UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Debugging di una libreria per spiegazioni controfattuali e suo utilizzo nell'eXplainable Affective Computing

Relatori:

Ing: Antonio Luca Alfeo

Prof: Mario G.C.A. Cimino

Candidato:

Emanuele Respino

ANNO ACCADEMICO 2022/2023

Abstract

L'*emotion recognition* è il settore dell'IA che consente ai sistemi di identificare le emozioni, sviluppandone l'intelligenza emotiva necessaria per muoversi all'interno della società. Molti sistemi sono progettati per lavorare a fianco delle persone: il riconoscimento delle emozioni tramite IA è dunque fondamentale nel processo di integrazione macchina-uomo al fine di renderli più sensibili, empatici e adattabili alle esigenze emotive degli utenti. Questo può essere particolarmente utile in settori come quello medico: sistemi di riconoscimento delle emozioni possono essere utilizzati per monitorare lo stato emotivo dei pazienti, per fornire assistenza e supporto personalizzato, oppure al fine di identificare disturbi mentali, consentendo interventi mirati e tempestivi.

In questi contesti sensibili è doveroso poter individuare il processo decisionale che porta ad una determinata previsione, in modo da verificarne la validità. Il problema di molti modelli è però la loro struttura, poiché questa rende difficile individuare i meccanismi alla base delle decisioni prese, inficiandone l'applicabilità.

Lo studio condotto è basato su K-EmoCon, un dataset multimodale che combina l'annotazione di emozioni da tre punti di vista differenti a misurazioni biometriche.

Il lavoro è stato inizialmente quello di testare alcuni algoritmi di classificazione, al fine di compararne le accuratezze: i dati biometrici, opportunamente processati, sono stati utilizzati dai modelli per effettuare predizioni sulla positività o negatività dell'emozione (*valence* - valenza) provata dal soggetto. Nello specifico, sono stati addestrati modelli basati su *MLPClassifier* e *SVClassifier* i quali hanno reso un'elevata accuratezza nei risultati.

Successivamente, il lavoro si è focalizzato nell'utilizzo di metodi di XAI (*eXplainable Artificial Intelligence*) allo scopo di determinare l'influenza e il contributo di ciascuna feature nelle scelte di classificazione. Nello specifico, il metodo scelto è stato quello del calcolo del controfattuale, nel quale si analizzano le variazioni minime delle features che consentono una differente classificazione del dato. Durante tali analisi, sono state effettuate modifiche alla libreria utilizzata, al fine di adeguarla alla tipologia di classificazione oggetto di studio.

Infine, i risultati ottenuti dai vari modelli sono stati comparati per identificare eventuali similarità nelle scelte decisionali adottate e accrescere la validità dei punti in comune: questo ha fornito informazioni utili ad una migliore comprensione delle componenti principali che influenzano le emozioni percepite durante relazioni sociali.

Indice

Capitolo 1

Introduzione

Negli ultimi anni, l'intelligenza artificiale ha avuto un impatto significativo in molteplici settori della società e dell'industria, contribuendo a trasformare abitudini, lavori e comunicazione. In particolar modo ha avuto un notevole impatto in settori quali la finanza, in cui l'IA è stata impiegata per l'analisi dei dati e l'incremento della sicurezza utente, il settore dei trasporti, per quanto riguarda lo sviluppo di veicoli autonomi e la gestione del traffico, e il settore medico, nel quale l'IA ha consentito una migliore precisione diagnostica, una maggiore efficacia dei trattamenti e un aumento dell'efficienza dell'assistenza sanitaria.

In alcuni campi specifici, effettuare una previsione errata non è un'opzione contemplabile, poiché potrebbe avere gravi ripercussioni sulla salute delle persone: basti pensare ad una manovra errata di un veicolo autonomo o una diagnosi medica sbagliata. Ovviamente però, per quanto i modelli possano essere ben addestrati, non sono perfetti. La questione sempre più stringente è dunque quella di riuscire a comprendere quando una previsione effettuata dal modello può essere valida oppure no. Garantire che le applicazioni di intelligenza artificiale siano accettabili, affidabili e aderenti a standard etici assume quindi un ruolo sempre più critico oggigiorno.

Per quanto però molte tecniche di IA possono restituire risultati accurati, questi algoritmi si comportano come *scatole nere*: la loro complessità impedisce all'utente che vi interagisce di comprendere i motivi che hanno portato ad una determinata scelta, precludendone dunque l'affidabilità.

In questo contesto svolge un ruolo fondamentale l'**eXplainable Artificial Intelligence**. La XAI è un programma di ricerca che si concentra nel rendere comprensibili e chiari i processi decisionali dei modelli di IA, proponendo metodi che tentano di spiegarne le caratteristiche interne.

Ad oggi, i metodi proposti dalla XAI sono molteplici, ma potrebbero essere raggruppati in due categorie principali:

- *Interpretabilità percepita.* Metodi che forniscono informazioni che possono essere percepite umanamente in modo *ovvio*, attraverso approcci audio, verbali

e/o visivi. In questa tipologia di tecniche, l'attenzione è posta nel rendere i risultati più semplici e trasparenti senza però inficiarne la validità.

- *Interpretabilità matematica.* Metodi che sfruttano le strutture matematiche per rivelare i meccanismi degli algoritmi di apprendimento automatico. In questo caso, i risultati prodotti sono più complessi e meno intuitivi, richiedendo dunque una maggiore conoscenza specifica dei metodi da parte dell'utente finale.

L'Intelligenza Artificiale spiegabile svolge dunque un ruolo fondamentale in vari contesti, specialmente in quello sanitario, in cui le decisioni prese dall'IA, a partire dai dati fisiologici misurati, devono essere giustificate pazienti, medici e altri soggetti interessati. I metodi utilizzati devono essere dunque chiari e di facile interpretazione, in modo da dare la possibilità anche a chi non è specializzato di comprendere le motivazioni dietro un risultato ottenuto[13].

In questo contesto, la ricerca svolta si è focalizzata sulla branca dell'intelligenza artificiale atta allo studio e al riconoscimento delle emozioni, l'**Affective Computing**. Lo sviluppo dell'intelligenza emotiva è estremamente utile in un'ampia varietà di contesti e può portare numerosi benefici sociali, sanitari ed economici, migliorando le interazioni umane, il benessere mentale e le esperienze utente:

- *Assistenza sanitaria e benessere.* Il riconoscimento delle emozioni può essere utilizzato per monitorare lo stato emotivo dei pazienti durante le terapie o le visite mediche. Può aiutare a identificare segnali precoci di stress, depressione o altri disturbi emotivi, consentendo un intervento precoce e una migliore assistenza.

- *E-learning e formazione.* Nei contesti educativi, il riconoscimento delle emozioni può essere utilizzato per valutare l'efficacia dell'apprendimento e personalizzarne l'esperienza in base allo stato emotivo degli studenti. Può anche aiutare gli insegnanti a fornire feedback più mirati e supporto personalizzato agli studenti.

- *Tecnologia indossabile e dispositivi smart.* L'integrazione del riconoscimento delle emozioni nei dispositivi indossabili o nei dispositivi smart può consentire un'interazione più intuitiva e personalizzata con gli utenti. Ad esempio, uno smartwatch potrebbe riconoscere lo stress dell'utente e suggerire tecniche di rilassamento o pause attive.

- *Assistenza ai clienti e marketing.* Le aziende possono utilizzare il riconoscimento delle emozioni per valutare la soddisfazione dei clienti e adattare le loro strategie di conseguenza. Ad esempio, un sistema di chatbot potrebbe riconoscere la frustrazione di un cliente e trasferire la conversazione a un operatore umano per un'assistenza più personalizzata.

- *Intrattenimento e gaming.* Nei settori dell'intrattenimento e del gaming, il riconoscimento delle emozioni può essere utilizzato per creare esperienze più coinvolgenti e immersive. Ad esempio, un videogioco potrebbe adattare la difficoltà del gioco in base al livello di eccitazione o coinvolgimento dell'utente.

In questi contesti dunque le tecniche di XAI possono svolgere un ruolo cruciale nell'integrazione dei dati fisiologici nei modelli di intelligenza artificiale, garantendo trasparenza, interpretabilità e fiducia nei confronti dei sistemi basati su tali dati.

Una delle sfide per il riconoscimento delle emozioni infatti risiede nella loro misurazione e definizione: le espressioni facciali sono spesso fuorvianti, soprattutto se rilevate in ambienti controllati, poiché non consentono ai soggetti di esprimere in maniera naturale le proprie emozioni. I dati possono essere inoltre vincolati a bias a seconda dell'insieme di soggetti scelti, oppure possono non offrire misurazioni fisiologiche del soggetto, né assicurare ambienti e conversazioni realistiche se basati su contenuti multimediali.

Lo studio condotto utilizza come dati i risultati di una ricerca svolta in Corea del Sud, la quale ha prodotto K-EmoCon, un dataset multimodale. Questo è basato su emozioni suscitate durante conversazioni spontanee e include dati rilevati da tutti e tre i punti di vista disponibili: soggetto, partner di dibattito e osservatore esterno. Inoltre, combina a questi dati misurazioni biometriche e audio-visive.

Nel lavoro effettuato, vengono esplorati due differenti algoritmi di intelligenza artificiale, nel contesto del riconoscimento delle emozioni, a partire da dati biometrici, analizzando accuratezza e spiegabilità dei modelli generati. Infatti, i vantaggi derivanti dalla XAI in questo ambito sono molteplici, consentendo di comprendere come i sistemi di IA riconoscono le emozioni umane e fornendo giustificazioni per le previsioni effettuate.

Nello specifico, i due algoritmi confrontati sono stati il *Multi-Layer Perceptron Classifier* (MLPC), basato su reti neurali, e il *Support Vector Classifier* (SVC), basato sulla suddivisione dello spazio in iperpiani: utilizzando approcci differenti, presentano ciascuno i propri vantaggi e svantaggi.

Una volta generati i modelli, si è tentato di spiegarne e confrontare i processi decisionali utilizzando il metodo XAI che consiste nella *counterfactual explanation*: a partire dal dato originale, è stato calcolato e analizzato il controfattuale, ricavandone le features più importanti per il relativo modello di analisi e potendo così confrontare tra loro i risultati ottenuti.

Il confronto e l'analisi di differenti modelli e dei loro processi decisionali tramite XAI è fondamentale al fine non solo di verificare gli algoritmi e i parametri più adatti ad affrontare un determinato problema, ma soprattutto allo scopo di rafforzare l'affidabilità delle previsioni effettuate consultando le motivazioni che l'hanno prodotta. Questi sono aspetti cruciali in contesti medico-sanitari, come quello del riconoscimento delle emozioni, in quanto consentono di fornire al medico le motivazioni per le diagnosi effettuate.

Capitolo 2

Related Works

Questo capitolo offre una panoramica sulle ricerche correlate al campo di studio, offrendo il contesto a partire dal quale l'analisi si sviluppa. In particolare, saranno messe in luce problematiche legate al riconoscimento delle emozioni, proponendo le motivazioni che hanno spinto alla ricerca di un approccio differente, che tenti di contribuire positivamente al contesto in cui si pone.

2.1 Rischi dell'Affective Computing

Come precedentemente introdotto, il campo dell'*Affective Computing* si focalizza nello sviluppo di modelli di machine learning capaci di riconoscere, interpretare e rispondere alle emozioni umane [15].

L'utilizzo delle IA in questo campo porta però con sé vari rischi, legati sia alla duttilità delle teorie psicologiche che tentano di definire le emozioni, che ai problemi tecnologici da affrontare e agli interrogativi etici e giuridici del loro utilizzo [1][4][7]:

- **Accuratezza delle previsioni**. Riconoscere e interpretare stati emotivi complessi in modo più accurato e affidabile è uno dei problemi nella ricerca dell'Affective Computing: gli algoritmi di machine learning richiedono grandi quantità di dati etichettati per riconoscere ed interpretare le emozioni con precisione, ma i dati raccolti sono spesso non sufficientemente ampi e diversificati.

- **Violazione dei diritti**. L'utilizzo di un'ampia mole di dati multimodali può d'altra parte condurre a problemi di privacy, in quanto i dati raccolti possono portare alla identificazione del soggetto coinvolto.

- **Preconcetti e bias**. Non esiste uno schema standard per l'etichettatura delle emozioni, poiché queste vengono interpretate in modo diverso da culture e individui diversi. Questo può complicare il confronto dei risultati tra studi e lo sviluppo di modelli di riconoscimento delle emozioni efficaci in popolazioni diverse.

- **Modelli non adattivi**. È necessario sviluppare un modello di riconoscimento delle emozioni più personalizzato e adattivo, in grado di tenere conto delle differenze individuali e dei cambiamenti negli stati emotivi nel tempo. Ciò richiede di apprendere dai modelli emotivi individuali degli utenti e sviluppare modelli che si adattino a tali pattern emotivi.

- **Modelli come *scatole nere***. Un altro problema riguarda lo sviluppo di modelli di machine learning più robusti e interpretabili, specialmente in ambito di Affective Computing. I modelli di machine learning utilizzati in questo campo, come le reti neurali, sono spesso considerati *scatole nere*, il che rende difficile comprendere il loro funzionamento interno e ottimizzarli per applicazioni specifiche. Questo problema preclude la fruibilità del sistema anche da parte di utenti non specializzati nel settore delle IA e rende il modello inutilizzabile in tutti quei contesti, ad esempio il settore sanitario, in cui è fondamentale visionare le motivazioni dietro i processi decisionali effettuati, al fine di comprovarne l'affidabilità. Ciò rende l'utilizzo di sistemi di IA, come quello in questione, di non facile utilizzo e meno applicabile nei diversi settori.

2.2 Il problema della spiegabilità del modello

Con particolare riguardo alla spiegabilità dei modelli, è già stata introdotta l'Intelligenza Artificiale Spiegabile, la quale punta alla comprensione delle scelte decisionali che il modello prende durante le proprie previsioni[14], attraverso l'analisi delle features che maggiormente le caratterizzano. Le tecniche sviluppate in questo campo sono molteplici, ma molte presentano comunque limitazioni importanti che non le rendono adatte ad essere applicate nel campo dell'Affective Computing.

2.2.1 Modelli SHAP e LIME

Ad oggi, i metodi SHAP[6] e LIME[12] sono i più utilizzati nel campo della spiegabilità dei modelli:

- **SHapley Addictive exPlanation**. SHAP è una tecnica di spiegazione dei modelli di machine learning che utilizza la teoria dei giochi per attribuire importanza alle variabili in base al loro contributo nella predizione, attraverso il calcolo dei valori di Shapley. Esso fornisce una spiegazione locale delle previsioni del modello, consentendo di comprendere il ruolo di ciascuna variabile ai fini predittivi.

- **Local Interpretable Model-agnostic Explanations**. LIME è una tecnica di spiegazione dei modelli di machine learning che mira a spiegare singole predizioni. Esso genera un modello interpretabile localmente intorno alla predizione di interesse e utilizza questo per spiegare l'importanza locale delle features per il modello di machine learning principale. Inoltre, è indipendente dallo specifico algoritmo di IA analizzato.

Nonostante i diversi vantaggi che questi metodi offrono, essi presentano alcuni punti critici. Difatti, possono richiedere risorse computazionali significative, specialmente su modelli complessi o dataset di grandi dimensioni, in quanto di complessità esponenziale rispetto al numero di features. Inoltre, non consentono di ottenere informazioni aggiuntive sulla specifica istanza trattata.

2.2.2 Counterfactual explanations

I metodi di **counterfactual explanation** consentono di dare una spiegazione di quanto eventuali modifiche del dato analizzato potrebbero inficiare nella classificazione dello stesso. Nello specifico, mostrano quali sono i cambiamenti minimi necessari ad una istanza perché la sua classificazione cambi. Questo approccio consente, dunque, di ottenere spiegazioni maggiormente interpretabili sulla singola istanza, che tentano di rispondere alla domanda: *Cosa sarebbe successo se le circostanze fossero state diverse?* Queste spiegazioni risultano utili in vari contesti, al fine inoltre di consentire l'individuazione di bias o discriminazioni durante il processo decisionale[3].

Anche in questo caso, nonostante l'approccio semplice ed intuitivo che si tenta di implementare, la maggior parte di questi modelli presenta diverse problematiche. Difatti, benché i metodi proposti siano numerosi, molti di questi presentano svariati problemi: mancano di effettiva implementazione, presentano codice poco documentato, mostrano errori, necessitano di requisiti estremamente stringenti, non consentono il calcolo del controfattuale per molte istanze di dato.

2.3 Limitazioni correnti e contributo

L'approccio che viene proposto e analizzato in questo studio tenta di minimizzare e/o risolvere le questioni sollevate precedentemente:

- Tramite l'analisi e il confronto di differenti algoritmi di IA viene ricercato un modello che consenta prestazioni elevate e risultati accurati, nel contesto dell'Affective Computing.

- Attraverso l'Intelligenza Artificiale Spiegabile si tenta di studiare e identificare quali sono le caratteristiche principali che guidano i modelli adottati durante i processi decisionali, consentendo di rendere il sistema trasparente e interpretabile. Inoltre, l'analisi delle features più importanti potrebbe portare a osservazioni su quali tipologie di dato concentrare l'attenzione. Questo permetterebbe di minimizzare la tipologia di dati raccolti, consentendo di realizzare un approccio meno invasivo senza inficiare nelle prestazioni del sistema.

- Il metodo XAI preso in studio è il **CounterFactual Nearest Optimal Wololo (CFNOW)**, che permette il calcolo delle counterfactual explanations in modo rapido e semplice, consentendo una copertura quasi totale delle istanze

di dato per cui genera un risultato. Tramite l'analisi effettuata sarà possibile verificare l'utilità del metodo nel contesto dell'Affective Computing.

Dal punto di vista applicativo, tutto questo si tradurrebbe in un sistema efficiente, accurato e semplice per il riconoscimento delle emozioni. Ciò ne permetterebbe l'integrazione anche nei contesti più sensibili come quello sanitario, consentendo anche a chi non esperto del settore IA di interfacciarvisi.

Capitolo 3

Design e Implementazione

Sulla base di ciò che è stato evidenziato nelle pagine precedenti, in questo capitolo viene proposta la soluzione adottata e ne viene descritta la sua implementazione, indicando le esigenze a cui questa potrebbe andare a rispondere.

3.1 Design

In questa sezione viene fornita un'approfondita illustrazione dell'approccio utilizzato durante le varie fasi di analisi, facendo chiarezza in particolar modo sulla tipologia di modelli considerati e i metodi di XAI utilizzati.

3.1.1 Classificazione

La classificazione, nell'ambito del machine learning, è un tipo di problema in cui l'obiettivo è assegnare a ciascun elemento di un insieme di dati una delle diverse classi predefinite attraverso un algoritmo di apprendimento automatico. Una volta che il modello è stato addestrato su un training set, può essere utilizzato per effettuare previsioni sui nuove istanze.

Nel caso del dataset analizzato, i dati sono basati su rilevazioni biometriche suddivise in 18 features: *x*, *y*, *z*, *E4_BVP*, *E4_EDA*, *E4_HR*, *E4_IBI*, *E4_TEMP*, *Attention*, *delta*, *lowAlpha*, *highAlpha*, *lowBeta*, *highBeta*, *lowGamma*, *middleGamma*, *theta* e *Meditation*. Le classi da assegnare sono invece i valori da 1 a 5 di *self_valence*, che specificano quanto è positiva l'emozione provata, dal punto di vista del soggetto che la percepisce.

Per risolvere determinati problemi, esistono diversi algoritmi di classificazione. Lo studio effettuato si è concentrato sull'utilizzo di due di essi, di seguito spiegati.

Modelli utilizzati

La ricerca di algoritmi adatti alla gestione del problema analizzato ha condotto al confronto dei seguenti modelli:

- **Multi-Layer Perceptron Classifier**. Algoritmo di classificazione basato su rete neurale artificiale, la quale è composta da più livelli di neuroni (nodi), disposti in modo che ciascuno di essi sia connesso a tutti quelli dello strato inferiore e superiore. Il risultato è un grafo con un layer di neuroni in input, uno in output, e un certo numero di livelli intermedi che consentono, tramite anche a meccanismi di retropropagazione dell'errore, di adattarsi ai dati gestiti e migliorare le proprie prestazioni nel tempo[9].

- **Support Vector Classifier**. Algoritmo di classificazione in cui l'obiettivo è trovare l'iperpiano ottimale che separa i dati in classi differenti nello spazio delle features. Tale iperpiano è scelto in modo da massimizzare la robustezza del modello e minimizzare l'overfitting, massimizzando il margine tra i punti di dati più vicini di classi diverse. L'algoritmo consente di lavorare con dati lineari e non, eventualmente proiettandoli in uno spazio di dimensione maggiore in cui diventano linearmente separabili[2].

Tecniche di pre-processing

Le tecniche di pre-processing sono utilizzate per preparare i dati prima di utilizzarli nell'addestramento di modelli. Queste tecniche sono fondamentali per garantire che i dati siano nella forma più adatta, consentendo di migliorare le prestazioni complessive del modello ottenuto.

Nello specifico, durante l'analisi sono state confrontate tre tecniche di scaling, che consentono di trasformare le caratteristiche dei dati in un range e/o una distribuzione specifici, al fine di valutarne similitudini e differenze:

- **MinMaxScaler**. Scaler utile per ridimensionare i dati in modo che siano compresi tra un intervallo specifico, di solito 0 e 1. Questo può essere utile per algoritmi che richiedono che i dati siano in un intervallo specifico, come le reti neurali.

- **StandardScaler**. Scaler utilizzato per standardizzare i dati in modo che abbiano una media di zero e una deviazione standard unitaria, garantendo che i dati abbiano una distribuzione normale. Anche questo può essere utile per algoritmi come le reti neurali.

- **RobustScaler**. Scaler che viene utilizzato per ridimensionare i dati in modo robusto rispetto agli outliers, ovvero i valori che si discostano significativamente dalla maggioranza dei dati.

Valutazione dei modelli

Al fine di determinare l'accuratezza ottenuta al variare del modello, degli iperparametri scelti e del metodo di pre-processing utilizzato, sono stati presi in considerazione differenti metodologie e metriche di valutazione:

- **Cross-validation**. Tecnica utilizzata nella valutazione delle prestazioni di un modello, che consiste nell'andare a partizionare il training set in un certo numero di *fold* (di default 5), andando poi ad utilizzarli iterativamente per addestrare e valutare il modello mediante calcolo del **Mean-test score**. Ciò consente di fornire una valutazione più affidabile delle prestazioni del modello. Questa tecnica è implementata nella prima fase di analisi, durante la scelta dei migliori iper-parametri del modello.

- **Accuracy score**. Metrica scelta per valutare l'accuratezza dei modelli, spiegata in dettaglio successivamente.

3.1.2 Analisi della spiegabilità

Dopo aver addestrato e valutato i modelli, l'analisi si è concentrata nello studio della spiegabilità dei modelli tramite metodi di *eXplainable Artificial Intelligence*. Nello specifico, è stata posta l'attenzione sulla tecnica della *counterfactual explanation*.

Counterfactual explanation

La spiegazione controfattuale è una delle tecniche tramite le quali si tenta di spiegare il processo decisionale dei modelli addestrati. In particolare, spiega la decisione che il modello prende per una singola istanza di dato, mostrando quali sono i cambiamenti minimi necessari da applicare all'istanza in modo da modificarne la classificazione. Per minimo cambiamento si intende che ogni sottoinsieme di modifiche, rispetto a quelle scelte, non consente al modello di cambiare la classificazione dell'istanza.

Nello specifico, la libreria utilizzata (*cfnow* [3]) consente di generare controfattuali a partire da differenti tipologie di dato in maniera rapida. Inoltre, è di semplice utilizzo, in quanto è sufficiente specificare l'istanza del dato e la funzione di probabilità del modello adottato, e permette di generare controfattuali per la quasi totalità delle istanze.

Il processo svolto dalla libreria si suddivide in due fasi: si convertono prima i possibili differenti tipi di dato (tabulare, immagini o testo) in un singolo formato che successivamente viene utilizzato per la ricerca del controfattuale. Questa seconda fase a sua volta adotta una strategia in due step:

1. **CF search**. Questa fase ha l'obiettivo di trovare velocemente una soluzione, ovvero un set di modifiche che consentono all'istanza di cambiare classificazione, che siano ottime oppure no. In questo caso, le strategie possibili sono due:

   - *Greedy*. Approccio sbilanciato verso modifiche sequenziali che meglio potrebbero incrementare lo *score*, cioè la probabilità che l'istanza sia classificata in una differente classe rispetto a quella iniziale.

   - *Random*. Approccio che non segue percorsi specifici e che dunque potrebbe rendere migliori risultati nel caso di modelli non lineari.

