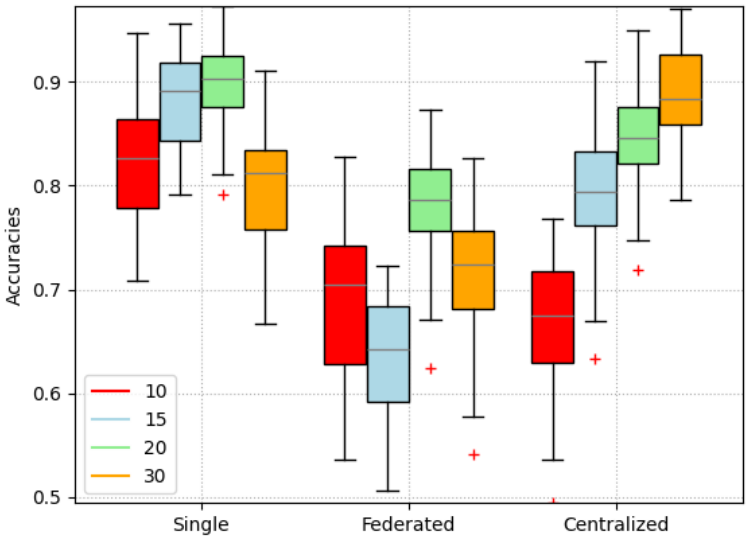


Figura 5.1: Boxplot test senza LOOCV

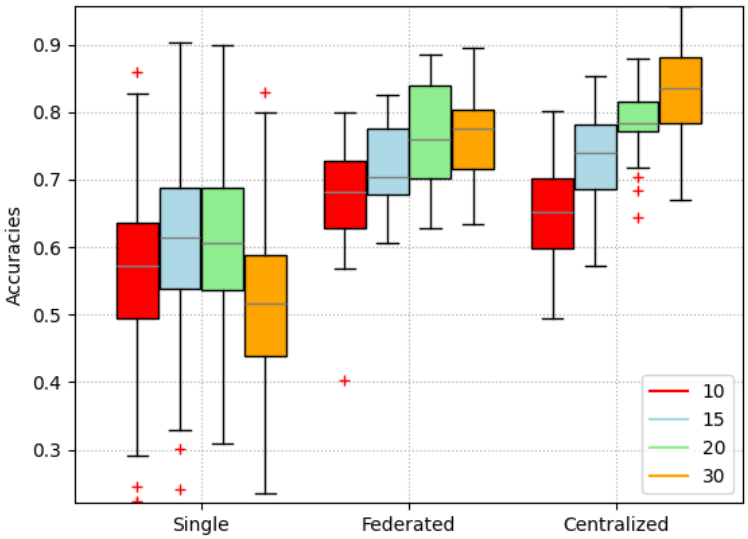


Figura 5.2: Boxplot test con LOOCV senza varaizione features

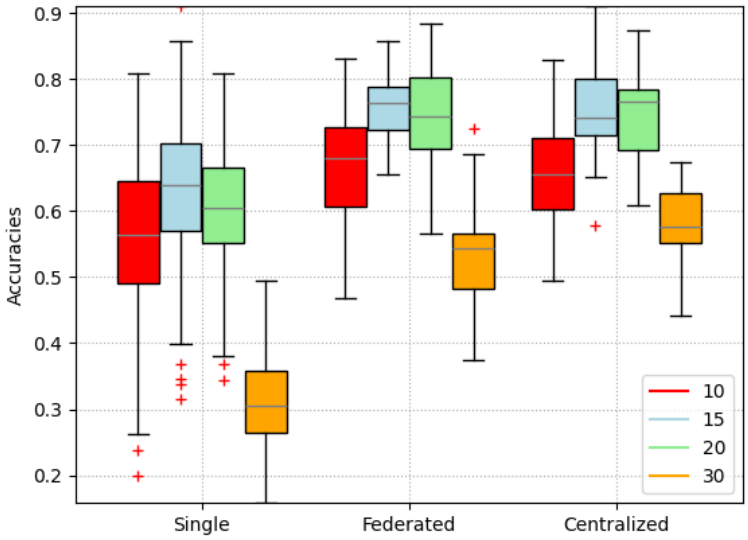


Figura 5.3: Boxplot test con LOOCV con variazione features

Osservando il grafico in 5.1 ci ritroviamo nel caso in cui non viene simulato il soggetto esterno che vuole unirsi al sistema e come anticipato nel terzo capitolo è evidente come il caso singolo funzioni meglio rispetto a centralizzato e federato a causa dell'elevata specializzazione del modello sul dataset. Nel caso in cui viene eseguita la LOOCV la situazione cambia, infatti sia in 5.2 che in 5.3 vediamo che come previsto il caso singolo cala l'accuracy in quanto il modello sarà troppo specializzato su un dataset per funzionare bene su quello di un altro soggetto. A questo punto entrano in gioco il sistema federato e centralizzato che come possiamo vedere hanno risultati simili (entrambi hanno un'accuratezza tra 0,7 e 0,8 e quindi più che soddisfacente) ma bisogna ricordare che il sistema centralizzato presenta il problema della privacy che non è trascurabile.