2. **CF improvement**. Si tenta di ottimizzare la soluzione precedentemente trovata minimizzando una funzione obiettivo come la distanza tra istanza iniziale e controfattuale. In particolare, viene utilizzata la **Manhattan distance**, ovvero la somma delle differenze assolute tra i valori di ciascuna feature. Questa fase di ottimizzazione potrebbe proseguire indefinitamente, per cui è importante impostare un adeguato limite di tempo per il processo.

Una volta calcolato il controfattuale, *cfnow* si appoggia ad un'altra libreria *counterplot* per consentire di visualizzarne graficamente i risultati. Vengono offerte tre tipologie di diagrammi, ciascuno dei quali ne evidenzia un differente aspetto:

- **Greedy**. Questo diagramma consente di visualizzare una determinata sequenza di cambiamenti nei valori delle features che portano dall'istanza di partenza al controfattuale. La sequenza è individuata in maniera greedy, scegliendo di volta in volta la feature che maggiormente aumenta lo **score**.

- **CounterShapley**. Mostra l'importanza relativa che ha ciascuna feature nella ricerca del controfattuale, mediante il calcolo dei *valori di Shapley*. Non tiene dunque conto di una specifica sequenza di cambiamenti, ma di tutte le possibili combinazioni.

- **Constellation**. Diagramma simile al greedy, ma che consente di visualizzare tutte le possibili sequenze di modifiche e lo score associato a ciascuna di esse.

Comparazione dei risultati

Per i modelli selezionati sono stati calcolati i controfattuali per un certo numero di istanze di dato, dopodiché ne è stata fatta un'analisi prima qualitativa e poi quantitativa.

Per l'**analisi qualitativa** sono stati generati i diagrammi greedy e countershapley e ne è stata effettuata la comparazione visiva, verificando coerenze e differenze tra le diverse tipologie sia di grafico che di modello.

Durante l'**analisi quantitativa** invece sono state individuate computazionalmente le features coinvolte nel calcolo del controfattuale e poi sono stati confrontati i set risultanti da ciascun modello, per ogni istanza di dato selezionata, tramite il **Jaccard Index**.

Attraverso queste analisi è stato possibile individuare le feature che maggiormente incidono nei processi decisionali di ciascun modello, potendo inoltre osservare similitudini e differenze tra i sistemi trattati.

3.2 Impostazione sperimentale

Precedentemente sono state citate alcune metriche con le quali è stato deciso di impostare la sperimentazione. Sono state scelte l'*accuracy score* e il *Jaccard index*, ma durante le analisi è stato fatto uso anche di altre definizioni di *score* e *distanza*.

3.2.1 Mean-test score

Metrica utilizzata per valutare le prestazioni del modello durante la fase di ottimizzazione dei parametri nell'ambito della *cross-validation*. Indica la media delle valutazioni ottenute dal modello su differenti partizioni dei dati di set, fornendo dunque una misura della capacità del modello di generalizzare bene su dati non visti durante l'addestramento.

Un mean-test score più alto indica che il modello ha ottenuto buone prestazioni su dati di test, mentre un mean-test score più basso potrebbe indicare problemi di overfitting.

3.2.2 Accuracy score

L'accuracy score è una metrica utilizzata per valutare le prestazioni dei modelli addestrati. Consiste nel calcolare il rapporto tra le previsioni corrette e il numero di valutazioni totali:

$$Accuracy = \frac{\#\,predizioni\,corrette}{\#\,predizioni\,totali}$$

Il valore può variare nel range [0, 1]. Una precisione più alta indica che il modello è in grado di fare previsioni più accurate, mentre una precisione più bassa indica che il modello fa più errori: 0 se nessuna previsione è risultata corretta, 1 se tutte le previsioni sono risultate esatte.

L'obiettivo è dunque quello di massimizzarne il valore durante la generazione e l'addestramento dei modelli di classificazione.

3.2.3 Manhattan distance

La Manhattan distance è una delle metriche utilizzate per il calcolo della distanza tra due punti in uno spazio a più dimensioni:

$$d = \sum_{i=1}^{n} |p_i - q_i|$$

Dove:

- p_i e q_i sono le coordinate rispettivamente dei punti p e q lungo l'asse i.
- n è il numero di dimensioni dello spazio.

Il risultato è un valore nel range $[0, +\infty)$. Un valore maggiore della distanza indica che i punti sono più distanti: è pari a 0 se e solo se i due punti coincidono.

Il processo di ottimizzazione del controfattuale tenta dunque di minimizzare tale metrica, in modo da ricavare una soluzione più vicina possibile a quella originale.

3.2.4 Jaccard index

Il Jaccard Index è una metrica che consente di calcolare il grado di similarità tra due insiemi come il rapporto tra la loro intersezione e la loro unione:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Il valore può variare nel range [0, 1]. Un valore più alto indica un maggiore grado di coesione tra gli insiemi, un valore più basso invece indica un minor grado di sovrapposizione: il valore è pari a 0 se i due insiemi sono completamente differenti, 1 se coincidono.

L'indice può essere dunque utilizzato per valutare il grado di similarità dei modelli, consentendo di individuare eventuali punti in comune tra i diversi processi di classificazione.

3.2.5 Factual e counterfactual score

I grafici utilizzati per visualizzare le counterfactual explanations basano la loro interpretabilità sulla definizione di *score*. In questo contesto dunque, lo score di una determinata istanza di dato è definito come la probabilità che l'istanza venga classificata dal modello decisionale con la stella classe del controfattuale precedentemente calcolato. Di conseguenza, possiamo distinguere:

- **Factual score**: la probabilità che l'istanza di dato iniziale ricada nella stessa classe del controfattuale. Ovviamente, poiché per definizione il controfattuale appartiene ad una differente classe del fattuale, questo valore sarà minore del counterfactual score. Valori comunque elevati potrebbero indicare che l'istanza di dato si trova già vicino al confine decisionale che divide la propria classe da quella del controfattuale.

- **Counterfactual score**: la probabilità che il controfattuale calcolato ricada nella propria classe predetta. In questo caso, lo score sarà più elevato in quanto tale probabilità deve risultare maggiore di ciascuna delle altre probabilità di ricadere in una classe. Valori comunque bassi possono indicare una buona ottimizzazione della soluzione calcolata, anche se non è escluso il contrario.

- **Score intermedi**: la probabilità che le istanze intermedie, date dalla modifica dei valori nelle features dell'istanza di partenza, ricadano nella classe del controfattuale. Questi valori saranno maggiori del factual score e minori del counterfactual score, in quanto ciascuna modifica addizionale viene inserita allo scopo di aumentare lo score dell'istanza.

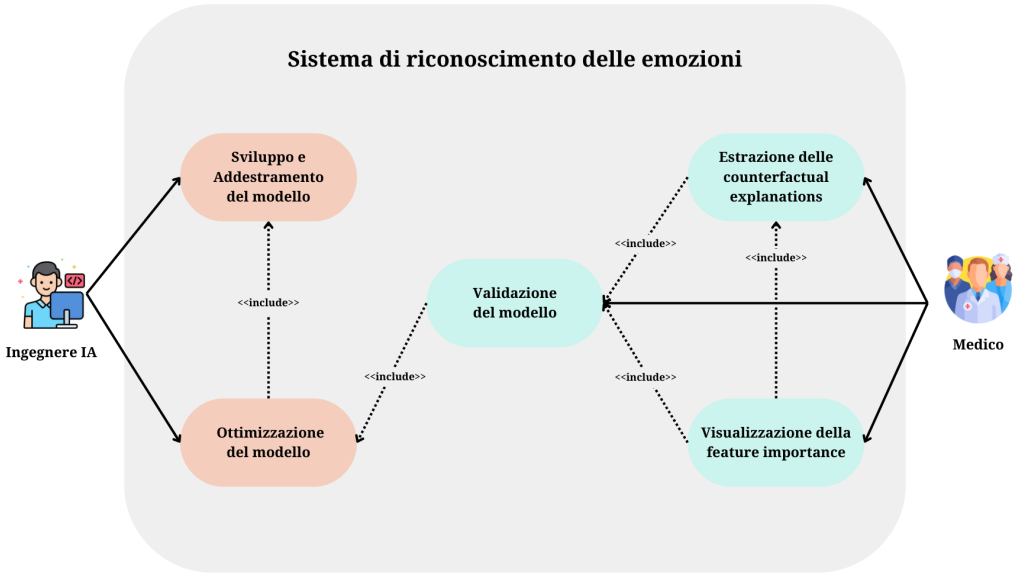


Figura 3.1: Caso d'uso in ambito sanitario

3.3 Implementazione

Nello studio svolto, il software prodotto basa le proprie previsioni su dati biometrici, rilevati mediante sensori applicati al soggetto. Dunque, il principale settore in cui questo sistema può trovare maggiore applicazione è l'ambito sanitario.

3.3.1 Use case

Un possibile caso d'uso è mostrato in Figura 3.1. In questo contesto sono presenti due attori principali:

- **Ingegnere IA**. Esso è l'attore responsabile dello sviluppo e dell'addestramento del sistema, collaborando con il medico nell'ottimizzazione di esso. Nello specifico, si occupa di:
  - *Sviluppare e addestrare il modello*. L'ingegnere ha il compito di sviluppare e addestrare un modello basato su algoritmi di machine learning, come MLPClassifier o SVClassifier, per il riconoscimento delle emozioni.
  - *Ottimizzare il modello*. Tramite feedback sulla validazione del modello ottenuti osservando i risultati, l'ingegnere ha inoltre il compito di ottimizzare il sistema creato.
- **Medico**. Il medico è l'attore principale, in quanto colui che usufruisce del prodotto software finale. Nel sistema proposto, questo si occupa di:

- *Visualizzare le features importance.* Il medico analizza i risultati prodotti dal sistema, visualizzando quali sono le caratteristiche principali che influiscono nelle scelte di classificazione e comprendendone le motivazioni su cui si fonda.
- *Estrarre le counterfactual explanation.* Il medico può anche andare a consultare più approfonditamente ciò che influisce nei processi decisionali del modello, analizzando i diagrammi relativi alle spiegazioni controfattuali legate alle singole istanze di dato, su cui sono basate le features importance.
- *Validare il modello.* Le osservazioni che il medico svolge consultando le caratteristiche principali e i controfattuali gli consentono di poter validare il modello utilizzato: verificando l'affidabilità delle previsioni ottenute genera importanti feedback utili all'ottimizzazione del sistema.

Da sottolineare come la XAI consente ad un soggetto non specializzato in IA (il medico) di utilizzare un sistema di intelligenza artificiale avanzato come quello per il riconoscimento delle emozioni.

Capitolo 4

Case Study

4.1 Dataset

[10] Una delle sfide per il riconoscimento delle emozioni risiede nella loro misurazione: al contrario del pensiero comune, le espressioni facciali sono spesso fuorvianti, soprattutto se rilevate in ambienti controllati, poiché non consentono ai soggetti di esprimere in maniera naturale le proprie emozioni, risultando spesso in espressioni prototipiche che scaturiscono raramente in natura. I dati possono essere inoltre vincolati a bias se l'insieme di soggetti scelti non è abbastanza eterogeneo o non sufficientemente ampio. Questo tipo di problema può essere in parte risolto utilizzando dataset basati su una vasta gamma di contenuti multimediali, come serie TV e film, i quali però non offrono misurazioni fisiologiche del soggetto, né assicurano ambienti e conversazioni che rispecchino la realtà.

In questo contesto nasce K-EmoCon, il primo dataset multimodale, basato su emozioni suscitate durante conversazioni spontanee, che include dati rilevati da tutti e tre i punti di vista disponibili: soggetto, partner di dibattito e osservatore esterno. Inoltre, combina a questi misurazioni biometriche e audio-visive.

4.1.1 Approccio utilizzato

Il dataset è stato progettato sulla base di uno scenario in cui due persone, indossando sistemi non intrusivi per la rilevazione di segnali fisiologici, interagiscono durante una conversazione.

Analisi multi-prospettica

La scelta di effettuare un'analisi multi-prospettica è basata su precedenti ricerche, le quali hanno riscontrato come l'utilizzo di più fonti per rilevare le emozioni incrementi l'accuratezza nel riconoscerle. Questo perché le emozioni sono fenomeni interni, i cui meccanismi non sono identificabili né dall'esterno, né dallo stesso soggetto che ne fa esperienza. Punti di vista differenti possono concordare su emozioni forti, ma queste sono rare in natura: è più probabile, invece, che queste siano in disaccordo

poiché il soggetto le maschera oppure perché ne risulta difficile l'interpretazione. Nello specifico, le parti coinvolte sono:

1. Soggetto: colui che fa esperienza dell'emozione e indica cosa ha provato;
2. Partner: la persona che interagisce con il soggetto ed è coinvolta nel contesto che induce un'emozione.
3. Osservatori esterni: persone che osservano il soggetto senza avere piena conoscenza del contesto che induce l'emozione.

Tipologia di conversazione

La conversazione è realizzata attraverso un dibattito su un problema sociale (la crisi dei rifugiati Yemeniti a Jeju), in cui sono presenti due soggetti casualmente accoppiati: questa particolare impostazione permette di simulare in modo appropriato una conversazione che potrebbe generarsi sul luogo di lavoro, attendendosi che i soggetti regolino le proprie emozioni in maniera appropriata, sulla base del contesto spontaneo e formale nel quale la conversazione si sviluppa.

Utilizzo dei dispositivi

I dispositivi per la rilevazione di segnali fisiologici sono stati utilizzati al fine di permettere lo studio delle discrepanze nella percezione delle emozioni ed esaminare l'accuratezza nel riconoscere quelle espresse con minore intensità.

4.1.2 Procedura dei dati raccolti

L'esperimento ha visto la partecipazione di 32 studenti del KAIST, i quali sono stati suddivisi casualmente in coppie e a cui è stato richiesto di conversare in inglese su di un dibattito in precedenza assegnato loro.

Ciascuna sessione di raccolta dati si è sviluppa in quattro fasi ed è stata amministrata da due sperimentatori, svolgendosi in due stanze in condizioni controllate, in cui i due partecipanti erano seduti ad un tavolo l'uno di fronte l'altro. Ciascuno dei due soggetti ha indossato i sensori necessari alle rilevazioni biometriche e audiovisive (POV in prima persona). Erano inoltre presenti sul tavolo due dispositivi per le riprese in seconda persona.

Preparazione

I soggetti hanno letto e firmato il consenso obbligatorio a partecipare alla raccolta dati e il consenso opzionale alla diffusione di dati contenenti informazioni di identificazione personale. Dopodiché hanno scelto se dibattere in favore o contro l'argomento di conversazione ed è stato deciso il primo interlocutore. A questo punto, dopo essere stato fornito materiale aggiuntivo per la preparazione del dibattito, sono stati equipaggiati dei dispositivi per i rilevamenti biometrici e audio-visivi.

Misurazioni di riferimento

Per circa due minuti sono stati presi parametri fisiologici di base mentre i partecipanti osservavano la clip Color Bars, poiché come osservato da uno studio precedente, questa clip induce un'emozione neutrale. Questa fase ha permesso inoltre di testare il corretto funzionamento dei dispositivi prima dell'inizio del dibattito.

Dibattito

Ciascun dibattito ha avuto una durata di circa dieci minuti, durante i quali espressioni facciali, movimenti della parte superiore del corpo e conversazione di ogni partecipante sono stati registrati. Ciascuno di loro è potuto intervenire durante il turno dell'altro in modo da permettere una conversazione più naturale.

Annotazione delle emozioni

Dopo una pausa di 15 minuti, ciascun partecipante ha esaminato la propria registrazione audio-visiva in seconda persona, seguita da quella del partner, annotando le emozioni percepite ogni 5 secondi dall'inizio alla fine del dibattito. Anche 5 valutatori esterni sono stati reclutati per annotare le emozioni dei partecipanti, attraverso la stessa modalità appena descritta.

L'utilizzo del *retrospective affect judgment protocol* è particolarmente indicato in studi come questo, nei quali l'attenzione del soggetto durante il processo che induce le emozioni è essenziale, come lo è il mantenere un colloquio naturale con il partner. L'utilizzo del punto di vista in seconda persona invece di quello in prima persona deriva da studi precedenti, i quali hanno riscontrato che la rilevazione di emozioni nel secondo caso è maggiormente soggetta a bias e distorsione.

4.1.3 Struttura del dataset

Il dataset include misurazioni dei segnali fisiologici dei partecipanti, ricavati tramite dispositivi indossabili, registrazioni audio-visive e annotazioni continue delle emozioni provenienti dai tre differenti punti di vista precedentemente specificati. Di seguito sono riportate le caratteristiche principali dei dati raccolti.

Preprocessing

Per la sincronizzazione temporale dei dati, tutti i timestamp sono stati convertiti a UTC +0, mentre la raccolta dei dati ricavata è stata tagliata in modo da mantenere solo le parti relative al dibattito e alle misurazioni di riferimento. Per quanto riguarda gli elementi audio-visivi, le clip relative alla conversazione sono state estratte. Le tracce audio dei partecipanti durante il dibattito sono state copiate separatamente in file WAV. I segnali fisiologici sono stati memorizzati fino al termine del dibattito, dunque i primi 1.5-2 minuti sono relativi alle misurazioni di riferimento in uno stato neutrale.

Numero di partecipanti	32 (20 uomini e 12 donne)
Età dei partecipanti	Dai 19 ai 36 anni (media = 23.8y, stdev = 3.3y)
Durata della sessione	Totale 172.92 min (media = 10.8 min, stdev = 1.04 min)
Categorie delle emozioni annotate	1-5: Arousal, Valence 1-4: Cheerful, Happy, Angry, Nervous, Sad Scelta singola: Common BROMP affective categories + less common BROMP affective categories
Segnali fisiologici misurati	3-axis Acc. (32Hz), BVP (64Hz), EDA (4Hz), heart rate (1Hz), IBI (n/a), body temperature (4Hz), EEG (8 band, 32Hz), ECG (1Hz)

Tabella 4.1: Sommario sulla raccolta dati

Conversazioni audio	172.92 min (da 16 conversazioni)
Filmati delle conversazioni	223.35 min (da 21 partecipanti)
Segnali fisiologici	Specificati di seguito
Annotazione delle emozioni (# annotazioni ogni 5 secondi)	Self: 4,159 Partner: 4,159 5 osservatori esterni: 20,803

Tabella 4.2: Contenuto del dataset

Contenuto

Il dataset è suddiviso in varie cartelle e file che comprendono dati e metadati relativi allo studio effettuato. Nello specifico, oltre alle registrazioni audio-visive dei partecipanti durante le conversazioni, sono inclusi i seguenti segnali fisiologici:

1. Attention: misura del livello di attenzione del partecipante, nel range [1; 100]. Il valore 0 indica che il dispositivo non ha calcolato un valore affidabile. Un valore maggiore indica un livello maggiore di attenzione.
2. BrainWave: misura, tramite EEG, della potenza relativa delle onde cerebrali in 8 bande:
   (a) Delta (0.5-2.75 Hz);
   (b) Theta (3.5-6.75 Hz);
   (c) Low-Alpha (7.5-9.25 Hz);
   (d) High-Alpha (10-11.75 Hz);
   (e) Low-Beta (13-16.75 Hz);
   (f) High-Beta (18-29.75 Hz);
   (g) Low-Gamma (31-39.75 Hz);
   (h) Middle-Gamma (41-49.75 Hz).
3. Meditation: misura il livello di rilassamento del partecipante, nel range [0; 100]. Un valore maggiore indica un livello di rilassamento maggiore.
4. Polar_HR: frequenza del battito cardiaco misurata tramite ECG.
5. E4_ACC: misurazioni dell'accelerometro a 3 assi del bracciale indossato dal partecipante, campionato a 32Hz nel range [-2g; 2g]. Le colonne sono x, y, z.
6. E4_BVP: Impulso di volume di sangue, registrato tramite PPG, campionato a 64Hz.
7. E4_EDA: Attività elettrodermica, campionato a 4Hz in µS.
8. E4_HR: frequenza cardiaca media, calcolata in una finestra di 10 secondi, per cui valori relativi ai primi 10 secondi non sono presenti. Calcolato alla frequenza di 1Hz sulla base del BVP.
9. E4_IBI: intervallo temporale tra battiti cardiaci successivi, calcolato a partire dal BVP.
10. E4_TEMP: temperatura corporea del partecipante, campionato a 4Hz in gradi Celsius.

Il dataset contiene inoltre le annotazioni delle emozioni per ogni partecipante, da ciascuno dei tre punti di vista. Nello specifico, relativamente alle annotazioni degli osservatori esterni, sono presenti sia le annotazioni di ciascun osservatore, sia il valore aggregato per maggioranza.

Validazione dei dati

La distribuzione e frequenza delle emozioni annotate dimostra come siano fortemente tendenti ad essere rilevate come neutre, in accordo con ciò che si verifica in natura. Riguardo i segnali fisiologici invece alcuni dati sono mancanti a causa di problemi del dispositivo o errori umani.

Limitazioni

La raccolta dati è soggetta a diverse limitazioni, quali:

- I sensori EEG sono molto suscettibili a rumore, anche il solo movimento degli occhi potrebbe causare picchi nelle misurazioni;
- La modalità di conversazione potrebbe aver spinto alcuni partecipanti a regolare e mitigare le proprie emozioni rispetto ad una normale conversazione;
- Il *retrospective affect judgment protocol* potrebbe aver introdotto involontari effetti di autovalutazione delle emozioni, anche se questo può essere in parte compensato dal giudizio multi-prospettico applicato nello studio;
- L'impronta demografica dei partecipanti potrebbe aver introdotto alcuni bias, in quanto risultano giovani, con un'educazione di alto livello e in maggioranza di etnia asiatica;
- Diverse variabili non sono state prese in considerazione, come il livello di inglese dei partecipanti e la loro familiarità con l'argomento della conversazione.

4.2 Librerie utilizzate

Il codice è stato interamente implementato in Python, sfruttando varie librerie sia per la creazione e l'addestramento dei modelli, sia per il calcolo e l'analisi dei controfattuali.

Qui di seguito sono elencate le principali librerie utilizzate e le relative finalità per le quali sono state impiegate:

- **scikit-learn**. Libreria fondamentale, che tramite i propri moduli consente di usufruire facilmente di tutte le funzioni e le classi necessarie a pre-processare i dati, implementare e addestrare algoritmi di classificazione e analizzare le prestazioni dei modelli generati. Nello specifico, sono state utilizzate:

  - *train_test_split*. Funzione utilizzata per suddividere un dataset in *training set*, utilizzato per addestrare il modello, e *test set*, utilizzato per valutarne le prestazioni. Di default, il 75% del dataset ricade nel training set, il rimanente 25% nel test set.
  - *MLPClassifier*. Classe che consente di implementare l'algoritmo di classificazione *Multi-Layer Perceptron* basato su reti neurali. Può essere addestrato mediante training e test sets.
  - *SVC*. Classe che consente di implementare l'algoritmo di classificazione *Support Vector Classifier* basato sulla costruzione di un iperpiano ottimale per la suddivisione dello spazio delle features. Anch'esso può essere addestrato tramite train e test sets.
  - *RobustScaler*, *StandardScaler*, *MinMaxScaler*. Classi che consentono la trasformazione e la standardizzazione dei dati prima di applicare gli algoritmi di IA e di analisi.
  - *GridSearchCV*. Classe che implementa un metodo esaustivo di ottimizzazione degli iper-parametri, consentendo di trovare la combinazione che massimizza le prestazioni del modello.
  - *accuracy_score*. Funzione che implementa una metrica di valutazione delle prestazioni del modello di classificazione ottenuto, in cui si calcola la percentuale delle previsioni corrette rispetto al totale.
  - *confusion_matrix*. Funzione che consente di visualizzare il numero di predizioni corrette ed erronee effettuate dal modello fornendo una valutazione più dettagliata delle previsioni del modello.

- **cfnow**[3]. Libreria che implementa un metodo rapido e in due step per il calcolo del controfattuale, il quale può essere applicato a partire da differenti tipologie di dato. Consente inoltre di visualizzarne i risultati appoggiandosi ad un'ulteriore libreria, **counterplots**[8], la quale permette di generare i grafici delle *counterfactual explanation* al fine di analizzarne le features più importanti. Nello specifico, sono state utilizzate:

  - *find_tabular*. Funzione che consente, a partire da un dato di tipo tabulare e una funzione di probabilità, di calcolare una *counterfactual explanation*. Come parametro di ritorno, restituisce l'oggetto *_CFTabular*.
  - *_CFTabular*. Classe contenente vari attributi, tra cui il miglior controfattuale calcolato e la funzione *generate_counterplots*. Questa si aggancia alla libreria grafica, consentendo la generazione di 3 tipologie di grafici per la visualizzazione delle features più importanti: *greedy*, *countershapley* e *constellation*.

  Per adattare la libreria al dataset utilizzato e consentire la corretta visualizzazione dei grafici, si sono rese necessarie alcune modifiche al codice che tali librerie implementano.

- **pandas**. Libreria che mette a disposizione strutture dati flessibili e potenti per gestione di grandi quantità di dati (*DataFrame* e *Series*), oltre che a utili funzioni per la lettura e scrittura di file in formato tabulare (*read_ cvs* e *to_ csv*).

- **numpy**. Libreria per il calcolo scientifico che consente di svolgere operazioni su array multidimensionali rapidamente e in modo efficiente.

- **matplotlib**. Libreria ampiamente utilizzata per la creazione di grafici di vario tipo, che consente di visualizzare al meglio i risultati ottenuti dalle analisi svolte.

- **seaborn**. Libreria, a sua volta basata su *matplotlib*, utilizzata per la visualizzazione di alcuni grafici più complessi in modo semplice ed efficiente.

Capitolo 5

Debugging della libreria CFNOW

La libreria *cfnow* è stata utilizzata per la visualizzazione delle counterfactual explanations tramite i grafici greedy, countershapley e constellation.

In una fase iniziale dell'analisi, sia durante visualizzazione di questi grafici che in fase di esecuzione del codice, sono state riscontrate anomalie nei risultati ottenuti. Ciò ha portato alla decisione di analizzare più a fondo l'implementazione della libreria, al fine di accertarsi della presenza di eventuali problemi.

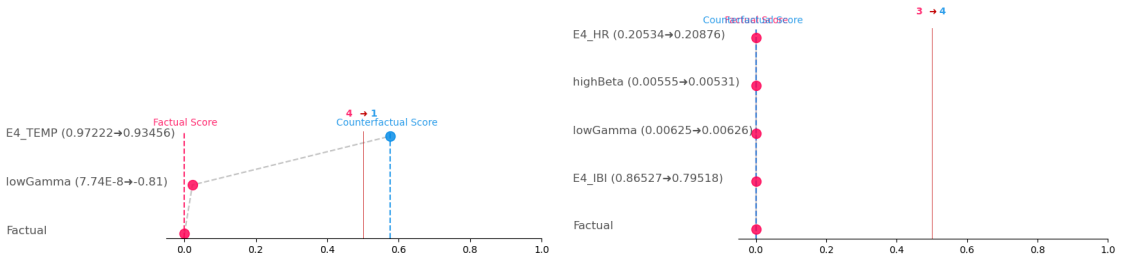
In questo capitolo vengono descritte le modifiche e le considerazioni effettuate che hanno portato alla correzione della libreria, rendendola applicabile al dataset analizzato e consentendo il proseguo dell'analisi.

5.1 Analisi del problema

Inizialmente è stato ipotizzato che il problema potesse essere dovuto alla classificazione errata del dato da parte del modello, ma poi è stato osservato che si estendeva alla maggioranza dei dati considerati, fatto in contrasto con l'elevata accuratezza del modello utilizzato.

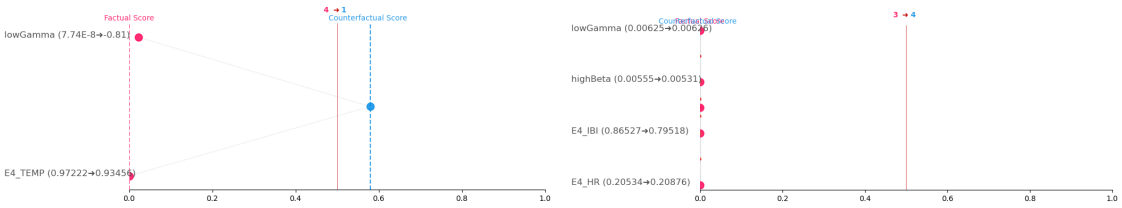
Osservando il diagramma Greedy (Figura 5.1), ma la stessa considerazione vale anche per Constellation (Figura 5.2), la successione dei punti visualizzati dovrebbe indicare il progressivo aumento dello *score*, invece questi risultano tutti allineati allo *score* iniziale. Ciò dovrebbe voler dire che nessuna di queste feature dà contributo al calcolo del controfattuale, di conseguenza non dovrebbe essere stata modificata e non dovrebbe comparire nel grafico. Invece, la variazione viene correttamente segnalata alla sinistra del grafico stesso.

Osservando il diagramma CounterShapley (Figura 5.3), si osservano percentuali di contributo delle feature inverosimili se non impossibili, come valori negativi o che oltrepassano il 100%.



(a) Sample n. 3 (b) Sample n. 21816

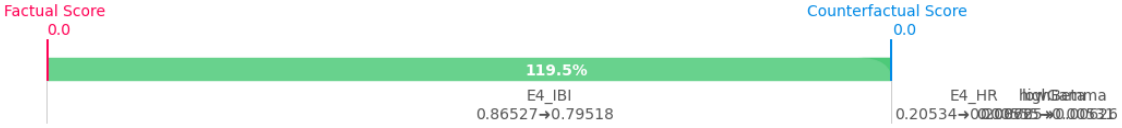
Figura 5.1: Risultati iniziali del diagramma greedy.



(a) Sample n. 3 (b) Sample n. 21816

Figura 5.2: Risultati iniziali della constellation.

(a) Sample n. 3



(b) Sample n. 21816

Figura 5.3: Risultati iniziali del countershapley.

5.1.1 Problema riscontrato

Il problema è stato riscontrato essere il fatto che la libreria ***counterplots*** non permette la visualizzazione di dati riguardanti classi non binarie, mentre la libreria *cfnow* permette la gestione anche di questa tipologia di classi.

Le anomalie sorgono perché *cfnow* sfrutta *counterplots* per la creazione dei diagrammi: per "aggirare" la limitazione di *counterplots*, *cfnow* semplicemente "tronca", al primo elemento a priori, la funzione di probabilità del modello utilizzato, quando viene passato come parametro per la classe ***CreatePlot*** di *counterplots*. Questo comporta che, se né l'istanza di partenza né il controfattuale ricadono nella prima classe, i diagrammi si baseranno su probabilità sbagliate e sempre basse, risultando dunque nei grafici visualizzati inizialmente.

Il problema, dunque, si presenta se si utilizza *cfnow* con classi non binarie, mentre non si presenta altrimenti.

Le modifiche sono state apportate in modo da consentire l'utilizzo dell'intera funzione di probabilità del modello, cercando di apportare il minor numero di modifiche alle strutture preesistenti. Da questo punto di vista, la funzione ***create_adapted_model_and_class()*** è stata comunque modificata del tutto, anche se tentando di mantenere la struttura dell'implementazione preesistente.

5.2 Modifiche effettuate

Analizzando la libreria, sono stati riscontrati alcuni problemi e sono dunque state apportate modifiche al codice della libreria in modo da poter gestire correttamente il dataset in oggetto. Di seguito sono riportate le modifiche effettuate.

5.2.1 Metodo generate_counterplots()

Metodo della classe *_CFTabular* del modulo *cf_finder.py* appartenente alla libreria ***cfnow***.

```
1         return cpl.CreatePlot(
2             factual=np.array(self.factual.tolist()),
3             cf=np.array(self.cfs[idx_cf].tolist()),
4             feature_names=self.col_names,
5             # model_pred=lambda x: self.model_pred[:, 0]
6             model_pred=self.model_pred
7         )
8
```

Listing 5.1: Modifiche apportate al metodo generate_counterplots().

Nel codice 5.1 stato modificato il parametro *model_pred* passato alla funzione *cpl.CreatePlot()*. Commentato, è presente il parametro passato inizialmente.

5.2.2 Costruttore di CreatePlot

La classe è presente nel modulo *__init__.py* appartenente alla libreria ***counterplots***.

```
        # If feature_names is not provided, create it
        if feature_names is None:
            feature_names = [f'f{i+1}' for i in range(len(factual))]

        # NEW: Find factual and CF class indexes
        factual_index = np.argmax(model_pred(factual.reshape(1, -1)))
        cf_index = np.argmax(model_pred(cf.reshape(1, -1)))
        class_indexes = {0: factual_index, 1: cf_index}

        # If class_names is not provided, create it
        if class_names is None:
            # class_names = {0: '0', 1: '1'}
            class_names = {0: str(class_indexes[0] + 1),
                           1: str(class_indexes[1] + 1)}

```

Listing 5.2: Modifica di variabili preesistenti e inizializzazione di nuove nel costruttore di CreatePlot.

Come si osserva nel codice 5.2, nel costruttore viene aggiunta la variabile *class_indexes* che specifica gli indici della classe non binaria a cui appartengono l'istanza iniziale e il controfattuale. Inoltre, viene modificata l'inizializzazione di default della variabile *class_names*, in concordanza con la nuova variabile (commentato, è presente l'inizializzazione originale).

```
        # Make verifications
        verify_factual_counterfactual_shape(factual, cf)
        verify_factual_cf(factual, cf)
        verify_feature_names(factual, feature_names)
        verify_model_prediction(model_pred, factual)
        # verify_binary_class(model_pred, factual)
        verify_class_names(class_names)

```

Listing 5.3: Eliminazione della verifica di classi binarie.

Successivamente è stata commentata la verifica di classi binarie, la quale avrebbe altrimenti sollevato errore (Codice 5.3).

```
        # Adapt model and class names
        self.adapted_model, self.class_names, self.class_indexes = create_adapted_model_and_class(
            factual=factual,
            model_pred=model_pred,
            class_names=class_names,
            class_indexes=class_indexes)

```

Listing 5.4: Chiamata alla funzione create_adapted_model_and_class().

Infine, essendo poi modificata anche la funzione *create_adapted_model_and_class()*, è stato inserito il parametro *class_indexes* nella chiamata di funzione ed è stata aggiunta la variabile di classe *self.class_indexes* come terzo valore di ritorno (codice 5.4).

5.2.3 Funzione create_adapted_model_and_class()

Funzione presente nel modulo *process.py* appartenente alla libreria ***counterplots***.

```
def create_adapted_model_and_class(
        factual,
        model_pred,
        class_names,
        class_indexes):

    pred_test = np.array(model_pred(np.array([factual])))
    pred_test = np.array([pred_test[0][class_indexes[0]], pred_test[0][class_indexes[1]]])

    def adapted_model(x):
        pred = np.array(model_pred(x))
        return pred[:, class_indexes[1]] if pred_test[1] < 0.5 else pred[:, class_indexes[0]]

    adapted_class = {0: class_names[0], 1: class_names[1]} if pred_test[1] < 0.5 \
        else {0: class_names[1], 1: class_names[0]}

    adapted_indexes = {0: class_indexes[0], 1: class_indexes[1]} if pred_test[1] < 0.5 \
        else {0: class_indexes[1], 1: class_indexes[0]}

    return adapted_model, adapted_class, adapted_indexes

```

Listing 5.5: Nuova implementazione di create_adapted_model_and_class().

Per questione di semplicità, tutto il corpo originale della funzione è stato commentato e non è qui riportato. Nella modifica effettuata (Codice 5.5) si cerca però di mantenere la stessa struttura di partenza.

5.2.4 Funzione make_countershapley_plot()

Funzione presente nel modulo *plots.py* appartenente alla libreria ***counterplots***.

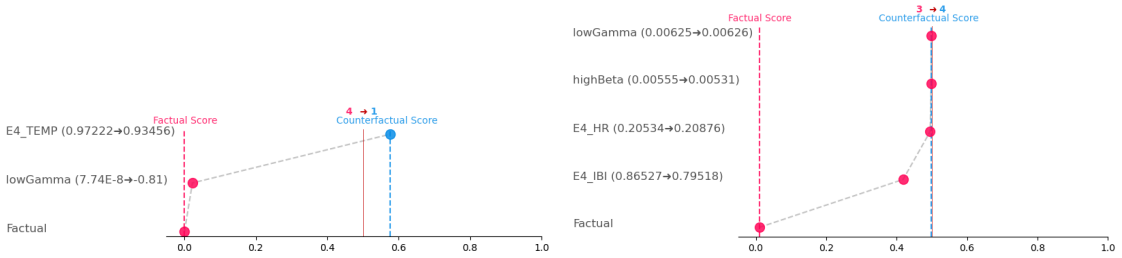
```
        # Draw bar for the factual score
    plt.bar(0, scale_y - 10, width=0.5, color='#ff0055', linewidth=1)
    plt.text(-5, scale_y + fontsize * 4,
             f'Factual Score [{classes[0]}]', color='#ff0055', fontsize=fontsize)
    plt.text(0, scale_y, factual_score, color='#ff0055',
             fontsize=fontsize)

    # Plot bar for the counterfactual score
    plt.bar(current_x, scale_y - 10, width=0.5, color='#008ae7', linewidth=1)
    plt.text(current_x - 10, scale_y + fontsize * 4,
             f'Counterfactual Score [{classes[1]}]', color='#008ae7', fontsize=fontsize)

```

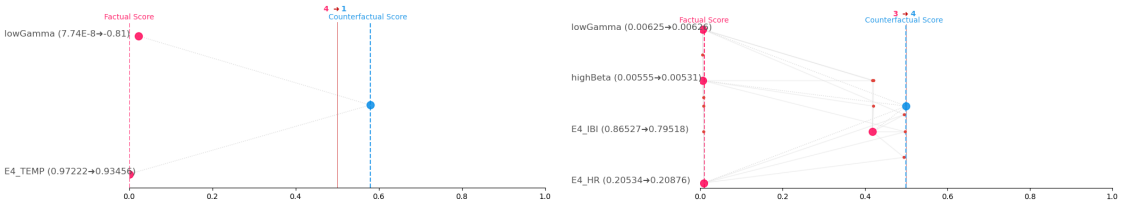
Listing 5.6: Aggiunta delle classi coinvolte nel grafico generato

Modifica estetica per la visualizzazione del diagramma, dunque non fondamentale ai fini del corretto funzionamento della libreria. La funzione riceve come parametro (*classes*) i nomi delle classi coinvolte, ma nel corpo della funzione non viene utilizzato. La funzione è stata modificata in modo da poter visualizzare anche il nome delle classi di fattuale e controfattuale. Nel codice 5.6 sono riportati i due blocchi di codice in cui viene fatto uso della variabile, una volta apportata la modifica.



(a) Sample n. 3 (b) Sample n. 21816

Figura 5.4: Risultati iniziali del diagramma greedy.



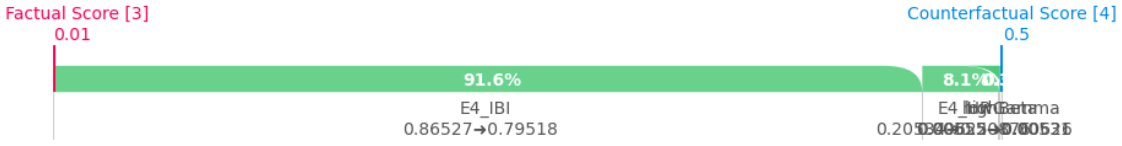
(a) Sample n. 3 (b) Sample n. 21816

Figura 5.5: Risultati iniziali della constellation.

5.3 Validazione delle modifiche

I diagrammi generati, successivamente alle modifiche svolte, sono risultati questa volta coerenti con le aspettative (Figure 5.4, 5.5, 5.6). Nello specifico, si può notare come non ci sia stato un cambiamento per i grafici già inizialmente visualizzati correttamente: questo perché quelli sono i campioni per cui le istanze iniziali o i controfattuali calcolati ricadono nella prima classe e dunque il programma non risente del problema.

(a) Sample n. 3



(b) Sample n. 21816

Figura 5.6: Risultati iniziali del countershapley.

5.4 Limitazioni

5.4.1 Osservazioni sui grafici

In alcuni diagrammi greedy (Figure B.39, B.10, B.45, B.54) è possibile notare che il *cf score* non supera mai il valore di 0.5, non varcando la soglia rossa. Non si tratta però di un errore, ma di una conseguenza delle modifiche effettuate alla libreria. In caso di classi binarie, unica eventualità consentita inizialmente dalla libreria *counterplot*, l'unico modo che ha il dato per essere classificato allo stesso modo del controfattuale è che *cf score* aumenti oltre il valore di 0.5. Il contesto in cui siamo però prevede una classe non binaria e, di conseguenza, il controfattuale ammette uno score inferiore, a patto che sia comunque superiore a tutte le probabilità delle altre classi.

5.4.2 Funzionalità implementata

La funzionalità inserita in 5.5 non è stata aggiunta alle altre ma è andata a sostituirle, pur tentando di mantenere la struttura originale con cui la funzione è implementata. Questa scelta è stata ritenuta opportuna date le limitazioni di tempo, ma consente comunque di soddisfare a pieno le necessità dell'analisi affrontata.

Una migliore implementazione potrebbe andare invece ad aggiungere tale funzionalità a quelle già preesistenti, ampliando ulteriormente l'insieme dei problemi che la libreria può affrontare.

Capitolo 6

Risultati sperimentali

Nel seguente capitolo sono presentati i risultati sperimentali ottenuti attraverso l'implementazione e l'esecuzione delle metodologie descritte precedentemente. L'obiettivo principale di questa fase è valutare le prestazioni e l'efficacia delle tecniche proposte.

I risultati qui presentati forniscono una valutazione dei modelli di machine learning analizzati, sia dal punto di vista dell'accuratezza che della spiegabilità, evidenziando le problematiche emerse durante le analisi e le osservazioni emerse dall'analisi.

6.1 Qualità dei dati analizzati

In primo luogo, è stata condotta un'analisi preliminare dei dati, con l'obiettivo primario di comprenderne la natura e le caratteristiche prima di qualsiasi manipolazione o elaborazione. Questa fase è molto utile per il successivo processo di pre-processing dei dati: capire la distribuzione delle variabili, la presenza di outliers e le eventuali relazioni tra le caratteristiche dei dati è importante per guidare le decisioni sulle le tecniche di pre-processing da applicare.

Nello specifico, sono stati visualizzati:

- Dimensioni del dataset, eventuale presenza di valori nulli o duplicati;
- Bar plot dell'etichetta *self_valence*;
- Istogramma e box plot delle features biometriche;
- Plot temporale delle misurazioni dei segnali cerebrali.

In questo modo è stato constatato come il dataset sia privo di dati mancanti o duplicati. È stato verificato, inoltre, che le variabili presentano scale dimensionali differenti (Figura 6.1). Per quanto riguarda nello specifico il valore dei segnali cerebrali, invece, è possibile constatare la presenza di alcuni picchi, che potrebbero essere associati ad outliers probabilmente dovuti all'alta sensibilità dei sensori utilizzati, suscettibili anche al semplice battito delle ciglia (Figura 6.2). È possibile

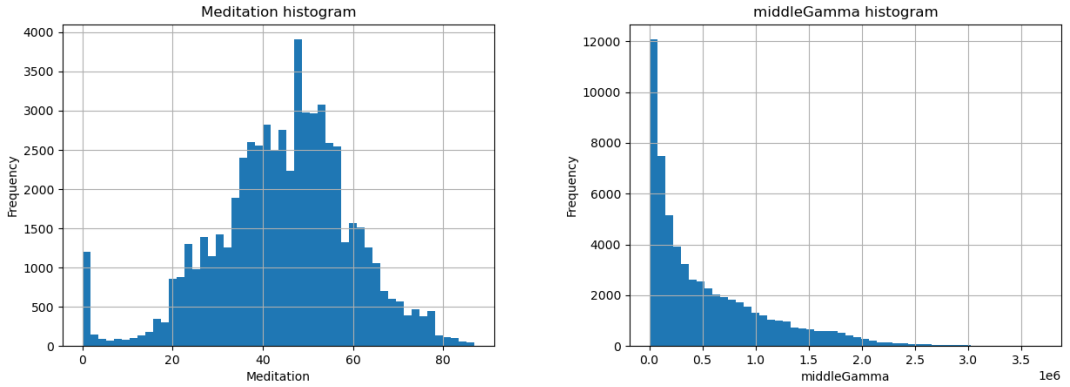


Figura 6.1: Le differenti scale di valore tra due features lungo l'asse orizzontale.

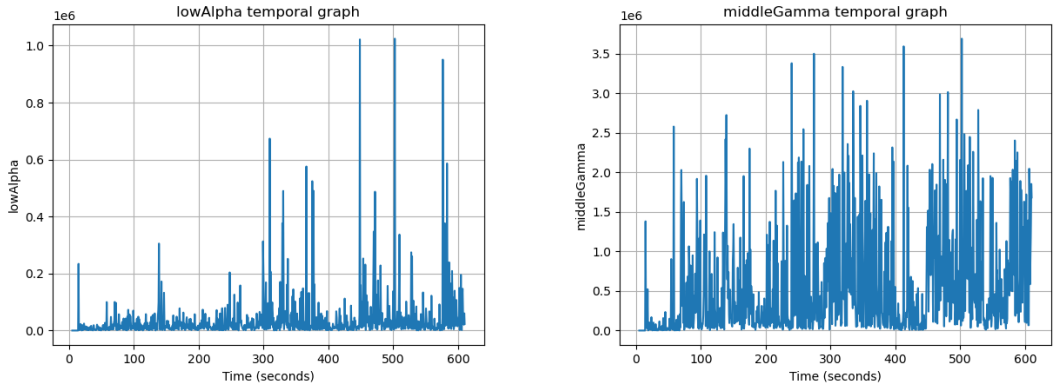


Figura 6.2: Grafici temporali di alcune bande celebrali.

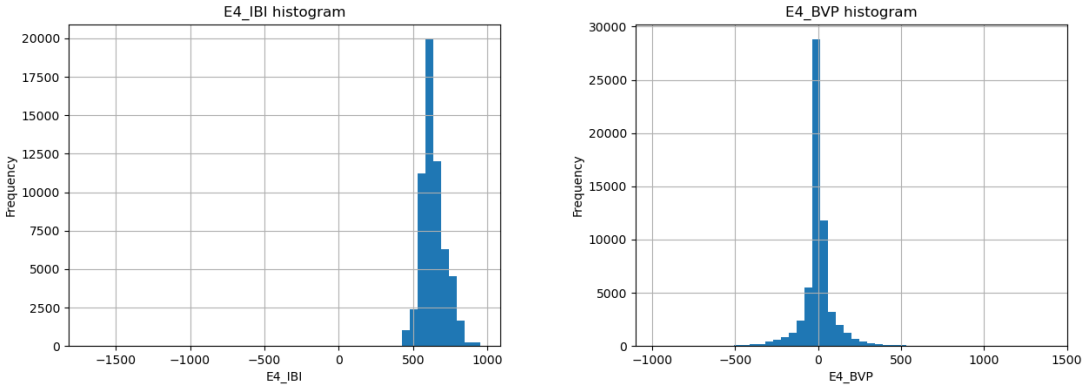


Figura 6.3: Due variabili skewed.

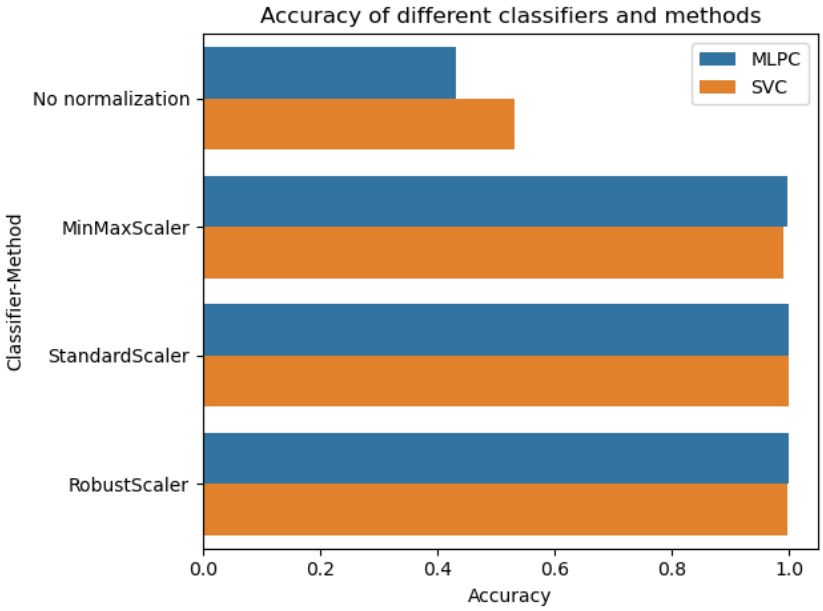


Figura 6.4: Accuracy media per i differenti modelli addestrati.