Conclusioni

Il diffondersi dei dispositivi indossabili ha portato alla proliferazione di casi d'uso che ogni giorno producono grandi set di dati appartenenti a vari ambiti. Questa grande quantità di dati ha spinto lo sviluppo della Human Activity Recognition, un campo di ricerca che riguarda il riconoscimento delle attività compiute da un soggetto basandosi su dati rilevati da dei sensori(giroscopio e accelerometro). Lo sviluppo di questo campo ha portato con se numerose sfide, tra cui la difficoltà nell' implementazione sui dispositivi embedded, a causa delle loro risorse limitate. I numerosi studi hanno presentato le difficoltà poste dall'utilizzo di tecniche di machine learning supervised e lo studio condotto in [6] ha mostrato come la tecnica unsupervised SOM sia in grado di fornire accuracy molto vicine ai metodi supervised principali, ponendosi come un valido modello di classificazione. Tuttavia sui dispositivi embedded il processo di learning della SOM è dispendioso, in quanto composto dalla fase di addestramento e poi da quella di uso. Di conseguenza all interno di questo elaborato si è cercato di creare un modello generalizzato, utilizzabile da nuovi utenti senza dover eseguire il training. Come sistemi sono stati presi in considerazione il sistema centralizzato, quello federato e quello singolo. Il primo consiste nel convogliare tutti i dati in un server su cui poi viene addestrato il modello. Tuttavia questo approccio presenta problemi di privacy in quanto gli utenti devono inviare i loro dati raccolti. In risposta a questo problema entra in gioco il sistema federato, che per ogni client addestra un modello locale e poi crea un modello generalizzato unendo i pesi dei modelli locali. Infine il sistema singolo rappresenta il soggetto che

addestra il modello esclusivamente sul suo dataset. Oltre al test standard, per simulare una situazione reale in cui arriva un soggetto nuovo, tutti i sistemi sono stati testati anche utilizzando la **Leave One Out Cross Validation**. Dopo aver analizzato i risultati ottenuti possiamo concludere che il sistema singolo a livello di accuratezza è il migliore in quanto altamente specializzato sul dataset del soggetto. Tuttavia l'utilizzo esclusivo di questo sistema richiede che ogni soggetto esegua l'addestramento da zero in quanto non esiste un modello da cui partire. Per quanto riguarda la creazione di un modello generalizzato, il sistema federato ha prodotto risultati molto simili e in alcuni casi superiori rispetto a quello centralizzato, con il vantaggio di non presentare problematiche legate alla privacy. Inoltre ha prodotto ottimi risultati anche con le limitazioni imposte dal dispositivo, indicando chiaramente che questa potrebbe essere una via promettente da seguire. Tuttavia questo lavoro non ha ancora raggiunto la sua forma definitiva e ci sono ancora molte cose da testare e eventualmente migliorare come:

- un ulteriore test da effettuare può essere quello di verificare se i valori delle feature tra soggetti diversi si discostano, andando così ad intaccare in modo negativo i risultati e la creazione del modello generalizzato.
- testare varie strategie per il sistema federato cercando quella con risultati migliori.

Bibliografia

[1] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Hei Li Kwing, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[2] G. De Leonardis, S. Rosati, G. Balestra, V. Agostini, E. Panero, L. Gastaldi, and M. Knaflitz. Human activity recognition by wearable sensors : Comparison of different classifiers for real-time applications. In *2018 IEEE International Symposium on Medical Measurements and Applications (MeMeA)*, pages 1–6, 2018.

[3] Oscar D. Lara and Miguel A. Labrador. A survey on human activity recognition using wearable sensors. *IEEE Communications Surveys & Tutorials*, 15(3):1192–1209, 2013.

[4] Reyes-Ortiz, Jorge, Anguita, Davide, Ghio, Alessandro, Oneto, Luca, Parra, and Xavier. Human Activity Recognition Using Smartphones. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C54S4K.

[5] Konstantin Sozinov, Vladimir Vlassov, and Sarunas Girdzijauskas. Human activity recognition using federated learning. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing*

& Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pages 1103–1111, 2018.

[6] Angelo Trotta, Federico Montori, Giacomo Vallasciani, Luciano Bononi, and Marco Di Felice. Optimizing iot-based human activity recognition on extreme edge devices. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 41–48, 2023.

[7] Giuseppe Vettigli. Minisom: minimalistic and numpy-based implementation of the self organizing map, 2018.