	No pre-processing	*MinMaxScaler*	*StandardScaler*	*RobustScaler*
MLPC	0.433	0.996	**0.999**	**0.999**
SVC	0.531	0.991	**0.999**	0.997

Tabella 6.1: Valori di accuracy media dei modelli addestrati.

notare un altro problema che potrebbe inficiare nelle prestazioni del modello: alcune variabili sono *skewed*, cioè in tali features alcuni valori sono molto più ricorrenti di altri (Figura 6.3).

6.2 Valutazione e selezione dei modelli

Segue la fase di pre-processing, durante la quale i dati vengono manipolati prima di essere utilizzati dal classificatore: questa fase è particolarmente importante per l'addestramento ottimale di un qualunque algoritmo di machine learning.

Nel caso studiato non erano presenti dati mancanti, quindi la fase di pre-processing si è limitata alla riduzione della dimensionalità dei dati e alla loro normalizzazione. Il numero di dimensioni è stato ridotto eliminando la colonna *seconds*, quelle relative all'intensità dell'emozione provata (*_arousal*) e quelle legate a punti di vista del partner e degli osservatori esterni (*partner_* ed *external_*). L'addestramento dei modelli è stato ripetuto effettuando la normalizzazione con le varie tipologie di *scaler*, per poi confrontarne l'efficacia calcolando l'**accuratezza media** del modello ottenuto (Figura 6.4, Tabella 6.1).

max_ iter	*No pre-processing*	*StandardScaler*	*MinMaxScaler*	*RobustScaler*
100	0.497	**0.999**	0.974	**0.999**
200	0.522	**0.999**	0.990	**0.999**
300	0.522	**0.999**	0.996	**0.999**

Tabella 6.2: Valori di mean-test score per classificatore MLP.

C param	*No pre-processing*	*StandardScaler*	*MinMaxScaler*	*RobustScaler*
1	0.454	**0.978**	0.914	0.920
10	0.493	**0.995**	0.963	0.976
100	0.521	**0.998**	0.990	0.996

Tabella 6.3: Valori di mean-test score per classificatore SVC.

Oltre che all'analisi delle tecniche di pre-processing, lo studio si è soffermato nella ricerca dei migliori iper-parametri da applicare ai modelli. Ciò ha implicato una ricerca esaustiva della migliore combinazione mediante **cross-validation**. Nello specifico, sono stati analizzati i seguenti parametri:

- MLPClassifier: *max_ iter = {100, 200, 300}*.
- SVClassifier: *C = {1, 10, 100}*, *probability = True*.

I risultati sono visibili in Figura 6.5, mentre i valori specifici per ciascun modello nelle Tabelle 6.2 e 6.3.

Dai risultati si può notare come la fase di pre-processing infici positivamente nella performance finale dei modelli. In generale, tutti i modelli addestrati dopo la fase di pre-processing mostrano un'accuratezza molto elevata, oltre il 98%, anche se valori leggermente più elevati sono raggiunti dal classificatore MLP.

Inoltre, si può notare come anche l'aumento dei valori per i parametri selezionati rende il sistema più performante.

Relativamente alla fase di pre-processing, dai risultati possiamo vedere che non c'è sostanziale differenza tra i risultati ottenuti. Per questo motivo è stato deciso di proseguire le analisi adottando il **MinMaxScaler**. Questo consente di ridimensionare i dati nel range $[0, 1]$: così facendo, non si perdono informazioni legate alla distribuzione dei valori delle features e inoltre rende più semplice interpretare il modello nelle fasi successive di analisi.

6.3 Analisi dei controfattuali

Dopo aver valutato attentamente le prestazioni dei modelli, lo studio si è concentrato nel tentare di spiegare i processi decisionali che caratterizzano il modello, mediante la **counterfactual explanation**. L'analisi effettuata è stata suddivisa in due fasi: la prima, qualitativa, ha consentito di comparare visivamente i risultati ottenuti,

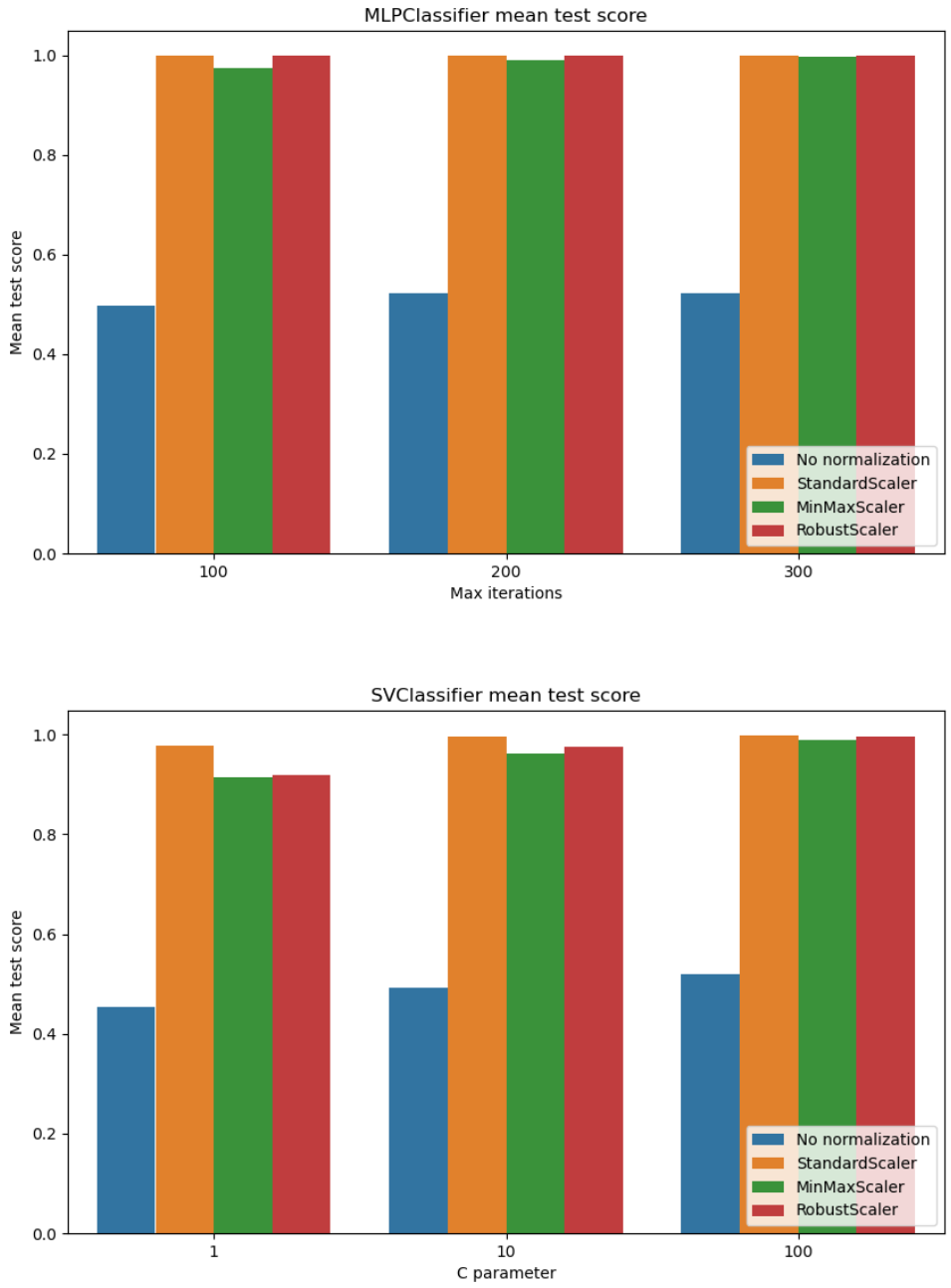


Figura 6.5: Risultati della ricerca dei migliori iper-parametri per differenti modelli.

mentre la seconda, quantitativa, ha permesso di verificare eventuali osservazioni sollevate nella fase precedente, consentendo un'analisi più completa e approfondita dei risultati.

In questo studio è stato utilizzato l'approccio *random* per la ricerca del controfattuale, mentre il tempo limite per l'ottimizzazione della soluzione è stato impostato a 60 secondi durante l'analisi qualitativa e a 15 secondi per l'analisi quantitativa.

6.3.1 Analisi qualitativa

In questa prima fase di analisi sono state selezionate casualmente 20 istanze di dato da analizzare. Per ciascun campione e ciascun modello è stato calcolato il controfattuale. Dopodiché ne sono stati visualizzati i diagrammi greedy e countershapley, al fine di individuare caratteristiche dei grafici, eventuali pattern e correlazioni, sia tra grafici di differente tipologia associati alla stessa istanza, sia tra i modelli di classificazione studiati.

Nello specifico, sono stati utilizzati i seguenti termini per descrivere similitudine o differenza tra i vari diagrammi: *identici, molto simili, mediamente simili, poco simili, completamente diversi* (in questo modo ordinati dal maggiore al minore grado di similitudine).

Analisi di diagrammi della stessa tipologia

Durante questa analisi, è stata posta l'attenzione sulle caratteristiche specifiche di ciascun tipo di grafico studiato, greedy e countershapley, al fine di individuarne eventuali peculiarità, congruenze e incoerenze.

Nello specifico, lo studio si è focalizzato sui grafici prodotti utilizzando il classificatore MLP. Durante l'analisi, sono state riscontrate le seguenti osservazioni:

- **Diagrammi greedy**. Per alcune istanze di dato è stata necessaria la modifica di un numero elevato di features per raggiungere il counterfactual score. Ciò potrebbe essere dovuto al fatto che tali istanze si trovano molto all'interno della propria regione di classificazione, e dunque il controfattuale si trova ad una distanza maggiore (Figure B.8, B.14, B.15, B.16, B.19). Altri samples, non necessariamente appartenenti alle stesse classi, sono risultati *identici* poiché presentano la stessa sequenza di features modificate e potrebbero dunque trovarsi nella stessa posizione rispetto alla classe (o *regione*) di cui ciascuno fanno parte (Figure B.3, B.17). Altri ancora sono tra loro *poco simili* o addirittura *completamente diversi*, nonostante magari appartengano alla stessa classe (Figure B.1, B.19). Infine è interessante notare come alcuni samples, vicini tra loro, presentino grafici *molto simili*, in cui in particolare il factual score è elevato: ciò sta a significare che si trovano in prossimità di un confine decisionale, cioè la zona di confine tra regioni dello spazio di features a cui il modello attribuisce una differente classificazione (Figure B.10, B.11, B.12).

- **Diagrammi countershapley**. Da questo tipo di diagramma si ricavano diverse analogie, ma anche altrettante differenze, rispetto alle correlazioni individuate nei diagrammi greedy. Inoltre, per il metodo con il quale è generato non facendo riferimento ad una particolare combinazione, consente di dare una visuale migliore su quali siano gli attributi più importanti. Anche in questo caso possiamo individuare grafici di vario tipo, più o meno simili (Figure B.22, B.23), determinati da una o più features (Figure B.24, B.37, B.35, B.38).

Per riassumere, osservate le analogie tra le componenti di alcuni campioni, potrebbe essere possibile avanzare ipotesi su quali componenti direzionino maggiormente il processo decisionale del modello verso una certa classe: per esempio, è possibile notare che *lowGamma* e *E4_IBI* siano le componenti principali della maggior parte dei grafici. Ovviamente, per avere conferme su determinate osservazioni, è necessaria la successiva analisi sistematica.

Analisi di differenti grafici della stessa istanza

Nella successiva analisi, è stata invece posta l'attenzione sulle discrepanze e similitudini riscontrate tra le due tipologie di grafico, greedy e countershapley, relativamente alla stessa istanza.

Anche in questo caso, lo studio si è focalizzato, almeno inizialmente, sui grafici prodotti utilizzando il classificatore MLP. Durante l'analisi, è stato osservato come fossero presenti sia elevati gradi di similitudine (Figure B.9, B.28), ma anche grandi differenze (Figure B.14 con B.33, B.16 con B.35), a seconda dell'istanza visionata.

Inoltre, è stato notato come la prima feature della sequenza greedy sia solitamente quella che maggiormente incide nella valutazione generale del countershapley, anche se nella specifica sequenza il suo contributo è poco significativo (Figure B.1 con B.20, B.3 con B.22). Questo potrebbe significare che la feature funge da "innesco" per il cambio della classe e dunque il suo valore risulta importante al fine di classificare l'istanza iniziale.

Infine, è stato possibile notare che l'ordine della sequenza greedy tende, almeno in parte, a rispecchiare l'ordine di importanza delle feature nel grafico greedy (Figure B.15 con B.34, B.18 con B.37).

Correlazioni tra grafici di differenti modelli

Come ultima fase dell'analisi qualitativa, i grafici risultanti dai due modelli sono stati affiancati per verificare se, al variare del modello utilizzato, le osservazioni precedentemente effettuate sono sempre valide.

Con l'introduzione dei grafici basati su SVClassifier, è stato possibile constatare che, in generale, le relazioni osservate con il classificatore MLP valgono anche per SVClassifier, con però alcune differenze da sottolineare. Difatti, nel caso del modello SVC, le sequenze greedy rispecchiano sempre quello che poi è l'ordine di importanza delle feature nel countershapley (Figure B.14, B.33): questo non sempre accade in

MLPC (Figure B.14, B.33). Inoltre, per gli stessi campioni, le feature modificate e i loro contributi possono variare sensibilmente a seconda del modello: in SVC predominano *E4_IBI, E4_HR* ed *E4_EDA*, mentre in MLPC *E4_IBI*, *E4_HR* e *lowGamma*.

6.3.2 Analisi quantitativa

In questa seconda fase di analisi, invece, è stato utilizzato un approccio quantitativo, che ha permesso di verificare l'eventuale validità delle osservazioni precedentemente effettuate. Inoltre, questa analisi ha consentito uno studio più approfondito e generale dei processi decisionali caratterizzanti i modelli confrontati.

Per queste analisi, è stato campionato il 10% del dataset a disposizione (6054 istanze di dato), in modo da minimizzare il bias dovuto alla dimensione, ma senza inficiare eccessivamente nel costo computazionale.

Per ciascun campione sono stati calcolati i controfattuali associati a ciascun modello di classificazione analizzato. Dopodiché, questi sono stati confrontati con il dato iniziale per individuare i due set di features modificate e calcolarne il grado di similarità.

Difatti, se una feature viene modificata durante il calcolo del controfattuale, significa che tale attributo probabilmente è importante per il modello al fine di caratterizzare la classe di partenza.

I risultati, durante la processazione, sono stati salvati con cadenza regolare su un apposito file in formato *csv*, in modo da evitare eccessiva perdita di dati in caso di errori e problemi imprevisti, visto anche il rilevante carico computazionale necessario.

Similarità dei processi decisionali

Per ciascun insieme di set di features, calcolati sulla base di differenti modelli, per la stessa istanza di dato, è stato calcolato il coefficiente di similarità. Come metrica è stato scelto il **Jaccard Index**.

Dopodiché, ne è stata calcolata la distribuzione dei valori, individuando inoltre media e deviazione standard, come riassunto in Figura 6.6: è possibile notare come il Jaccard Index presenta, nella maggior parte dei casi, un valore pari o prossimo a 1, mentre la media è 0.863 e la deviazione standard 0.267: ciò è sintomo di un **elevato grado di similitudine** tra i set calcolati e, di conseguenza, anche tra i controfattuali generati.

Feature importance

Per verificare ulteriormente la similitudine tra i processi decisionali dei due modelli studiati, MLPClassifier e SVClassifier, è stata calcolata anche la frequenza con cui ciascuna feature è coinvolta nel calcolo del controfattuale. I risultati sono visibili in Figura 6.7.

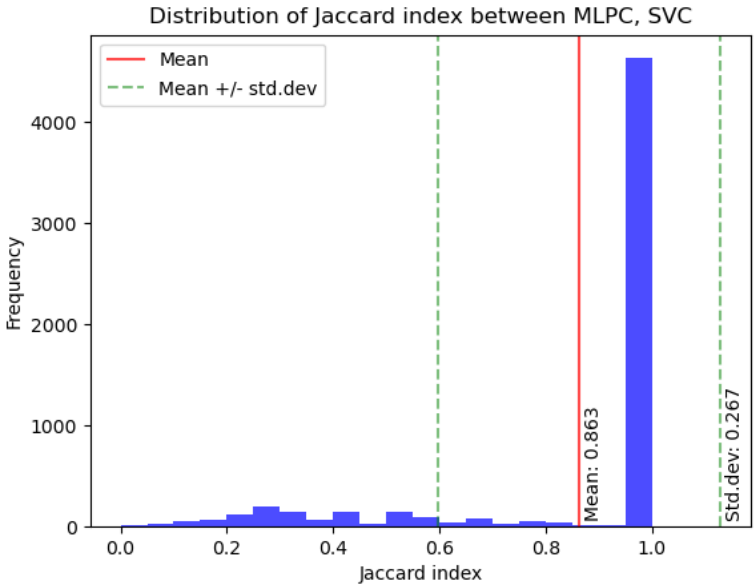


Figura 6.6: Distribuzione degli indici di Jaccard calcolati per i set di features coinvolti nell'analisi del controfattuale.

Come si può notare, i risultati sono importanti. In primo luogo è possibile osservare come questi siano simili per entrambi i classificatori, indicando ancora una volta l'elevato grado di similitudine tra i processi decisionali di ciascun modello. Inoltre, la figura mette in risalto un altro importante risultato: alcune features sono significativamente più frequenti di altre, indicando come queste **incidano maggiormente** nelle scelte di classificazione effettuate dai modelli.

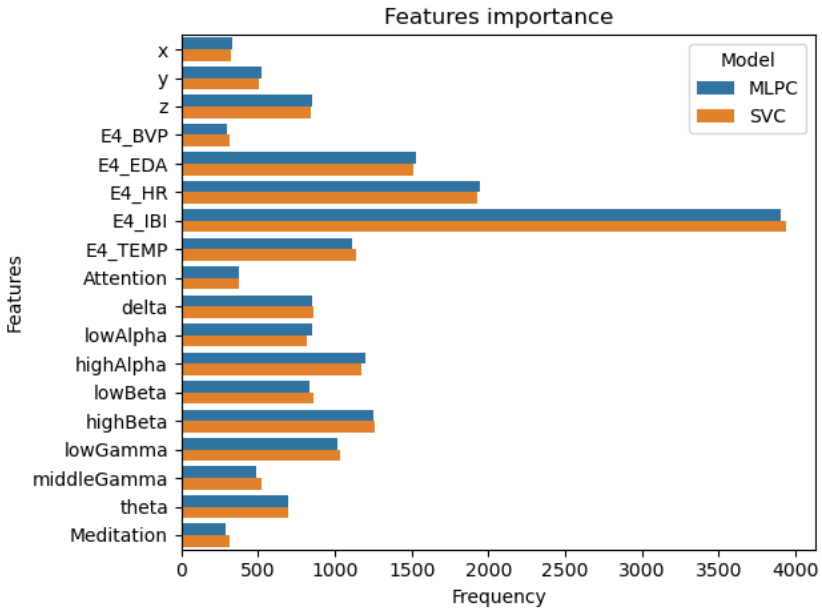


Figura 6.7: Frequenza delle features coinvolte nel calcolo del controfattuale nei modelli studiati.

Capitolo 7

Conclusioni

Dai risultati ottenuti, possiamo trarre valide conclusioni su quanto analizzato nello studio, in particolare per quanto riguarda i modelli più pertinenti al campo dell'Affective Computing, la validità del metodo XAI adottato, la spiegabilità dei modelli e le possibili implementazioni.

7.1 Miglior modello ottenuto

La valutazione delle performance e dell'accuratezza dei modelli addestrati ha dimostrato come il classificatore basato su **Multi-Layer Perceptron** abbia ottenuto le migliori prestazioni. Questo è probabilmente dovuto al fatto che tale modello si basa su reti neurali, dunque ciò lo rende capace di apprendere relazioni complesse e non lineari nei dati. Di conseguenza, risulta maggiormente adatto a catturare la complessità dei dati biometrici analizzati, i quali possono essere influenzati da molteplici fattori e interazioni tra variabili.

Tuttavia, valori così elevati nell'accuratezza dei risultati potrebbero essere sintomo di *overfitting*, ovvero un eccessivo adattamento ai dati di addestramento che ne riduce la capacità di classificarne di nuovi. L'eventuale presenza di questo fenomeno è comunque mitigata dall'utilizzo della tecnica di *GridSearch*, poiché utilizza il meccanismo di *cross-validation* per calcolare le prestazioni del modello. Si ritiene dunque necessaria un'attenta analisi preliminare degli iper-parametri, al fine di bilanciare correttamente il modello e massimizzarne le prestazioni.

Al fine di confermarne o smentirne le elevate capacità di classificazione, potrebbero comunque rendersi necessari alcuni test aggiuntivi, come l'utilizzo di un ulteriore dataset di test.

7.2 Spiegabilità dei modelli

L'analisi delle *counterfactual explanation* ha portato a varie conclusioni:

- **Analisi qualitativa**. L'analisi ha permesso di constatare come, almeno per i campioni studiati, i grafici risultanti fossero simili. Le piccole divergenze dei

risultati qualitativi, ottenuti dalla comparazione dei grafici di alcuni campioni per i differenti modelli, può essere dovuta al loro metodo di implementazione. Difatti, in quanto basati su differenti algoritmi, MLPClassifier risulta molto più complesso e meno interpretabile di SVClassifier, ma può catturare meglio relazioni non lineari tra dati e classi. Queste caratteristiche dell'MLPC potrebbero tradursi in alcuni grafici che presentano maggiori variazioni (in particolare nel diagramma greedy) e una minore relazione tra grafici di diverso tipo. Inoltre, il *cf score* mediamente più elevato, potrebbe far ipotizzare che il classificatore MLP assicuri previsioni con una maggiore certezza, confermandone nuovamente le migliori prestazioni.

- **Analisi quantitativa**. I risultati osservati dalla distribuzione del Jaccard Index (Figura 6.6) hanno permesso di constatare l'**elevato grado di coerenza** tra i processi decisionali dei due modelli, come già intuito in fase di analisi qualitativa.

La coerenza nei processi decisionali potrebbe indicare che entrambi i modelli hanno catturato efficacemente i pattern nei dati di addestramento e sono in grado di generalizzare bene su nuovi dati. Questo comporta una **maggiore robustezza delle previsioni** e una **maggiore affidabilità** nell'accuratezza del modello.

7.2.1 Features importance

Le ulteriori analisi svolte sui risultati ottenuti (Figura 6.7) hanno permesso inoltre di individuare quali sono le feature più frequenti: ***E4_IBI***, seguita da ***E4_HR*** ed ***E4_EDA***. Ciò è un risultato fondamentale, che sta ad indicare come tali features sono particolarmente rilevanti e significative nel processo decisionale dei modelli di classificazione e nella loro capacità predittiva. Questo è importante per diversi fattori:

- **Comprensione dei modelli**. Le features più rilevanti sono quelle che maggiormente incidono nel processo decisionale dei modelli di classificazione e nella loro capacità predittiva, consentendo dunque una migliore comprensione delle motivazioni dietro le scelte effettuate dal classificatore.

- **Selezione delle features**. Identificare le feature più importanti può consentire di semplificare e migliorare i modelli eliminando le feature meno informative o ridondanti.

- **Comprensione del problema**. Le feature più importanti possono inoltre fornire suggerimenti sulle caratteristiche più rilevanti del problema che il modello sta cercando di risolvere. Questo può essere utile per comprendere meglio il problema stesso e identificarne i fattori chiave.

Le features identificate valutano i seguenti parametri:

- **E4_IBI**: Tempo trascorso tra due battiti cardiaci consecutivi.
- **E4_HR**: Numero di battiti cardiaci al minuto.
- **E4_EDA**: Misura della conducibilità elettrica della pelle.

In questo contesto dunque, i modelli proposti hanno identificato una connessione stretta tra la valenza emotiva percepita dal soggetto stesso (*self valence*) e il battito cardiaco (*IBI*, *HR*) ma anche tra la valenza emotiva e l'attività elettrodermica (*EDA*).

7.2.2 Riscontri con altre pubblicazioni

Le correlazioni tra valenza emotiva e i fattori individuati trova riscontro in varie pubblicazioni, avvalorando la robustezza dei risultati trovati e dei modelli generati.

Relazione tra valenza emotiva e battito cardiaco

Solitamente, durante esperienze emotive positive o rilassate, si osserva un aumento dell'IBI e una diminuzione della frequenza cardiaca. Questo perché le emozioni positive tendono a essere associate a una maggiore attivazione del sistema nervoso parasimpatico, che rallenta la frequenza cardiaca e porta a intervalli tra i battiti cardiaci più lunghi (IBI più lungo). Al contrario, durante esperienze emotive negative o stressanti, si osserva spesso un aumento della frequenza cardiaca e una diminuzione dell'IBI, poiché il sistema nervoso simpatico si attiva per preparare il corpo alla risposta di *combattimento o fuga*[11] .

Relazione tra valenza emotiva e attività elettrodermica

L'attività elettrodermica può variare in base all'intensità e alla natura delle emozioni. In generale, esperienze emotive intense, sia positive che negative, possono causare un aumento dell'EDA a causa di un aumento della sudorazione della pelle[5].

7.2.3 Limitazioni durante le analisi

Durante l'analisi della spiegabilità del modello, sono stati impostati i seguenti parametri:

- *Percentuale dei campioni analizzati*: 20 istanze per l'analisi qualitativa, 10% del dataset durante l'analisi quantitativa. Il numero di campioni scelto deve essere abbastanza elevato da costituire una valida rappresentazione del dataset. Di contro, valori troppo elevati, soprattutto nel caso di dataset ampi, aumentano la durata del processo.
- *Tempo limite per la generazione dei controfattuali*: 60 secondi durante l'analisi qualitativa, 15 secondi durante quella quantitativa. Un valore elevato consente

di operare valida ottimizzazione della soluzione, a discapito però dell'aumento dei tempi di processazione. In particolare, è necessario importare accuratamente il parametro, in quanto la fase di ottimizzazione potrebbe prolungarsi indefinitamente.

L'impostazione di questi parametri è stata ritenuta sufficiente a generare risultati validi. Ciononostante, uno studio più approfondito potrebbe aumentare tali valori al fine di ottenere risultati più affidabili, validando o confutando quelli già proposti.

7.3 Possibili applicazioni

I risultati ottenuti hanno un risvolto anche dal punto di vista applicativo, in quanto, eliminando le caratteristiche meno incisive, potrebbe essere possibile comunque creare un sistema per il riconoscimento emotivo dalle buone prestazioni. Ciò si traduce in un minor numero di sensori, consentendo una raccolta dei dati meno invasiva e più semplice per l'utente.

Inoltre, ruolo fondamentale in tutta l'analisi è stato ricoperto dall'Intelligenza Artificiale Spiegabile, che ha consentito di avvalorare l'affidabilità dei modelli prodotti e di migliorare la comprensione dei fenomeni determinanti la valenza emotiva di un soggetto. Ciò ha permesso di creare un'interfaccia, tramite *counterfactual explanations* e *features importance*, maggiormente fruibile anche da utenti non specializzati nel settore IA. Questo consente di ampliare anche ai contesti sensibili, come quello sanitario, l'implementazione di questa tecnologia.

7.4 Sviluppi futuri

Le analisi sono state fatte tenendo conto solo di specifici dati biometrici e della valenza emotiva del soggetto stesso che la percepisce. Ulteriori studi potrebbero ripetere le analisi effettuate sulle altre tipologie di annotazione presenti nel dataset studiato, variando il punto di vista di chi annota l'emozione, oppure considerandone l'intensità (*arousal*) anziché la valenza. Tali studi potrebbero evidenziare se i fattori incidenti il riconoscimento delle emozioni variano al variare del punto di vista dell'osservatore oppure no, implicando anche la rilevazione di eventuali punti in comune con queste analisi.

Inoltre, lo studio è stato effettuato selezionando opportune metriche di valutazione e parametri. Prossimi sviluppi potrebbero implicare nuove analisi con differenti parametri e metriche, al fine di verificare se e quanto queste incidano nei risultati ottenuti.

Infine, sono emerse alcune limitazioni dovute alle tecniche utilizzate. In particolare, un significativo costo computazionale è stato rilevato durante l'ultima fase di analisi, nel processo di calcolo dei controfattuali al fine di estrarne gli indici e le features più rilevanti. Inoltre, il metodo XAI di tipo matematico utilizzato è risultato talvolta di difficile interpretazione, non consentendo sempre effettuare considerazioni

corrette sui dati. Altri studi potrebbero dunque sfruttare altri metodi per la spiegabilità dei modelli, sia matematici che percettivi, in modo da semplificare l'utilizzo del sistema da parte dell'utente finale.

Appendice A

Software

Questa appendice contiene una descrizione generale dell'organizzazione del progetto Python, utilizzato durante le varie fasi di di analisi. Vengono inoltre riportati i codici delle classi e dei principali script implementati.

A.1 Struttura generale

Il progetto, creato tramite PyCharm, si suddivide in varie cartelle:

- **img**: contiene immagini grafiche sui risultati ottenuti nelle varie fasi del progetto, inclusi alcuni snippet di codice.
- **lib**: contiene le varie classi implementate, che hanno permesso di rendere il codice più semplice e organizzato.
- **params**: contiene due file di costanti che hanno reso il codice più flessibile in fase di modifica.
- **resource**: contiene i file *csv* dei dataset e dei risultati ottenuti durante il calcolo degli indici.
- **script**: contiene i file che, utilizzando le implementazioni fornite da *lib* e *params*, hanno consentito le analisi.
- **works**: questa cartella contiene a sua volta le cartelle di ciascun modello addestrato, organizzate in modo da contenere i moduli di configurazione, i modelli addestrati, le valutazioni delle prestazioni e i risultati ottenuti dalle analisi XAI.

La sintassi è stata convalidata attraverso l'uso di *Pylint*.

A.2 Cartella *params*

I due file presenti hanno consentito di poter variare le analisi effettuate di volta in volta in maniera rapida, senza dover ricorrere a modifiche del codice implementato. Di seguito i due moduli.

A.2.1 path.py

Definisce le cartelle del progetto, consentendo di utilizzare i differenti modelli sviluppati in modo semplice e versatile.

```
"""
Defines directories and paths managed in the project.
================================================

This file defines the directories and paths used in the project.
It includes definitions for various directories such as raw_data,
raw_data_img, csv_dir, conf_dir, preprocessed_data_img, results,
and explanation.

It also defines file names such as dataset, prep, params, and model.

Additionally, it defines paths for dataset, preprocessed_dataset,
parameters, model, results, x_train, x_test, y_train, and y_test.

The file also includes code to generate each directory if it does not already exist,
and checks if the dataset exists before continuing the program.

"""

import os
import sys

# Define options for the method and classifier to use in the project
METHODS = ['No normalization', 'MinMaxScaler', 'StandardScaler', 'RobustScaler']
CLASSIFIERS = ['MLPC', 'SVC']

# Define the method and classifier used in the project
PREP_METHOD = METHODS[0]
CLS_METHOD = CLASSIFIERS[0]
ROOT = 'works'
WORK_DIR = os.path.join(ROOT, f'{CLS_METHOD}-{PREP_METHOD}')

# Define the directories used in the project
DIR = {
    'raw_data': 'resource',
    'raw_data_img': os.path.join('img', 'raw_data'),
    'csv_dir': os.path.join(WORK_DIR, 'data'),
    'conf_dir': os.path.join(WORK_DIR, 'config'),
    'preprocessed_data_img': os.path.join(WORK_DIR, 'img', 'preprocessed_data'),
    'results': os.path.join(WORK_DIR, 'results'),
    'explanation': os.path.join(WORK_DIR, 'img', 'explained_CF')
}

# Define the file names used in the project
FILE_NAME = {
    'dataset': 'K-EmoCon_15.csv',
    'prep': 'preprocessedDataset.csv',
    'params': 'modelConfiguration.json',
    'model': 'fitted_model.sav'
}

# Define the paths used in the project
PATH = {
    'dataset': os.path.join(DIR['raw_data'], FILE_NAME['dataset']),
    'preprocessed_dataset': os.path.join(DIR['csv_dir'], FILE_NAME['prep']),
    'parameters': os.path.join(DIR['conf_dir'], FILE_NAME['params']),
    'model': os.path.join(DIR['conf_dir'], FILE_NAME['model']),
    'accuracy': os.path.join(DIR['results'], 'accuracy.txt'),
    'x_train': os.path.join(DIR['csv_dir'], 'x_train.csv'),
    'x_test': os.path.join(DIR['csv_dir'], 'x_test.csv'),
    'y_train': os.path.join(DIR['csv_dir'], 'y_train.csv'),
    'y_test': os.path.join(DIR['csv_dir'], 'y_test.csv')
}

# Generate each directory used if not already exists
for k, d in DIR.items():
    if not os.path.exists(d):
        os.makedirs(d)

# If the dataset does not exist, stop the program
if not os.path.exists(PATH['dataset']):
    print(f'ERROR: {PATH["dataset"]} does not exist.')
    sys.exit(1)
```

A.2.2 attribute_specifications.py

Specifica gli attributi del dataset, in modo da potervisi riferire rapidamente e in modo semplice.

```
"""
Specifications of the attributes in the dataset.
================================================

The file defines various constants related to the attributes in the dataset.
It includes the mapping of attribute indices to their names, class labels,
data labels, skewed data indices, brainwave bands, and categorical labels.

Constants:
    `ATTRIBUTES` (dict): A dictionary mapping attribute indices to their names.
    `CLASS_LABELS` (pandas.Series): A series containing class labels.
    `DATA_LABELS` (pandas.Series): A series containing data labels.
    `SKEWED_DATA` (pandas.Series): A series containing indices of skewed data.
    `BRAINWAVE_BANDS` (pandas.Series): A series containing brainwave band indices.
    `CATEGORICAL_LABELS` (pandas.Series): A series containing indices of categorical labels.
"""

import pandas as pd

# Dict mapping attribute indices to their names
ATTRIBUTES = {
    0: 'seconds',
    1: 'external_arousal',
    2: 'external_valence',
    3: 'partner_arousal',
    4: 'partner_valence',
    5: 'self_arousal',
    6: 'self_valence',
    7: 'x',
    8: 'y',
    9: 'z',
    10: 'E4_BVP',
    11: 'E4_EDA',
    12: 'E4_HR',
    13: 'E4_IBI',
    14: 'E4_TEMP',
    15: 'Attention',
    16: 'delta',
    17: 'lowAlpha',
    18: 'highAlpha',
    19: 'lowBeta',
    20: 'highBeta',
    21: 'lowGamma',
    22: 'middleGamma',
    23: 'theta',
    24: 'Meditation'
}

# Class labels indices
CLASS_LABELS = pd.Series(range(6, 7))

# Data labels indices
DATA_LABELS = pd.Series(range(7, 25))

# Skewed data indices
SKEWED_DATA = pd.Series([10, 13, 16, 17, 18, 19, 20, 21, 22, 23])

# Brainwave bands indices
BRAINWAVE_BANDS = pd.Series(range(16, 24))

# Categorical labels indices
CATEGORICAL_LABELS = pd.Series([6, 15, 24])
```

A.3 Cartella *lib*

Contiene le varie classi implementate, che sono riportate di seguito.

A.3.1 classifier.py

Implementa la classe **Classifier** che consente di creare, addestrare e ottimizzare un modello di classificazione.

```
"""
This module contains the Classifier class.

The Classifier class is used to classify data based on certain criteria or algorithms.
It includes methods for segregating the data into training and testing sets,
performing grid search to find the best parameters, calculating the accuracy of the model,
saving and loading the model configuration, and generating results.
"""

import os
import json
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from numpy import ravel
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import joblib
from params.attribute_specifications import ATTRIBUTES, DATA_LABELS, CLASS_LABELS


class Classifier:
    """
    Classify the data.

    This class provides methods for classifying data using different models. It supports
    segregation of data into training and testing sets, performing grid search to find the
    best parameters, calculating the accuracy of the model, saving and loading the model
    configuration, and generating results including accuracy, best parameters, grid search
    results, and confusion matrix.

    Attributes:
        `data` (pd.DataFrame): The input data for classification.
        `partition` (dict): A dictionary containing the segregated training and testing sets.
        `class_method` (str): The classification model to be used.
        `model` (object): The classification model object.
        `grid` (dict): A dictionary containing the grid search results and best parameters.
        `score` (float): The accuracy score of the model.
        `y_pred` (array-like): The predicted labels for the testing set.

    Methods:
        `segregation()`: Segregate the data into training and testing sets.
        `_get_parameters()`: Get the parameters for the model.
        `grid_search(cval=5)`: Perform grid search to find the best parameters.
        `calculate_accuracy()`: Calculate the accuracy of the model.
        `save_config()`: Save the model configuration.
        `load_config()`: Load the model configuration.
        `generate_results()`: Generate results including accuracy, best parameters, grid search
            results, and confusion matrix.
    """

    CLASSIFIERS = ['MLPC', 'SVC']

    def __init__(self, data, model):
        """Initialize the classifier.

        Args:
            data (pd.DataFrame): The input data for classification.
            model (str): The classification model to be used.
        """

        self.data = data
        self.partition = {
            'x_train': pd.DataFrame(),
            'x_test': pd.DataFrame(),
            'y_train': pd.DataFrame(),
            'y_test': pd.DataFrame()
        }

        self.class_method = model
        if self.class_method == self.CLASSIFIERS[0]:
            self.model = MLPClassifier(random_state=0)
        else:
            self.model = SVC(random_state=0, probability=True)

```

```
78          self.grid = {
79              'results': None,
80              'best_params': None,
81          }
82
83          self.score = None
84          self.y_pred = None
85
86      def segregation(self):
87          """Segregate the data into training and testing sets.
88
89              This method splits the data into training and testing sets
90              using the train_test_split function from the scikit-learn library.
91              It assigns the training and testing data to the 'x_train', 'x_test',
92              'y_train', and 'y_test' attributes of the 'partition' dictionary.
93
94              Returns:
95                  None
96              """
97
98          print('Segregating the data into training and testing sets...')
99          class_columns = [ATTRIBUTES[i] for i in iter(CLASS_LABELS)]
100         data_columns = [ATTRIBUTES[i] for i in iter(DATA_LABELS)]
101         [self.partition['x_train'], self.partition['x_test'],
102          self.partition['y_train'], self.partition['y_test']] = \
103             train_test_split(self.data[data_columns], self.data[class_columns])
104
105         # Print shapes
106         print(f'x_train shape: {self.partition["x_train"].shape}')
107         print(f'x_test shape: {self.partition["x_test"].shape}')
108         print(f'y_train shape: {self.partition["y_train"].shape}')
109         print(f'y_test shape: {self.partition["y_test"].shape}')
110
111     def _get_parameters(self):
112         """Get the parameters for the model.
113
114             Returns:
115                 dict: A dictionary containing the parameters for the model.
116                       The specific parameters depend on the selected classifier.
117                       For the Multi-layer Perceptron classifier, the dictionary
118                       contains the 'max_iter' parameter with possible values of
119                       [100, 200, 300]. For the Support Vector Classifier, the
120                       dictionary contains the 'C' parameter with possible values
121                       of [1, 10, 100] and the 'probability' parameter set to True.
122             """
123
124         # Multi-layer Perceptron
125         if self.class_method == self.CLASSIFIERS[0]:
126             return {
127                 'max_iter': [100, 200, 300]
128             }
129
130         # Support Vector Classifier
131         return {
132             'C': [1, 10, 100],
133             'probability': [True]
134         }
135
136     def grid_search(self, cval=5):
137         """
138         Perform grid search to find the best parameters.
139
140         Parameters:
141         - cval (int): Number of cross-validation folds (default: 5)
142         """
143
144         print('Performing grid search to find the best parameters and model...')
145         grid = GridSearchCV(self.model, self._get_parameters(), cv=cval)
146         grid.fit(self.partition['x_train'], ravel(self.partition['y_train']))
147         self.model = grid.best_estimator_
148         self.grid['results'] = grid.cv_results_
149         self.grid['best_params'] = grid.best_params_
150         print(f'Best parameters found: {self.grid["best_params"]}')
151
152     def calculate_accuracy(self):
153         """Calculate the accuracy of the model.
154
155             Returns:
156                 float: The accuracy score of the model.
157             """
158
159         print('Calculating the accuracy of the model...')
160         self.y_pred = self.model.predict(self.partition['x_test'])
161         self.score = accuracy_score(self.partition['y_test'], self.y_pred)
162
163     def save_config(self,
```

```
164                     model_path='fitted_model.sav',
165                     parameters_path='modelConfiguration.json'):
166         """Save the model and its parameters.
167 
168             This method saves the trained model and its best parameters to disk.
169             The model is saved using joblib.dump() function, and the parameters
170             are saved in a JSON file.
171 
172             Args:
173                 model_path (str): The path to save the model
174                     (default: 'fitted_model.sav')
175                 parameters_path (str): The path to save the parameters
176                     (default: 'modelConfiguration.json')
177 
178             Returns:
179                 None
180             """
181 
182         print('Saving settings...')
183         joblib.dump(self.model, model_path)
184         with open(parameters_path, 'w', encoding='utf-8') as file:
185             json.dump(self.grid['best_params'], file, indent=4)
186 
187     def load_config(self,
188                     model_path='fitted_model.sav',
189                     parameters_path='modelConfiguration.json'):
190         """
191         Load the model in 'self.model',
192         load the parameters which is based on in 'self.grid["best_parameters"]'.
193 
194             Args:
195                 model_path (str): The path to save the model
196                     (default: 'fitted_model.sav')
197                 parameters_path (str): The path to save the parameters
198                     (default: 'modelConfiguration.json')
199 
200         Returns:
201             bool: True if the model exists, else False.
202         """
203 
204         if not os.path.exists(model_path):
205             print('No model found.')
206             return False
207 
208         print('Loading settings...')
209         self.model = joblib.load(model_path)
210         with open(parameters_path, 'r', encoding='utf-8') as file:
211             self.grid['best_params'] = json.load(file)
212 
213         return True
214 
215     def generate_results(self, dir_path='results', accuracy_path='accuracy.txt'):
216         """Generate results.
217 
218             This method generates and prints the accuracy, best parameters,
219             and grid search results.
220             It also saves the accuracy score to a file and generates
221             a confusion matrix for each class.
222 
223             Args:
224                 dir_path (str): The path to the directory where the results
225                                 will be saved (default: 'results')
226                 accuracy_path (str): The path to save the accuracy score
227                                      (default: 'accuracy.txt')
228 
229         """
230         print('Generating results...')
231 
232         # Results
233         print(f'Accuracy: {self.score:.5f}')
234         print(f'Best parameters: {self.grid["best_params"]}')
235         print(f'Grid search results: {self.grid["results"]}')
236         with open(accuracy_path, 'a', encoding='utf-8') as file:
237             file.write(f'{str(self.score)}\n')
238 
239         # Confusion matrix
240         print('Generating confusion matrix...')
241         class_columns = [ATTRIBUTES[i] for i in iter(CLASS_LABELS)]
242         for i, label in enumerate(class_columns):
243             cm_plot = confusion_matrix(self.partition['y_test'], self.y_pred)
244             plt.figure(figsize=(10, 7))
245             axs = sns.heatmap(cm_plot, annot=True, fmt='d')
246             plt.title(f'Confusion matrix for class "{label}"')
247             plt.ylabel('True label')
248             plt.xlabel('Predicted label')
249             axs.set_xticklabels([str(int(tick.get_text()) + 1) for tick in axs.get_xticklabels()])
```

```
250             axs.set_yticklabels([str(int(tick.get_text()) + 1) for tick in axs.get_yticklabels()])
251             plt.savefig(os.path.join(dir_path, f'{label}_confusion_matrix.png'))
```

A.3.2 data_explorer.py

Implementa la classe **DataExplorer** che permette di esplorare un set di dati inerenti al dataset analizzato.

```
"""
This module contains the DataExplorer class.

The DataExplorer class is used to visualize and plot the characteristics of a given dataset.
It includes methods for exploring the data, creating bar plots, temporal graphs, histograms,
and box plots for different types of data (categorical, brainwave bands, and other data).

Typical usage example:

    >>> import pandas as pd
    >>> data = pd.read_csv('data.csv')
    >>> explorer = DataExplorer(data, 'save_dir')
    >>> explorer.exploration()
"""

import os
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from params.attribute_specifications import (
    ATTRIBUTES, CLASS_LABELS, DATA_LABELS, BRAINWAVE_BANDS
)


class DataExplorer:
    """Explore the data.

    This class provides methods to explore and visualize data.

    Attributes:
        data (pandas.DataFrame): The data to be explored.
        save_dir (str): The directory to save the exploration results.
        raw_dir (str): raw data directory (if exploring prep data).
            Default None.
        prep_method (str): prep method used (if exploring prep data).
            Default None.
    """

    def __init__(self, data, save_dir, raw_dir=None, prep_method=None):
        """Initialize the data exploration.

        Args:
            data (pandas.DataFrame): The data to be explored.
            save_dir (str): The directory to save the exploration results.
            raw_dir (str): raw data directory (if exploring prep data).
                Default None.
            prep_method (str): prep method used (if exploring prep data).
                Default None.
        """

        self.data = data
        self.save_dir = save_dir
        self.raw_dir = raw_dir
        self.prep_method = prep_method

    def exploration(self):
        """Explore the data.

        This method performs various data exploration tasks, such as checking for null values,
        duplicates, and creating visualizations of the data.
        """

        print(f'Shape: {self.data.shape}')

        print('Searching for null values...')
        if self.data.isnull().values.any():
            print(f'Null values found: {self.data.isnull().sum().sum()}')
        else:
            print('No null values found!')
```

```
70
71          print('Searching for duplicates...')
72          if self.data.duplicated().any():
73              print(f'Duplicates found: {self.data.duplicated().sum()}')
74          else:
75              print('No duplicates found!')
76
77          # # Number of values occurrences in 'seconds' column
78          # print('Seconds column value counts:')
79          # print(self.data['seconds'].value_counts())
80
81          print('Creating images...')
82          class_columns = {i: ATTRIBUTES[i] for i in iter(CLASS_LABELS)}
83          data_columns = {i: ATTRIBUTES[i] for i in iter(DATA_LABELS)}
84          bw_columns = {i: ATTRIBUTES[i] for i in iter(BRAINWAVE_BANDS)}
85
86          # Plot categorical data characteristics
87          for index, label in class_columns.items():
88              self._categorical_plot(index, label)
89
90          # Plot brainwave band characteristics
91          if self.raw_dir is None:
92              for index, label in bw_columns.items():
93                  self._bw_plot(index, label)
94
95          # Plot other data characteristics
96          for index, label in data_columns.items():
97              if label in CLASS_LABELS:
98                  continue
99              self._data_plot(index, label)
100
101     def _categorical_plot(self, index, label):
102         """
103         Plot categorical data characteristics.
104
105         Parameters:
106         index (int): The index of the plot.
107         label (str): The label of the plot.
108
109         Returns:
110         None
111         """
112         print(f'Plotting {label}...')
113         index = str(index).zfill(2)
114
115         # Bar plot of the data
116         _, axis = plt.subplots()
117         self._create_bar(axis, label)
118         plt.savefig(os.path.join(self.save_dir, f'{index}_{label}_bar.png'))
119         plt.close()
120
121     def _create_bar(self, axs, label):
122         """
123         Create a bar plot of the data.
124
125         Parameters:
126             axs (matplotlib.axes.Axes): The axes object to plot on.
127             label (str): The label of the data to plot.
128
129         Returns:
130             None
131         """
132
133         self.data[label].value_counts().sort_index().plot(kind='bar', ax=axs)
134         axs.set_xlabel(label)
135         axs.set_ylabel('Frequency')
136         axs.set_title(f'{label} bar plot')
137         axs.tight_layout()
138         axs.xaxis.set_major_locator(MaxNLocator(integer=True))
139
140     def _bw_plot(self, index, label):
141         """
142         Plot brainwave band characteristics.
143
144         Parameters:
145         index (int): The index of the brainwave band.
146         label (str): The label of the brainwave band.
147
148         Returns:
149         None
150         """
151
152         print(f'Plotting {label} brainwave...')
153         index = str(index).zfill(2)
154
155         # Temporal graph of the data
```

```
156         if self.raw_dir is None:
157             _, axis = plt.subplots()
158             self.create_temporal_graph(axis, label)
159         else:
160             fig, axs = plt.subplots(1, 2, figsize=(15, 5), constrained_layout=True)
161             fig.suptitle(f'{label} - Before and after {self.prep_method}')
162             path = os.path.join(self.raw_dir, f'temporal_graph_{index}_{label}.png')
163             img = mpimg.imread(path)
164             axs[0].imshow(img)
165             axs[0].axis('off')
166             axs[0].set_aspect(aspect='equal')
167             box = axs[1].get_position()
168             axs[1].set_position([box.x0, box.y0, box.width, box.height - 0.05])
169             self.create_temporal_graph(axs[1], label)
170         plt.savefig(os.path.join(self.save_dir, f'temporal_graph_{index}_{label}.png'))
171         plt.close()
172 
173     def create_temporal_graph(self, axs, label):
174         """
175             Create a temporal graph of the data.
176 
177             Parameters:
178             axs (matplotlib.axes.Axes): The axes object to plot the graph on.
179             label (str): The label of the data to be plotted on the y-axis.
180 
181             Returns:
182             None
183             """
184         times = []
185         values = []
186 
187         times_count = self.data['seconds'].value_counts().sort_index()
188         for time, count in times_count.items():
189             for i in range(count):
190                 times.append((1 - i / count) * time + (i / count) * (time + 5))
191                 values.append(self.data[self.data['seconds'] == time][label].iloc[i])
192         axs.plot(times, values)
193         axs.set_xlabel('Time (seconds)')
194         axs.set_ylabel(label)
195         axs.set_title(f'{label} temporal graph')
196         axs.grid(True)
197 
198     def _data_plot(self, index, label):
199         """
200             Plot data characteristics.
201 
202             Parameters:
203             index (int): The index of the data.
204             label (str): The label of the data.
205 
206             Returns:
207             None
208             """
209         print(f'Plotting {label}...')
210         index = str(index).zfill(2)
211 
212         # Histogram of the data
213         if self.raw_dir is None:
214             _, axis = plt.subplots()
215             self._create_histogram(axis, label)
216         else:
217             fig, axs = plt.subplots(1, 2, figsize=(15, 5), constrained_layout=True)
218             fig.suptitle(f'{label} - Before and after {self.prep_method}')
219             path = os.path.join(self.raw_dir, f'histogram_{index}_{label}.png')
220             img = mpimg.imread(path)
221             axs[0].imshow(img)
222             axs[0].axis('off')
223             axs[0].set_aspect(aspect='equal')
224             box = axs[1].get_position()
225             axs[1].set_position([box.x0, box.y0, box.width, box.height - 0.05])
226             self._create_histogram(axs[1], label)
227         plt.savefig(os.path.join(self.save_dir, f'histogram_{index}_{label}.png'))
228         plt.close()
229 
230         # Box plot of the data
231         if self.raw_dir is None:
232             _, axis = plt.subplots()
233             self._create_box_plot(axis, label)
234         else:
235             fig, axs = plt.subplots(1, 2, figsize=(15, 5), constrained_layout=True)
236             fig.suptitle(f'{label} - Before and after {self.prep_method}')
237             path = os.path.join(self.raw_dir, f'box_plot_{index}_{label}.png')
238             img = mpimg.imread(path)
239             axs[0].imshow(img)
240             axs[0].axis('off')
241             axs[0].set_aspect(aspect='equal')
```

```
            box = axs[1].get_position()
            axs[1].set_position([box.x0, box.y0, box.width, box.height - 0.05])
            self._create_box_plot(axs[1], label)
        plt.savefig(os.path.join(self.save_dir, f'box_plot_{index}_{label}.png'))
        plt.close()

    def _create_histogram(self, axs, label):
        """
        Create a histogram of the data.

        Parameters:
        axs (matplotlib.axes.Axes): The axes object to plot the histogram on.
        label (str): The label of the data to plot.

        Returns:
        None
        """

        axs.hist(self.data[label], bins=50)
        axs.set_xlabel(label)
        axs.set_ylabel('Frequency')
        axs.set_title(f'{label} histogram')
        axs.grid(True)

    def _create_box_plot(self, axs, label):
        """Create box plot of the data.

            Args:
                axs (matplotlib.axes.Axes): The axes object to plot the box plot on.
                label (str): The label of the data to create the box plot for.

            Returns:
                None
            """

        # Create the box plot
        box_plot = axs.boxplot(self.data[label])
        # Add annotations for the whisker values
        lower_whisker = box_plot['whiskers'][0].get_ydata()[1]
        upper_whisker = box_plot['whiskers'][1].get_ydata()[1]
        axs.annotate(f'Lower: {lower_whisker:.2f}', (1, lower_whisker),
                     textcoords="offset points", xytext=(-70, -10), ha='center')
        axs.annotate(f'Upper: {upper_whisker:.2f}', (1, upper_whisker),
                     textcoords="offset points", xytext=(-70, 10), ha='center')
        axs.set_xlabel(label)
        axs.set_ylabel('Frequency')
        axs.set_title(f'{label} box plot')
        axs.grid(True)

    def create_bar_plot(self, axs, label):
        """
        Create a bar plot of the data.

        Parameters:
            axs (matplotlib.axes.Axes): The axes object to plot on.
            label (str): The label of the data to plot.

        Returns:
            None
        """

        axs.bar(self.data[label], self.data[label])
        axs.set_xlabel(label)
        axs.set_ylabel('Value')
        axs.set_title(f'{label} bar plot')
        axs.grid(True)
```

A.3.3 data_preprocessor.py

Implementa la classe **DataPreprocessor**, che consente di pre-processare i dati con lo scaler selezionato nel file *path.py*.

```
"""
This module contains the DataPreprocessor class.

The DataPreprocessor class is used to clean, reduce, and transform data
before it is used for classification. It includes methods for
```

```
handling missing values, encoding categorical variables, normalizing numerical variables,
and other preprocessing tasks.

Typical usage example:

    >>> import pandas as pd
    >>> data = pd.read_csv('data.csv')
    >>> preprocessor = DataPreprocessor(data)
    >>> preprocessed_data = preprocessor.preprocess('prep_method')
"""

from sklearn.preprocessing import RobustScaler, StandardScaler, MinMaxScaler
from params.attribute_specifications import ATTRIBUTES, DATA_LABELS


class DataPreprocessor:
    """Preprocess the data.

    This class provides methods to preprocess the data, including removing missing values,
    reducing the data, normalizing the data, and saving the preprocessed data.

    Attributes:
        data (DataFrame): The input dataset to be preprocessed.

    Methods:
        __init__(self, data): Initializes the DataPreprocessor object.
        preprocess(self): Preprocesses the data and returns the preprocessed dataset.
        reduce_data(self): Reduces the data by removing unnecessary columns.
        normalize_data(self): Normalizes the data using different scaling methods.
        save_preprocessed_data(self): Saves the preprocessed data to a CSV file.
    """

    def __init__(self, data, path='preprocessedDataset.csv'):
        """Initialize the DataPreprocessor class.

        Args:
            data (DataFrame): The input data to be preprocessed.
            path (str): File path to save preprocessed data.
        """

        self.data = data
        self.path = path
        self.prep_method = None

    def preprocess(self, prep_method):
        """Preprocess the data.

        Args:
            prep_method (str): method used to scale data

        Returns:
            :return dataFrame: the preprocessed dataset.
        """
        print('Preprocessing the data...')
        self.prep_method = prep_method
        # Remove missing values
        self.data.dropna(inplace=True)
        # Reduce data
        self.reduce_data()
        # Normalize data
        self.normalize_data()
        # Save preprocessed data
        self.save_preprocessed_data()
        return self.data

    def reduce_data(self):
        """Reduce the data.

        This method reduces the data by removing columns before the 'self_valence' column.
        """
        print('Reducing the data...')

        # Remove columns before 'self_valence' column
        sv_index = next((i for i, v in ATTRIBUTES.items() if v == 'self_valence'), None)
        self.data = self.data.iloc[:, sv_index:]

    def normalize_data(self):
        """Transform the data.

        This method applies scaling to the numerical columns of the data
        using the specified scaling method. It iterates over the data columns
        and applies the appropriate scaler based on the self.prep_method.
        The scaled values are then assigned back to the respective columns in the data.

        Returns:
            None
```

```
92          """
93
94          print('Transforming the data...')
95
96          # Normalize data
97          data_columns = [ATTRIBUTES[i] for i in iter(DATA_LABELS)]
98          for label in data_columns:
99              if self.prep_method == 'MinMaxScaler':
100                 scaler = MinMaxScaler()
101             elif self.prep_method == 'RobustScaler':
102                 scaler = RobustScaler()
103             elif self.prep_method == 'StandardScaler':
104                 scaler = StandardScaler()
105             else:
106                 return
107             self.data[label] = scaler.fit_transform(self.data[label].values.reshape(-1, 1))
108
109     def save_preprocessed_data(self):
110         """Save the preprocessed data.
111
112         This method saves the preprocessed data to a CSV file.
113         The file path is specified by path.
114         The data is saved without including the index column.
115
116         """
117         print('Saving the preprocessed data...')
118         self.data.to_csv(self.path, index=False)
```

A.3.4 explainer.py

Contiene la classe **Explainer** che permette di esplorare le scelte decisionali del modello per determinate istanze di dato, mediante il calcolo dei controfattuali e la visualizzazione dei grafici associati. Si appoggia a *cfnow*.

```
1  """
2  This module uses the cfnow library to explain models.
3
4  The cfnow library is a powerful tool for interpreting machine learning models.
5  It provides methods for generating counterfactual explanations,
6  which can help understand how a model makes its predictions.
7
8  """
9
10 import os
11 import warnings
12 import pandas as pd
13 import numpy as np
14 from cfnow import find_tabular
15 warnings.simplefilter(action='ignore', category=FutureWarning)
16
17
18 class Explainer:
19     """Explain the model.
20
21     This class provides methods to explain a machine learning model
22     by computing counterfactuals and Shapley values.
23     It takes a set of samples, data, model, and an optional timeout parameter
24     as input during initialization.
25
26     Attributes:
27         samples (list): The indices of the samples to be explained.
28         data (pandas.DataFrame): The input features.
29             It contains the input features (X) and the target classes (Y).
30         classes (pandas.DataFrame): The target classes.
31         model (object): The machine learning model.
32         timeout (float): The maximum time allowed for computing counterfactuals.
33
34     Methods:
35         plot_counterfactuals(mod=None): Compute and plot counterfactuals for each sample.
36         compute_shapley_values(): Compute Shapley values for each sample.
37
38     Private Methods:
39         _generate_cf_obj(sample): Generate a counterfactual object for a given sample.
40     """
41
42     # Available graph modes
```

```
43      MOD = ['greedy', 'countershapley', 'constellation']
44
45      def __init__(self, samples, data, model, timeout=None):
46          """Initialize the explainer.
47
48          Args:
49              samples (list): The indices of the samples to be explained.
50              data (dict): The input features and target classes.
51              model (object): The machine learning model.
52              timeout (float, optional): The maximum time allowed for computing counterfactuals.
53          """
54
55          self.samples = samples
56          self.data = data['X']
57          self.classes = data['Y']
58          self.model = model
59          self.timeout = timeout
60
61      def plot_counterfactuals(self, cls_method, exp_dir='explanation',  mod=None):
62          """Compute counterfactual for each sample
63
64          This method computes the counterfactual for each sample in the Explainer object.
65          It generates counterfactual objects using the _generate_cf_obj method
66          and generates plots for each counterfactual.
67
68          Args:
69              mod (list, optional): The graph modes to be generated.
70                  Defaults to None, which generates all available modes.
71              cls_method (str): classifier method used
72              exp_dir (str): directory to save graphs
73
74
75          Returns:
76              None
77          """
78
79          # Default: all
80          if mod is None:
81              mod = self.MOD
82          # Create dirs if not exist
83          if not os.path.exists(os.path.join(exp_dir, 'greedy')):
84              os.makedirs(os.path.join(exp_dir, 'greedy'))
85          if not os.path.exists(os.path.join(exp_dir, 'counterShapley')):
86              os.makedirs(os.path.join(exp_dir, 'counterShapley'))
87          if not os.path.exists(os.path.join(exp_dir, 'constellation')):
88              os.makedirs(os.path.join(exp_dir, 'constellation'))
89
90          # Compute counterfactuals for each sample
91          for sample in self.samples:
92
93              # Generate cf_obj
94              cf_obj, _, _ = self._generate_cf_obj(sample, cls_method)
95              if cf_obj is None:
96                  continue
97
98              # Generating plots
99              counter_plot = cf_obj.generate_counterplots(0)
100
101             if self.MOD[0] in mod:
102                 print('Generating greedy counterplot...')
103                 counter_plot.greedy(
104                     os.path.join(exp_dir, 'greedy',
105                                  f'{sample}_greedy_counterplot.png')
106                 )
107
108             if self.MOD[1] in mod:
109                 print('Generating counterShapley counterplot...')
110                 counter_plot.countershapley(
111                     os.path.join(exp_dir, 'counterShapley',
112                                  f'{sample}_counterShapley_counterplot.png')
113                 )
114
115             if self.MOD[2] in mod:
116                 print('Generating constellation counterplot...')
117                 counter_plot.constellation(
118                     os.path.join(exp_dir, 'constellation',
119                                  f'{sample}_constellation_counterplot.png')
120                 )
121
122     def compute_shapley_values(self, cls_method, exp_dir='explanation'):
123         """Compute Shapley values
124
125         This method computes the Shapley values for each sample.
126         Shapley values are a method for assigning importance scores
127         to features in a model. They quantify the contribution of each feature
128         towards the prediction made by the model.
```

```
        The Shapley values are computed using the counterfactual objects
        generated for each sample.

        Args:
            cls_method (str): classifier method used
            exp_dir (str): directory to save graphs

        Returns:
            None
        """

        # Choose sample already analyzed
        new_samples = []
        try:
            loaded_result = pd.read_csv(
                os.path.join(exp_dir, 'shapley_values.csv'),
                index_col=0,
                header=0
            )
        except (FileNotFoundError, pd.errors.EmptyDataError):
            print('No data already studied.')
        else:
            idx = set(loaded_result.index)
            new_samples = [sample_data for sample_data in self.samples if sample_data not in idx]

        # Compute counterfactuals for each sample
        for sample in new_samples:

            # Generate cf_obj
            cf_obj, factual_class, counterfactual_class = self._generate_cf_obj(sample, cls_method)
            if cf_obj is None:
                continue

            # Generating plots
            counter_plot = cf_obj.generate_counterplots(0)
            print("Generating countershapley values...")
            cs_values = counter_plot.countershapley_values()

            # Create a dictionary with 'sample', 'factual_class', and 'counterfactual_class'
            data = {
                'sample': sample,
                'factual_class': factual_class,
                'counterfactual_class': counterfactual_class
            }

            # Add each column from self.data.columns to the dictionary
            for column in self.data.columns:
                if column in cs_values['feature_names']:
                    index = cs_values['feature_names'].index(column)
                    data[column] = cs_values['feature_values'][index]
                else:
                    data[column] = 0

            # Append to file
            result_df = pd.DataFrame(data, index=[data['sample']])
            result_df.to_csv(
                os.path.join(exp_dir, 'shapley_values.csv'),
                mode='a',
                header=False,
                index=False
            )

    def _generate_cf_obj(self, sample, cls_method):
        """Generate counterfactual object

        This method generates a counterfactual object for a given sample.
        It takes the sample index as input and returns the counterfactual object,
        the factual class, and the counterfactual class.

        Parameters:
            sample (int): The index of the sample.

        Returns:
            cf_obj (object): The counterfactual object.
            factual_class (int): The factual class of the sample.
            counterfactual_class (int): The counterfactual class of the sample.
        """

        sample_data = pd.Series(self.data.iloc[sample], name='Factual')

        # Skipping misclassified data
        # In general, model prediction and probability prediction function could return
        # different classifications. We use probability prediction because cfnow use it.
        original_label = self.classes.iloc[sample]['self_valence']
        prob_pred_label = np.argmax(self.model.predict_proba([sample_data])[0]) + 1
        cls_pred_label = self.model.predict([sample_data])[0]
```

```
215
216         if prob_pred_label != original_label:
217             print(f"Sample {sample} is misclassified by probability function "
218                   f"(label: {original_label}, "
219                   f"probability prediction: {prob_pred_label}, "
220                   f"{cls_method} prediction: {cls_pred_label}"
221                   "), move on to the next one!")
222             return None, None, None
223
224         print(f"Sample {sample}: "
225               f"label={original_label}, "
226               f"prob={prob_pred_label}, "
227               f"class={cls_pred_label}")
228
229         # Generate the minimum change
230         print(f'Generating counterfactual object for sample {sample}...')
231         cf_obj = find_tabular(
232             factual=sample_data,
233             feat_types={c: 'num' for c in self.data.columns},
234             model_predict_proba=self.model.predict_proba,
235             limit_seconds=self.timeout
236         )
237
238         # Display the new class
239         cf_data = pd.Series(cf_obj.cfs[0], index=self.data.columns, name='CounterFact')
240         diff = pd.Series((cf_data - sample_data), index=self.data.columns, name='Difference')
241         result_df = pd.concat([sample_data, cf_data, diff], axis=1)
242         factual_class = prob_pred_label
243         prob_cf_class = np.argmax(self.model.predict_proba([cf_obj.cfs[0]])[0]) + 1
244         cls_cf_class = self.model.predict([cf_obj.cfs[0]])[0]
245         print(f"{result_df}")
246         print(f"Factual class: {factual_class}")
247         print(f"Counterfactual class (by probability function): {prob_cf_class}")
248         print(f"Counterfactual class (by {cls_method}): {cls_cf_class}")
249
250         return cf_obj, factual_class, prob_cf_class
```

A.3.5 jaccard_evaluer.py

Contiene la classe **JaccardEvaluer** che consente di valutare il grado di similarità di differenti modelli basandosi sulle counterfactual explanations. Si appoggia a *cfnow*.

```
1  """
2  This module contains the JaccardEvaluator class.
3
4  The JaccardEvaluator class is used to compute the Jaccard similarity coefficient
5  between the predictions of different classifiers. This coefficient measures
6  the similarity between two sets and is defined as the size of the intersection
7  divided by the size of the union of the sets.
8
9  Typical usage example:
10
11     >>> import pandas
12     >>> data = pandas.read_csv('data.csv')
13     >>> X = pandas.read_csv('X.csv')
14     >>> y = pandas.read_csv('y.csv')
15     >>> evaluator = JaccardEvaluer(
16     >>>     data={'dataset': data, 'X': X, 'y': y},
17     >>>     path='path'
18     >>> )
19     >>> evaluator.compute_jaccard(clfs)
20 """
21 import warnings
22 import pandas as pd
23 import numpy as np
24 import seaborn as sns
25 import matplotlib.pyplot as plt
26 from cfnow import find_tabular
27 from params.attribute_specifications import ATTRIBUTES, DATA_LABELS
28
29 warnings.filterwarnings("ignore", category=UserWarning)
30
31
32 def jaccard_similarity(*sets):
33     """
34     Compute the Jaccard similarity coefficient for a given set of sets.
35
```

```
    Parameters:
    sets (tuple): A tuple of sets for which the Jaccard similarity coefficient needs to be computed.

    Returns:
    float: The Jaccard similarity coefficient, which is a value between 0 and 1.
        If the union of all sets is empty, returns None.
    """
    intersection = set.intersection(*sets)
    union = set.union(*sets)
    if len(union) == 0:
        return None
    return len(intersection) / len(union)


class JaccardEvaluer:
    """
    The JaccardEvaluer class is used to measure the Jaccard values between classifiers.

    Attributes:
        data (dict): Contains dataset, X, y.
        path (str): The path to save the evaluation results (csv file).
        samples (list): List of selected samples.
        percentage (float): The percentage of data to sample.
        timeout (int): Timeout value for computation.
        save_freq (int): Frequency of saving results.

    Methods:
        __init__(self, data, path='jaccard_indexes.csv'): Initializes the JaccardEvaluer class.
        sample_data(self, percentage=0.1): Samples the data.
        compute_jaccard(self, clfs, timeout=15, save_freq=25):
            Computes Jaccard values between classifiers.
    """

    # Data columns labels for the dataset
    data_columns = [ATTRIBUTES[sample] for sample in iter(DATA_LABELS)]

    def __init__(self, data, path='jaccard_indexes.csv'):
        """
        Initialize the JaccardEvaluer class.

        Parameters:
            data (dict): Contains dataset, X, y.
            path (str): The path to save the evaluation results (csv file).

        Returns:
            None
        """

        self.data = data
        self.path = path
        self.samples = None
        self.percentage = None
        self.timeout = None
        self.save_freq = None

    def sample_data(self, percentage=0.1):
        """Sample the data

        This method samples the data based on the specified percentage.
        It selects a subset of the dataset randomly and sets it
        as the samples attribute of the JaccardEvaluer object.

        Parameters:
            percentage (float): The percentage of data to sample. Defaults to 0.1.

        Returns:
            None
        """

        # Initialize percentage
        self.percentage = percentage

        # Choose samples
        sample_size = int(np.round(self.data['dataset'].shape[0] * self.percentage))
        self.samples = self.data['dataset'].sample(n=sample_size, random_state=42).index
        print(f"Selected indexes: {len(self.samples)}")

        # Delete samples already computed
        try:
            read_df = pd.read_csv(self.path, index_col=0)
        except (FileNotFoundError, pd.errors.EmptyDataError):
            print("File not found.")
        else:
            read_samples = set(read_df.index)
            self.samples = [data for data in self.samples if data not in read_samples]

```

```
122     def compute_jaccard(self, clfs, timeout=15, save_freq=25):
123         """Compute Jaccard values between classifiers
124 
125         This method computes the Jaccard values between classifiers for each sample in the dataset.
126         It generates counterfactual objects using the find_tabular function and calculates the
127         Jaccard index based on the changed features. The results are saved periodically and
128         the final results are printed once all samples are processed.
129 
130         Parameters:
131             clfs (list): List of classifiers.
132             timeout (int): Timeout value for computation. Defaults to 15.
133             save_freq (int): Frequency of saving results. Defaults to 25.
134 
135         Returns:
136             None
137         """
138 
139         self.timeout = timeout
140         self.save_freq = save_freq
141 
142         jaccard_dict = {}
143         i = 0
144         for sample in self.samples:
145             print(f"Computing jaccard index for sample {sample}...")
146 
147             # Generating sets
148             sampled_data = pd.Series(self.data['X'].iloc[sample], name='Factual')
149             feat_types = {idx: 'num' for idx in self.data_columns}
150             sample_sets = []
151             for classifier in clfs:
152                 # Generate cf object
153                 print(f'Generating cf obj (model: {classifier.class_method})...')
154                 cf_obj = find_tabular(
155                     factual=sampled_data,
156                     feat_types=feat_types,
157                     model_predict_proba=classifier.model.predict_proba,
158                     limit_seconds=self.timeout
159                 )
160 
161                 # Computing changed features
162                 if len(cf_obj.cfs) != 0:
163                     cf_data = pd.Series(cf_obj.cfs[0], index=self.data_columns, name='CounterFact')
164                     diff = pd.Series((cf_data - self.data['X'].iloc[sample]),
165                                      index=self.data_columns, name='Difference')
166                     feature_set = set(diff[diff != 0].index)
167                     print(f"Changed Feature(sets): {feature_set}")
168                 else:
169                     print("No counterfactual found!")
170                     feature_set = set()
171 
172                 # Adding set to array
173                 sample_sets.append(feature_set)
174 
175             # Create dict with results info
176             jaccard_dict[sample] = self._jaccard_dict_row(
177                 sample=sample,
178                 sample_sets=sample_sets,
179                 clfs=clfs
180             )
181 
182             # Save results every self.save_freq samples
183             i += 1
184             if i == self.save_freq:
185                 self._save_results(
186                     jaccard_dict=jaccard_dict,
187                     sample_num=i
188                 )
189                 i = 0
190                 jaccard_dict = {}
191 
192         # Save last results
193         if i != 0:
194             self._save_results(
195                 jaccard_dict=jaccard_dict,
196                 sample_num=i
197             )
198         print(f"COMPUTED JACCARD INDEX FOR ALL {len(self.samples)} SAMPLES!")
199 
200     def plot_jaccard_hist(self, clfs, path='jaccard_indexes_histogram.png'):
201         """Plot Jaccard results
202 
203         This method plots the Jaccard results by creating
204         a bar plot of the feature importance.
205         It reads the results from the specified path and computes
206         the common features for each classifier.
207         The resulting bar plot shows the frequency of each feature
```

```
208         for each classifier.
209
210         Parameters:
211             clfs (list): List of classifiers.
212             path (str): The path to save the plot.
213
214         Returns:
215             None
216         """
217
218         print("Plotting jaccard index values distribution...")
219
220         # Read results
221         result_df = pd.read_csv(self.path, index_col=0, header=0)
222
223         # Compute media and std.dev
224         mean = result_df['jaccard_index'].mean()
225         std_dev = result_df['jaccard_index'].std()
226         print(f"Mean: {round(mean, 3)}")
227         print(f"Standard dev.: {round(std_dev, 3)}")
228
229         # Histogram plot
230         plt.figure()
231         plt.hist(result_df['jaccard_index'], bins=20, color='blue', alpha=0.7)
232         # Mean and std.dev lines
233         plt.axvline(x=mean, color='r', linestyle='-', label='Mean', alpha=0.7)
234         plt.axvline(x=mean + std_dev, color='g', linestyle='--',
235                     label='Mean +/- std.dev', alpha=0.5)
236         plt.axvline(x=mean - std_dev, color='g', linestyle='--', alpha=0.5)
237         # Text
238         plt.text(mean + 0.01, 50,
239                  f'Mean: {round(mean, 3)}', rotation=90, verticalalignment='bottom')
240         plt.text(mean + std_dev + 0.01, 50,
241                  f'Std.dev: {round(std_dev, 3)}', rotation=90, verticalalignment='bottom')
242         # Labels
243         plt.legend(loc='best')
244         plt.title(f"Distribution of Jaccard index between "
245                   f"{', '.join([classifier.class_method for classifier in clfs])}")
246         plt.xlabel('Jaccard index')
247         plt.ylabel('Frequency')
248         # Save
249         plt.savefig(path)
250         plt.close()
251
252     def plot_feature_importance(self, clfs, path='features_importance.png'):
253         """Plot feature importance
254
255         This method plots the feature importance by creating a bar plot.
256         It reads the results from the specified path and computes the
257         common features for each classifier.
258         The resulting bar plot shows the frequency of each feature for each classifier.
259
260         Parameters:
261             clfs (list): List of classifiers.
262             path (str): The path to save the plot.
263
264         Returns:
265             None
266         """
267
268         print("Plotting features importance...")
269
270         # Read results
271         result_df = pd.read_csv(self.path, index_col=0, header=0)
272
273         common_features = {}
274         for classifier in clfs:
275             common_features[classifier.class_method] = \
276                 {feat: sum(feat in row for row in result_df[classifier.class_method + '_features'])
277                  for feat in self.data_columns}
278
279         # Create dataframe
280         data = []
281         for model, feat in common_features.items():
282             for key, value in feat.items():
283                 data.append([model, key, value])
284         data = pd.DataFrame(data, columns=['Model', 'Feature', 'Frequency'])
285
286         # Plot
287         sns.barplot(x='Frequency', y='Feature', hue='Model', data=data)
288         plt.title('Features importance')
289         plt.xlabel('Frequency')
290         plt.ylabel('Features')
291         plt.tight_layout()
292         # Save
293         plt.savefig(path)
```

```
294
295     def _save_results(self, jaccard_dict, sample_num):
296         """Save jaccard indexes computed to file
297
298         This method saves the computed Jaccard indexes to a file.
299         It takes a dictionary of Jaccard indexes and the number of samples as input.
300         The method appends the Jaccard indexes to an existing file or creates
301         a new file if it doesn't exist.
302         It also prints the progress of saving the samples.
303
304         Parameters:
305             jaccard_dict (dict): Dictionary of Jaccard indexes.
306             sample_num (int): Number of samples.
307
308         Returns:
309             None
310         """
311
312         print(f"SAVING LASTS {sample_num} SAMPLE(S)...")
313
314         # Total number of samples to compute
315         sample_size = int(np.round(self.data['dataset'].shape[0] * self.percentage))
316
317         # Create dataframe
318         jaccard_df = pd.DataFrame.from_dict(jaccard_dict, orient='index')
319         try:
320             # If file already exists and not empty, append results
321             read_df = pd.read_csv(self.path, index_col=0, header=0)
322             jaccard_df.to_csv(self.path, mode='a', header=False)
323             perc = round((read_df.shape[0] + sample_num) * 100 / self.data['dataset'].shape[0], 2)
324             print(f"{read_df.shape[0] + sample_num} self.samples DONE "
325                   f"({perc}%) "
326                   f"REMAINING: {sample_size - read_df.shape[0] - sample_num}")
327         except (FileNotFoundError, pd.errors.EmptyDataError):
328             # Create file if not exists or empty
329             jaccard_df.to_csv(self.path,
330                               index_label='sample', header=True)
331             print(f"{sample_num} self.samples SAVED "
332                   f"({round(sample_num * 100 / self.data['dataset'].shape[0], 2)}%) "
333                   f"REMAINING: {sample_size - sample_num}")
334
335     def _jaccard_dict_row(self, sample, sample_sets, clfs):
336         """Create dict for Jaccard index computed on a sample
337
338         This method takes a sample, sample sets, and classifiers
339         as input and computes the Jaccard index for the sample.
340         It creates a dictionary containing the Jaccard index,
341         the original class of the sample, and the predicted class
342         and set features for each model.
343
344         Parameters:
345             sample (int): The index of the sample.
346             sample_sets (list): List of sets for the sample.
347             clfs (list): List of classifiers.
348
349         Returns:
350             dict: Dictionary containing the Jaccard index,
351             original class, predicted class, and set features for each model.
352         """
353
354         # Compute jaccard index
355         jaccard_index = jaccard_similarity(*sample_sets)
356         print(f"Jaccard index for sample {sample}: {jaccard_index}")
357
358         # Sampled data
359         sampled_data = pd.Series(self.data['X'].iloc[sample], name='Factual')
360
361         # Jaccard index and original sample class
362         row = {
363             'jaccard_index': jaccard_index,
364             'original_class': self.data['y'].iloc[sample]['self_valence']
365         }
366
367         # Predicted class and set features for each model
368         for k, sets in enumerate(sample_sets):
369             row[str(clfs[k].class_method) + '_class'] = \
370                 np.argmax(clfs[k].model.predict_proba([sampled_data])) + 1
371             row[str(clfs[k].class_method) + '_features'] = sets
372
373         return row
```

A.3.6 timer.py

Contiene la classe **Timer**, utile per verificare i tempi di esecuzione degli script.

```
"""
This module contains the Timer class.

The Timer class is used to measure the execution time of code.
It includes methods for starting, stopping, and resetting the timer,
as well as getting the elapsed time.

Typical usage example:

    >>> timer = Timer()
    >>> timer.start()
    >>> # Some code here...
    >>> timer.end()
"""
import sys
import os
import time


class Timer:
    """
    Timer class for measuring the execution time of code.

    Methods:
        __init__(self, target=None): Initializes the timer.
        start(self): Starts the timer.
        end(self): Prints the time elapsed since execution began.
    """

    def __init__(self, target=None):
        """Initialize timer with optional target name.

            Args:
                target (str, optional): The name of the target. Defaults to None.
        """

        self.start_time = None
        self.end_time = None
        if target is None:
            self.target = os.path.basename(sys.argv[0])
        else:
            self.target = target

    def start(self):
        """Starts timer"""
        self.start_time = time.time()

    def end(self):
        """Print time elapsed since execution began"""
        self.end_time = time.time()
        hrs = (self.end_time - self.start_time) // 3600
        minutes = ((self.end_time - self.start_time) % 3600) // 60
        sec = (self.end_time - self.start_time) % 60
        print(f"{self.target} "
              f"- Execution time: "
              f"{int(hrs)} hour(s) {int(minutes)} minute(s) {sec:.3f} second(s).")
```

A.4 Cartella *script*

I file python presenti in questa cartella sono stati utilizzati per eseguire le varie fasi di analisi basandosi sulle classi implementate e le costanti selezionate. Di seguito ne vengono descritti i principali moduli.

A.4.1 model_generation.py

Il modulo rappresenta il file eseguibile mediante il quale vengono effettuate le fase di esplorazione dei dati, pre-processing, addestramento e ottimizzazione del modello e valutazione delle prestazioni ottenute. Di seguito è riportato il codice implementato.

```
"""
This module contains the main script for generating machine learning models.

The script in this module is used to train machine learning models on a given dataset.
It includes steps for data preprocessing, model training, hyper-parameter tuning,
model evaluation, and saving the trained model.

"""

import sys
import pandas as pd
from lib.timer import Timer
from lib.data_explorer import DataExplorer
from lib.data_preprocessor import DataPreprocessor
from lib.classifier import Classifier
from params.path import PATH, PREP_METHOD, DIR, CLS_METHOD

FORCE_MODEL_UPDATE = False
STEPS_LIMIT = 3
EXPLORE_RAW_DATA = False
EXPLORE_PREPROCESSED_DATA = False
TESTS_NUMBER = 20

# Start timing
timer = Timer()
timer.start()

# Read the dataset
print('Reading the dataset...')
dataset = pd.read_csv(PATH['dataset'])

# Explore the data
if EXPLORE_RAW_DATA:
    e_timer = Timer('Raw data exploration')
    e_timer.start()
    de = DataExplorer(
        data=dataset,
        save_dir=DIR['raw_data_img']
    )
    de.exploration()
    e_timer.end()
if STEPS_LIMIT <= 1:
    # End program
    timer.end()
    sys.exit()

# Preprocess the data
p_timer = Timer('Data preprocessing')
p_timer.start()
dp = DataPreprocessor(
    data=dataset,
    path=PATH['preprocessed_dataset']
)
dataset = dp.preprocess(PREP_METHOD)
p_timer.end()
if EXPLORE_PREPROCESSED_DATA:
    ep_timer = Timer('Preprocessed data exploration')
    ep_timer.start()
    de = DataExplorer(
        data=dataset,
        save_dir=DIR['preprocessed_data_img'],
        raw_dir=DIR['raw_data_img'],
        prep_method=PREP_METHOD
    )
    de.exploration()
    ep_timer.end()
if STEPS_LIMIT == 2:
    # End program
    timer.end()
    sys.exit()

for i in range(TESTS_NUMBER):
    # Generate classifier and partition the data
    c = Classifier(dataset, CLS_METHOD)
    c.segregation()
```

```
76
77      # Look for an existing model
78      if not c.load_config(PATH['model'],
79                           PATH['parameters']
80                           ) or (FORCE_MODEL_UPDATE and i == 0):
81          # Perform classification
82          g_timer = Timer('Grid Search')
83          g_timer.start()
84          c.grid_search()
85          g_timer.end()
86          # Save the model
87          c.save_config(PATH['model'], PATH['parameters'])
88
89      # Calculate the accuracy of the model
90      c.calculate_accuracy()
91      # Generate data results
92      c.generate_results(DIR['results'], PATH['accuracy'])
93
94  # End timing
95  timer.end()
```

A.4.2 model_explanation.py

Il modulo rappresenta il file eseguibile mediante il quale è possibile visualizzare le counterfactual explanation dei campioni specificati. Di seguito è riportato il codice implementato.

```
1   """
2   This module uses the cfnow library to generate and plot
3   counterfactual explanations for a trained model.
4
5   The cfnow library is a powerful tool for interpreting machine learning models.
6   It provides methods for generating counterfactual explanations, which can help
7   understand how a model makes its predictions. This module uses these capabilities
8   to generate and plot counterfactual explanations for a trained model.
9
10  """
11
12  import sys
13  import pandas as pd
14  from lib.timer import Timer
15  from lib.data_preprocessor import DataPreprocessor
16  from lib.classifier import Classifier
17  from lib.explainer import Explainer
18  from params.path import DIR, PATH, PREP_METHOD, CLS_METHOD
19  from params.attribute_specifications import ATTRIBUTES, CLASS_LABELS
20
21  # Start timing
22  timer = Timer()
23  timer.start()
24
25  # Load the dataset
26  dataset = pd.read_csv(PATH['dataset'])
27
28  # Preprocess the dataset
29  dp = DataPreprocessor(
30      data=dataset,
31      path=PATH['preprocessed_dataset']
32  )
33  dataset = dp.preprocess(PREP_METHOD)
34
35  # Divide dataset in features and targets
36  class_columns = [ATTRIBUTES[sample] for sample in iter(CLASS_LABELS)]
37  X = dataset.drop(columns=class_columns, axis=1)
38  y = dataset[class_columns]
39
40  # Load the model
41  c = Classifier(dataset, CLS_METHOD)
42  if not c.load_config(PATH['model'], PATH['parameters']):
43      print(f'ERROR: Model not found for {CLS_METHOD} with {PREP_METHOD} '
44            'pre-processing method, please execute "model_generation.py" before.')
45      timer.end()
46      sys.exit(1)
47  print('Model successfully loaded.')
48
49  # Sample some data for each class
```

```
SAMPLES = [1, 9, 50, 70, 99, 127, 1000, 1100, 1300, 1910, 1916,
           1919, 1925, 7711, 19840, 27309, 29971, 33962, 39082, 45981]
# for value in dataset['self_valence'].unique():
#     SAMPLES.extend(dataset[dataset['self_valence'] == value].sample(n=10, random_state=42).index)
# Set time limit
TIMEOUT = 30
# Set plot mod
# MOD = ['greedy', 'countershapley']

# Explain data
e = Explainer(SAMPLES, {'X': X, 'y': y}, c.model, TIMEOUT)
e.plot_counterfactuals(DIR['explanation'], CLS_METHOD)

# End timing
timer.end()
```

A.4.3 jaccard_measure.py

Il modulo rappresenta il file eseguibile mediante il quale sono stati calcolati gli indici di Jaccard. Di seguito è riportato il codice implementato.

```
"""
This module contains the script used to compute the Jaccard measure
between two model counterfactuals.

The Jaccard measure, or Jaccard similarity coefficient,
is a statistic used for comparing the similarity and diversity of sample sets.
This module uses this measure to compare the counterfactuals generated by two different models.
"""

import sys
import os
import pandas as pd
from lib.timer import Timer
from lib.data_preprocessor import DataPreprocessor
from lib.classifier import Classifier
from lib.jaccard_evaluer import JaccardEvaluer
from params.path import PATH, CLASSIFIERS, PREP_METHOD, DIR
from params.attribute_specifications import ATTRIBUTES, CLASS_LABELS, DATA_LABELS

# Start timing
timer = Timer()
timer.start()

# Load the dataset
dataset = pd.read_csv(PATH['dataset'])

# Preprocess the dataset
dp = DataPreprocessor(
    data=dataset,
    path=PATH['preprocessed_dataset']
)
dataset = dp.preprocess(PREP_METHOD)

# Divide dataset in features and targets
class_columns = [ATTRIBUTES[sample] for sample in iter(CLASS_LABELS)]
data_columns = [ATTRIBUTES[sample] for sample in iter(DATA_LABELS)]
X = dataset.drop(columns=class_columns, axis=1)
y = dataset[class_columns]

# Load models
clfs = []
for model_type in CLASSIFIERS:
    c = Classifier(dataset, model_type)
    clfs.append(c)
    if not c.load_config(PATH['model'], PATH['parameters']):
        print(f'ERROR: Model not found for {model_type} with {PREP_METHOD} '
              'pre-processing method, please execute "model_generation.py" before.')
        timer.end()
        sys.exit(1)
print('Models successfully loaded.')

# Compute Jaccard values
j = JaccardEvaluer(
    data={'dataset': dataset, 'X': X, 'y': y},
    path=os.path.join(DIR['raw_data'], 'jaccard_indexes.csv')
)
```

```
57 j.sample_data()
58 j.compute_jaccard(clfs=clfs)
59
60 # Plotting results
61 j.plot_jaccard_hist(
62     clfs=clfs,
63     path=os.path.join('img', 'jaccard_measure', 'jaccard_indexes_histogram.png')
64 )
65 j.plot_feature_importance(
66     clfs=clfs,
67     path=os.path.join('img', 'jaccard_measure', 'features_importance.png')
68 )
69
70 # End timing
71 timer.end()
```

Appendice B

Grafici aggiuntivi

In questa appendice vengono visualizzati i grafici che sono stati generati durante l'analisi qualitativa dei controfattuali.

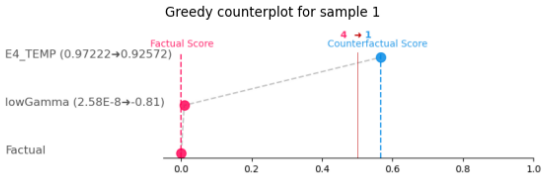


Figura B.1: Grafico greedy per campione n.1 (classificatore MLPC).

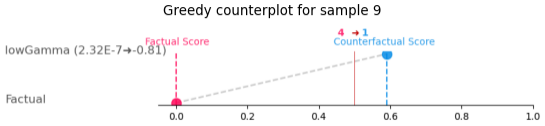


Figura B.2: Grafico greedy per campione n.9 (classificatore MLPC).

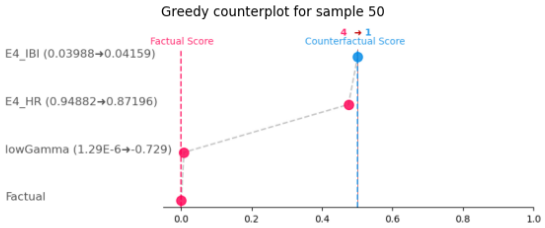


Figura B.3: Grafico greedy per campione n.50 (classificatore MLPC).

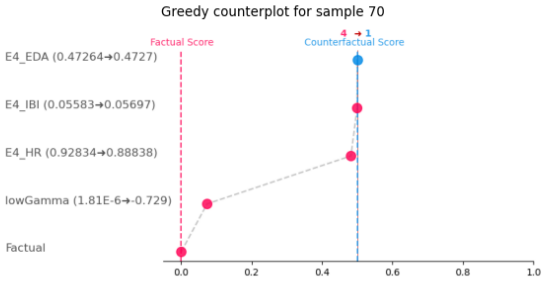


Figura B.4: Grafico greedy per campione n.70 (classificatore MLPC).

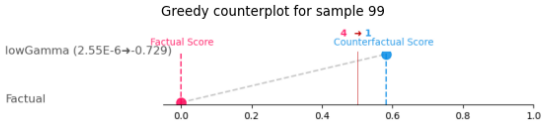


Figura B.5: Grafico greedy per campione n.99 (classificatore MLPC).

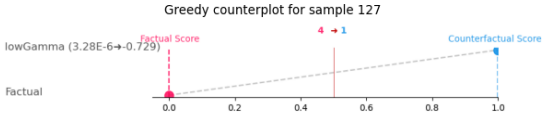


Figura B.6: Grafico greedy per campione n.127 (classificatore MLPC).

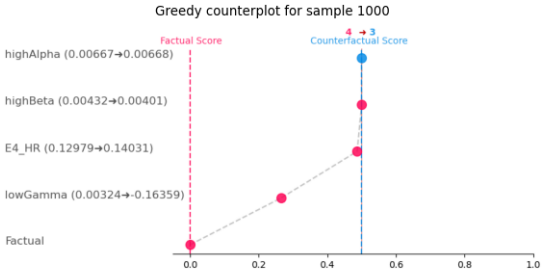


Figura B.7: Grafico greedy per campione n.1000 (classificatore MLPC).

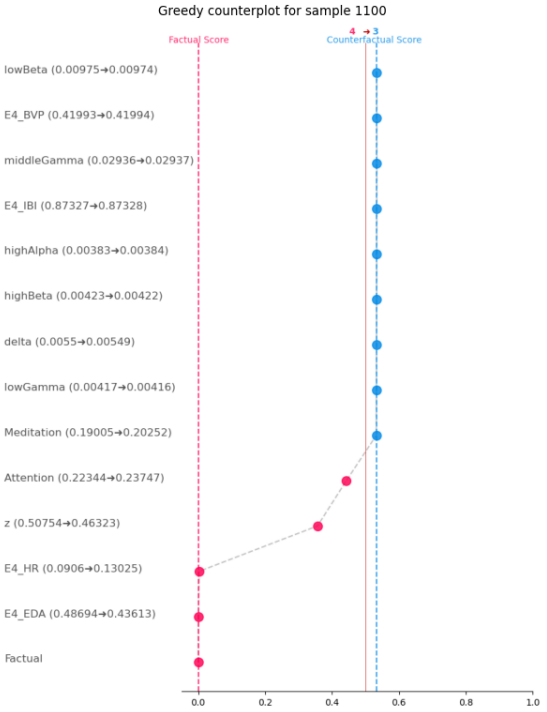


Figura B.8: Grafico greedy per campione n.1100 (classificatore MLPC).

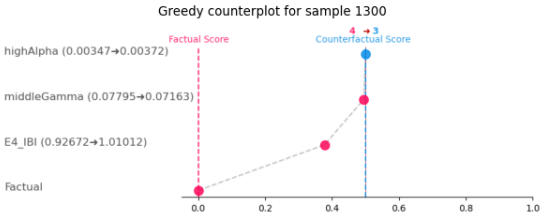


Figura B.9: Grafico greedy per campione n.1300 (classificatore MLPC).

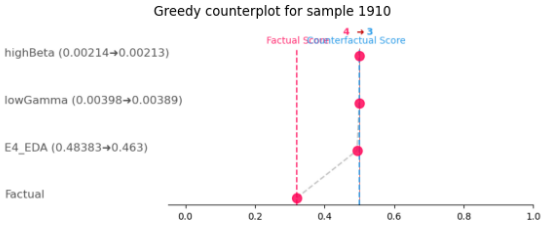


Figura B.10: Grafico greedy per campione n.1910 (classificatore MLPC).

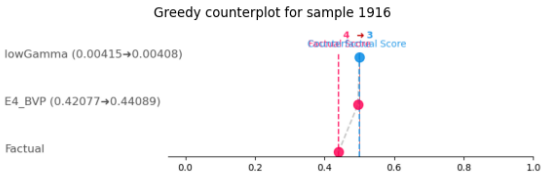


Figura B.11: Grafico greedy per campione n.1916 (classificatore MLPC).

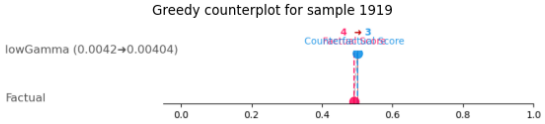


Figura B.12: Grafico greedy per campione n.1919 (classificatore MLPC).

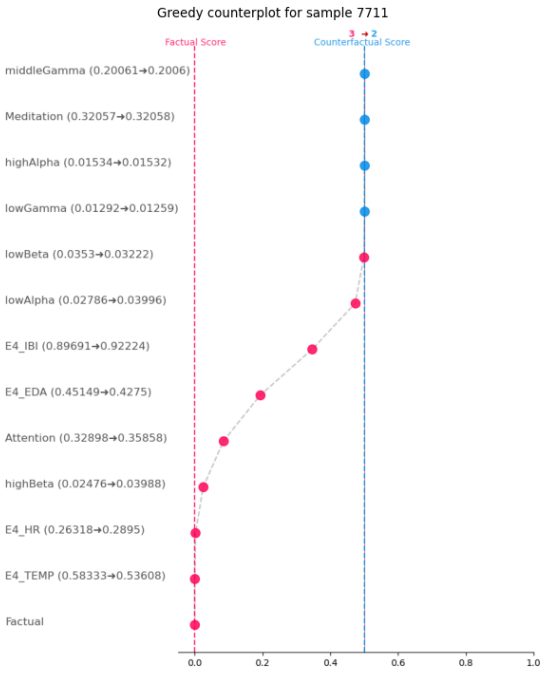


Figura B.13: Grafico greedy per campione n.7711 (classificatore MLPC).

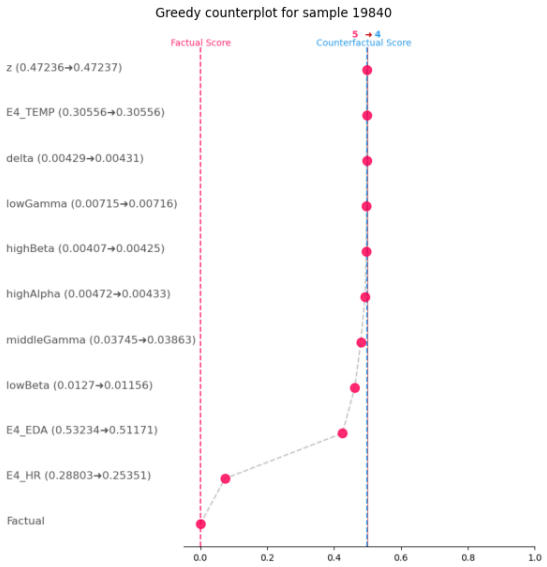


Figura B.14: Grafico greedy per campione n.19840 (classificatore MLPC).

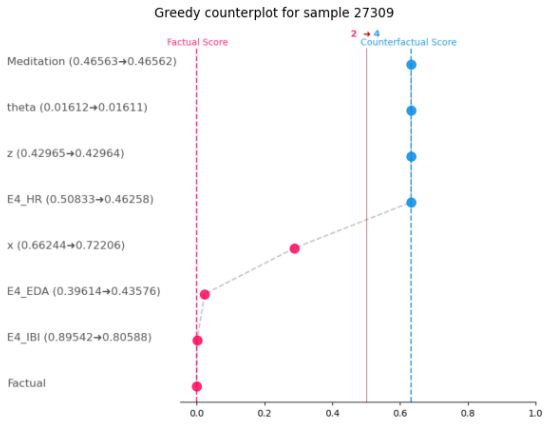


Figura B.15: Grafico greedy per campione n.27309 (classificatore MLPC).

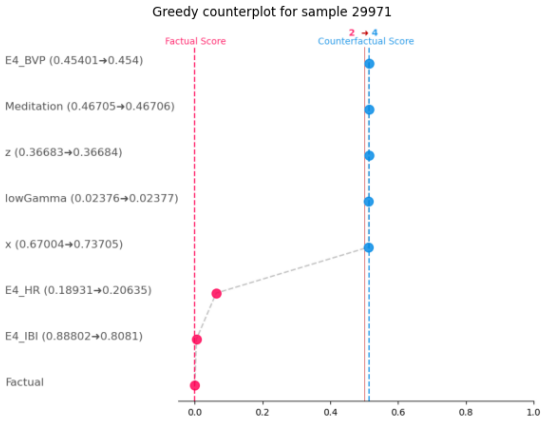


Figura B.16: Grafico greedy per campione n.29971 (classificatore MLPC).

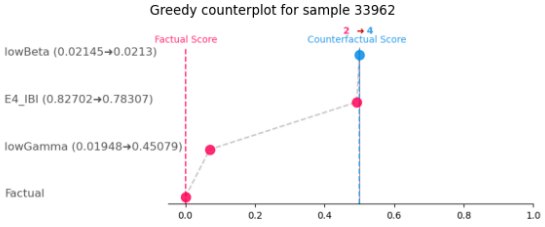


Figura B.17: Grafico greedy per campione n.33962 (classificatore MLPC).

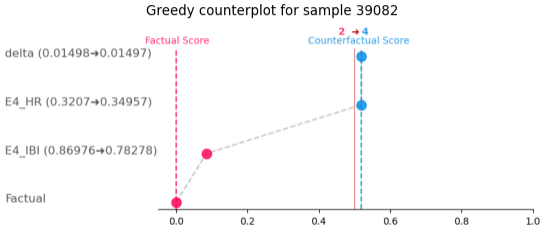


Figura B.18: Grafico greedy per campione n.39082 (classificatore MLPC).

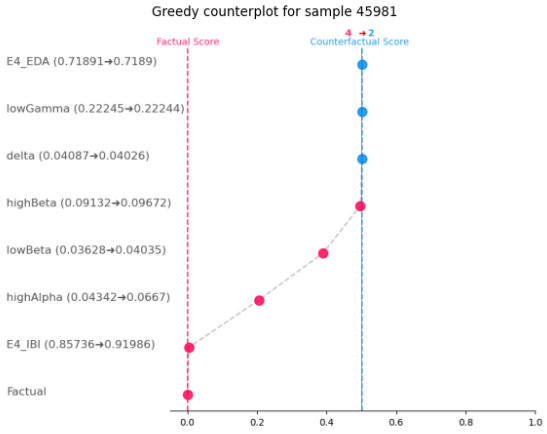


Figura B.19: Grafico greedy per campione n.45981 (classificatore MLPC).

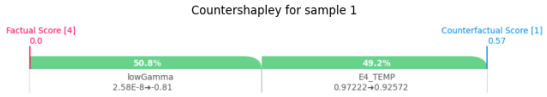


Figura B.20: Grafico counterShapley per campione n.1 (classificatore MLPC).

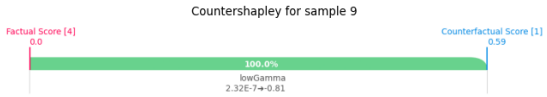


Figura B.21: Grafico counterShapley per campione n.9 (classificatore MLPC).

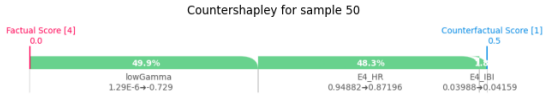


Figura B.22: Grafico counterShapley per campione n.50 (classificatore MLPC).

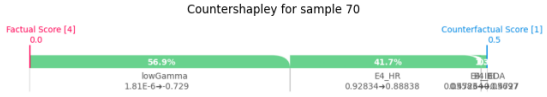


Figura B.23: Grafico counterShapley per campione n.70 (classificatore MLPC).

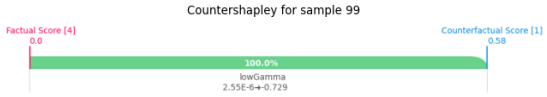


Figura B.24: Grafico counterShapley per campione n.99 (classificatore MLPC).

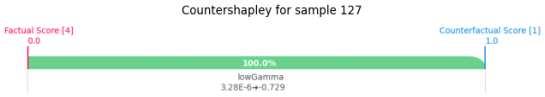


Figura B.25: Grafico counterShapley per campione n.127 (classificatore MLPC).

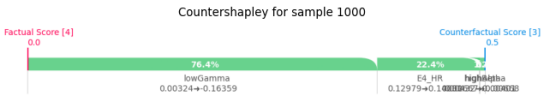


Figura B.26: Grafico counterShapley per campione n.1000 (classificatore MLPC).

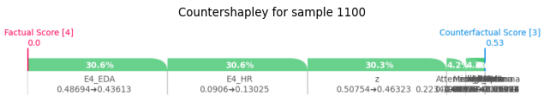


Figura B.27: Grafico counterShapley per campione n.1100 (classificatore MLPC).

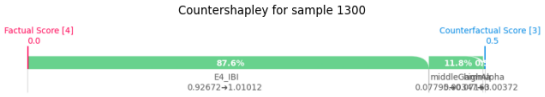


Figura B.28: Grafico counterShapley per campione n.1300 (classificatore MLPC).

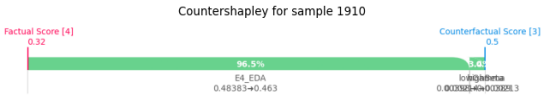


Figura B.29: Grafico counterShapley per campione n.1910 (classificatore MLPC).

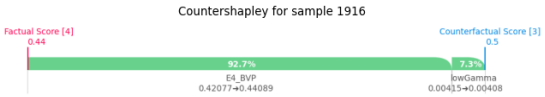


Figura B.30: Grafico counterShapley per campione n.1916 (classificatore MLPC).

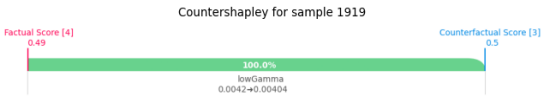


Figura B.31: Grafico counterShapley per campione n.1919 (classificatore MLPC).

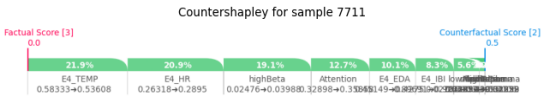


Figura B.32: Grafico counterShapley per campione n.7711 (classificatore MLPC).

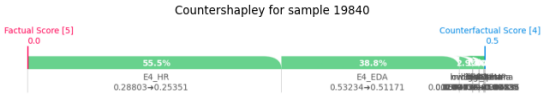


Figura B.33: Grafico counterShapley per campione n.19840 (classificatore MLPC).

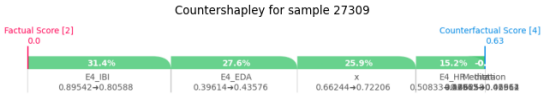


Figura B.34: Grafico counterShapley per campione n.27309 (classificatore MLPC).

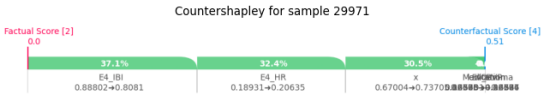


Figura B.35: Grafico counterShapley per campione n.29971 (classificatore MLPC).

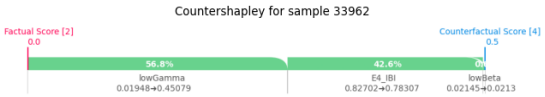


Figura B.36: Grafico counterShapley per campione n.33962 (classificatore MLPC).

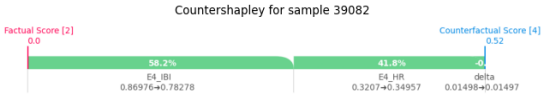


Figura B.37: Grafico counterShapley per campione n.39082 (classificatore MLPC).

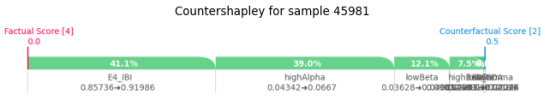


Figura B.38: Grafico counterShapley per campione n.45981 (classificatore MLPC).

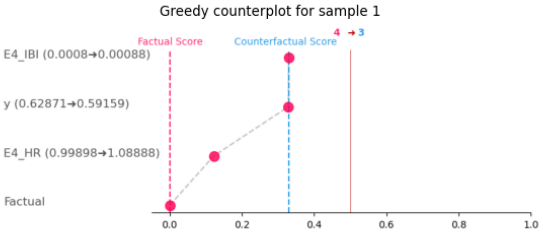


Figura B.39: Grafico greedy per campione n.1 (classificatore SVC).

Figura B.40: Grafico greedy per campione n.9 (classificatore SVC).

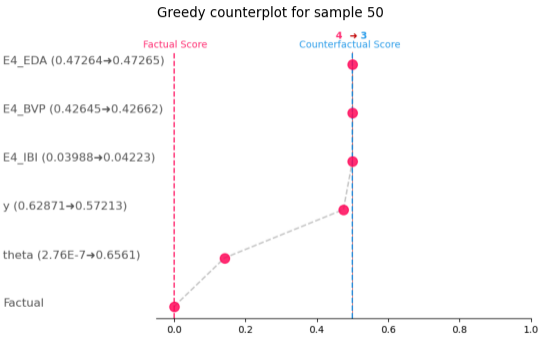


Figura B.41: Grafico greedy per campione n.50 (classificatore SVC).

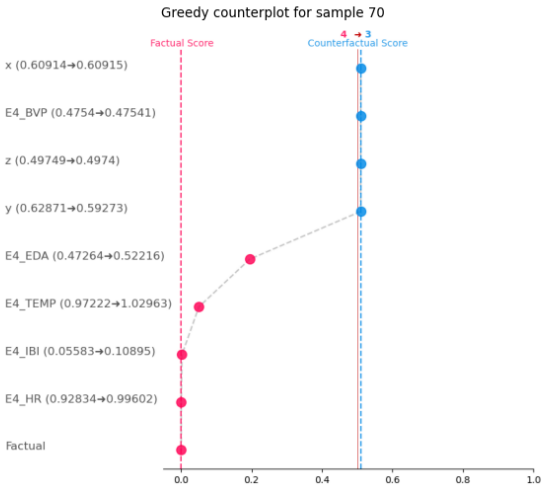


Figura B.42: Grafico greedy per campione n.70 (classificatore SVC).

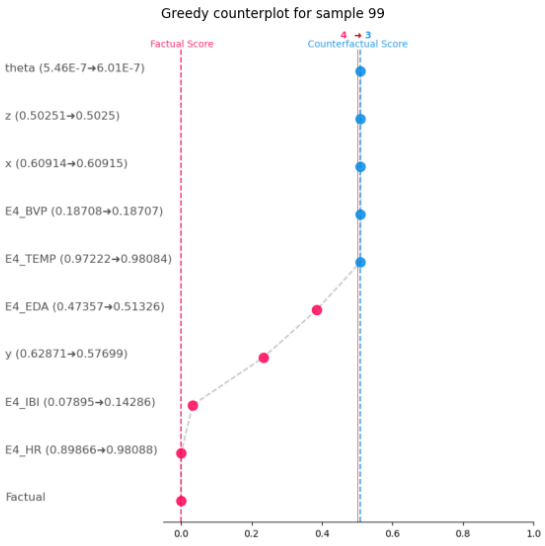


Figura B.43: Grafico greedy per campione n.99 (classificatore SVC).

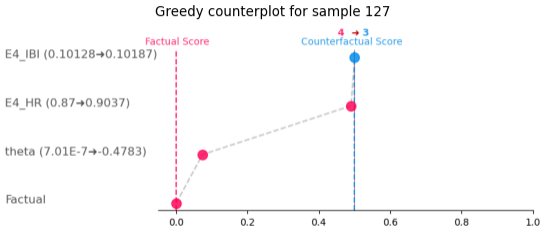


Figura B.44: Grafico greedy per campione n.127 (classificatore SVC).

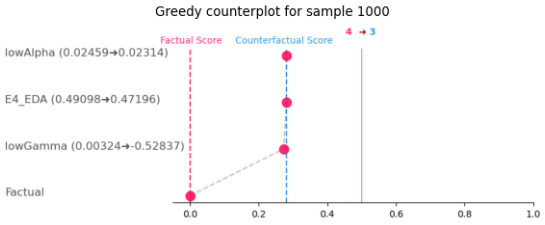


Figura B.45: Grafico greedy per campione n.1000 (classificatore SVC).

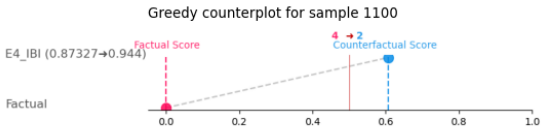


Figura B.46: Grafico greedy per campione n.1100 (classificatore SVC).

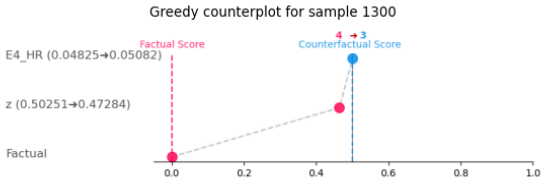


Figura B.47: Grafico greedy per campione n.1300 (classificatore SVC).

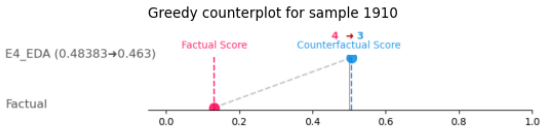


Figura B.48: Grafico greedy per campione n.1910 (classificatore SVC).

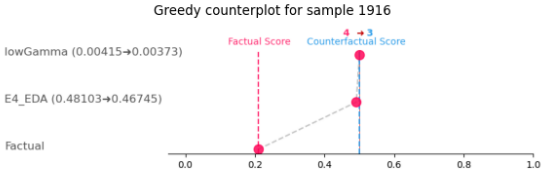


Figura B.49: Grafico greedy per campione n.1916 (classificatore SVC).

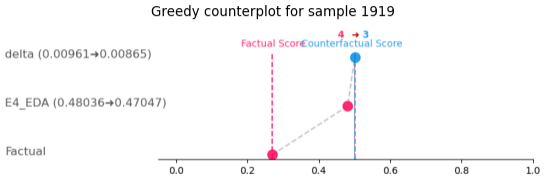


Figura B.50: Grafico greedy per campione n.1919 (classificatore SVC).

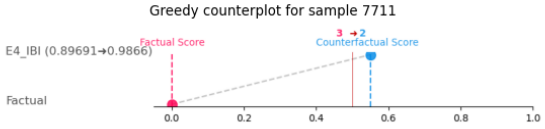


Figura B.51: Grafico greedy per campione n.7711 (classificatore SVC).

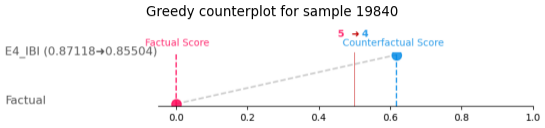


Figura B.52: Grafico greedy per campione n.19840 (classificatore SVC).

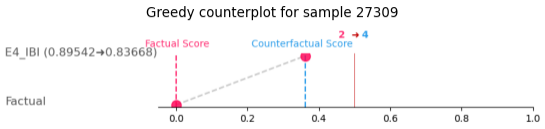


Figura B.53: Grafico greedy per campione n.27309 (classificatore SVC).

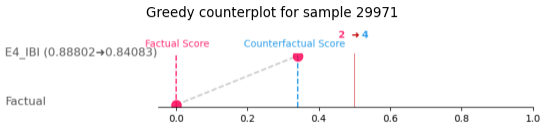


Figura B.54: Grafico greedy per campione n.29971 (classificatore SVC).

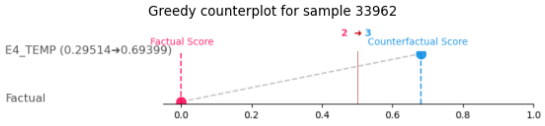


Figura B.55: Grafico greedy per campione n.33962 (classificatore SVC).

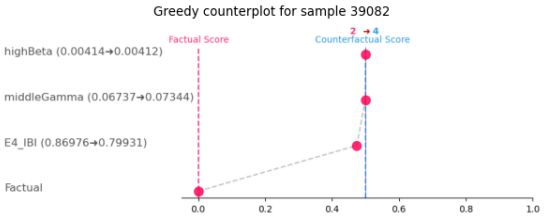


Figura B.56: Grafico greedy per campione n.39082 (classificatore SVC).

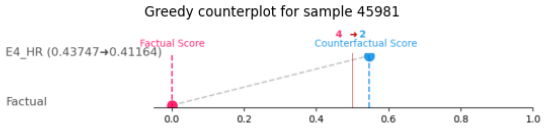


Figura B.57: Grafico greedy per campione n.45981 (classificatore SVC).

Figura B.58: Grafico counterShapley per campione n.1 (classificatore SVC).

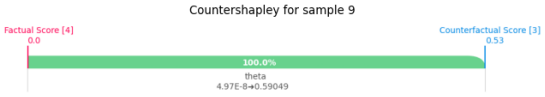


Figura B.59: Grafico counterShapley per campione n.9 (classificatore SVC).

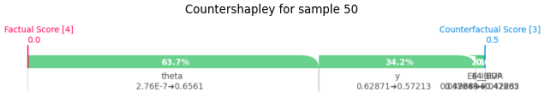


Figura B.60: Grafico counterShapley per campione n.50 (classificatore SVC).

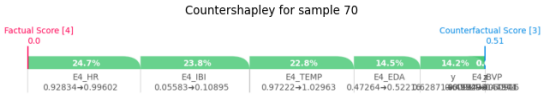


Figura B.61: Grafico counterShapley per campione n.70 (classificatore SVC).

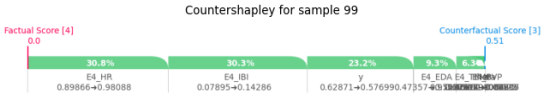


Figura B.62: Grafico counterShapley per campione n.99 (classificatore SVC).

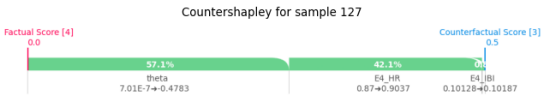


Figura B.63: Grafico counterShapley per campione n.127 (classificatore SVC).

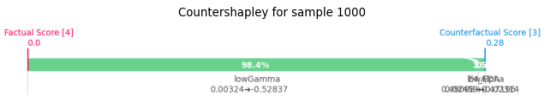


Figura B.64: Grafico counterShapley per campione n.1000 (classificatore SVC).

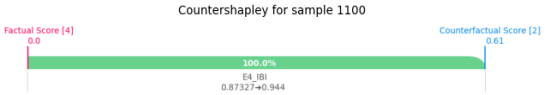


Figura B.65: Grafico counterShapley per campione n.1100 (classificatore SVC).

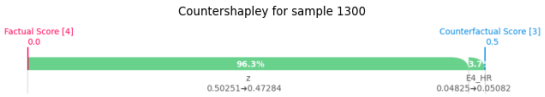


Figura B.66: Grafico counterShapley per campione n.1300 (classificatore SVC).

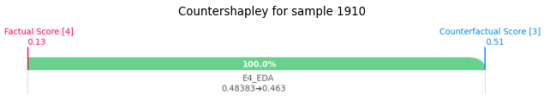


Figura B.67: Grafico counterShapley per campione n.1910 (classificatore SVC).

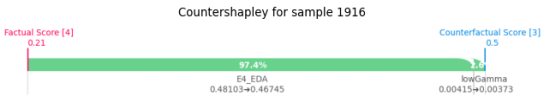


Figura B.68: Grafico counterShapley per campione n.1916 (classificatore SVC).

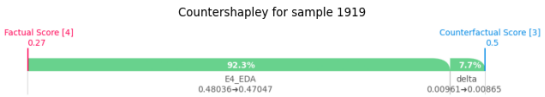


Figura B.69: Grafico counterShapley per campione n.1919 (classificatore SVC).

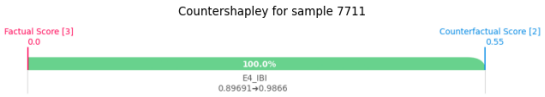


Figura B.70: Grafico counterShapley per campione n.7711 (classificatore SVC).

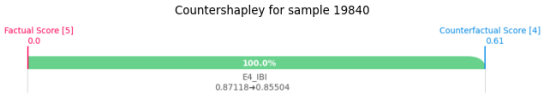


Figura B.71: Grafico counterShapley per campione n.19840 (classificatore SVC).

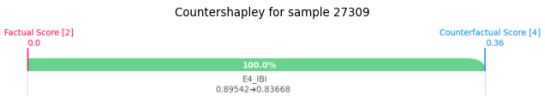


Figura B.72: Grafico counterShapley per campione n.27309 (classificatore SVC).

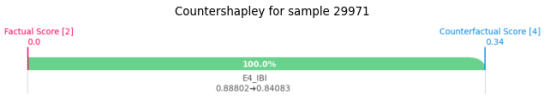


Figura B.73: Grafico counterShapley per campione n.29971 (classificatore SVC).

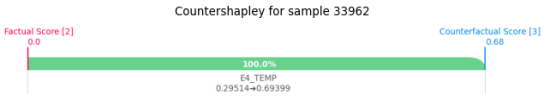


Figura B.74: Grafico counterShapley per campione n.33962 (classificatore SVC).

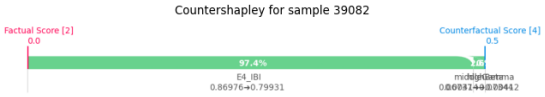


Figura B.75: Grafico counterShapley per campione n.39082 (classificatore SVC).

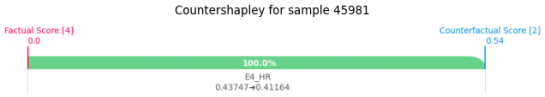


Figura B.76: Grafico counterShapley per campione n.45981 (classificatore SVC).

Bibliografia

[1] Sitara Afzal, Haseeb Ali Khan, Imran Ullah Khan, Md. Jalil Piran, and Jong Weon Lee. A comprehensive survey on affective computing; challenges, trends, applications, and future directions, 2023.

[2] Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing*, 408:189–215, 2020.

[3] Raphael Mazzine Barbosa de Oliveira, Kenneth Sörensen, and David Martens. A model-agnostic and data-independent tabu search algorithm to generate counterfactuals for tabular, image, and text data. *European Journal of Operational Research*, 2023.

[4] Laurence Devillers and Roddy Cowie. Ethical considerations on affective computing: An overview. *Proceedings of the IEEE*, 111(10):1445–1458, 2023.

[5] Alberto Greco, Gaetano Valenza, Luca Citi, and Enzo Pasquale Scilingo. Arousal and valence recognition of affective sounds based on electrodermal activity. *IEEE Sensors Journal*, 17(3):716–725, 2017.

[6] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[7] Roberta Savella Marco Martorana. Ia e riconoscimento delle emozioni: rischi e possibili vantaggi, 2023.

[8] Bjorge Meulemeester, Raphael Mazzine Barbosa De Oliveira, and David Martens. Calculating and visualizing counterfactual feature importance values, 2023.

[9] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, 1991.

[10] Cheul Young Park, Narae Cha, Soowon Kang, Auk Kim, Ahsan Habib Khandoker, Leontios Hadjileontiadis, Alice Oh, Yong Jeong, and Uichin Lee. K-emocon,

a multimodal sensor dataset for continuous emotion recognition in naturalistic conversations. *Scientific Data*, 7(1):293, 2020.

[11] Shahab Rezaei, Sadaf Moharreri, Nader Jafarnia Dabanloo, and Saman Parvaneh. Evaluating valence level of pictures stimuli in heart rate variability response. In *2015 Computing in Cardiology Conference (CinC)*, pages 1057–1060, 2015.

[12] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Erico Tjoa and Cuntai Guan. A survey on explainable artificial intelligence (xai): Toward medical xai. *IEEE Transactions on Neural Networks and Learning Systems*, 32(11):4793–4813, November 2021.

[14] Giulia Vilone and Luca Longo. Explainable artificial intelligence: a systematic review, 2020.

[15] Yan Wang, Wei Song, Wei Tao, Antonio Liotta, Dawei Yang, Xinlei Li, Shuyong Gao, Yixuan Sun, Weifeng Ge, Wei Zhang, and Wenqiang Zhang. A systematic review on affective computing: emotion models, databases, and recent advances. *Information Fusion*, 83-84:19–52, 2022.