UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura

Corso di Laurea in Ingegneria Informatica, Elettronica e delle Telecomunicazioni

Classificazione di segnali mediante programmazione genetica modulare

Relatore:
Prof. Stefano Cagnoni

Correlatore:
Dott.ssa Giulia Magnani

Tesi di Laurea di:
Arianna Cella

ANNO ACCADEMICO 2022-2023

Ringraziamenti

Ringrazio il professor Stefano Cagnoni e la dottoressa Giulia Magnani per la loro disponibilità durante il mio lavoro di tesi. La loro guida e competenza sono state fondamentali per questo progetto di ricerca.
Un sincero ringraziamento va alla mia famiglia, al mio fidanzato Matteo e a tutti i miei amici. In particolare, desidero ringraziare i miei compagni di corso per aver reso questa esperienza accademica più leggera e stimolante.

Indice

Introduzione

La classificazione dei segnali è un campo di ricerca in continua crescita che riveste un'importanza significativa in diversi settori scientifici e in applicazioni pratiche. I segnali sono sequenze di dati provenienti da diverse fonti, come sensori, dispositivi medici, reti di comunicazione e molti altri, e la loro classificazione, che consiste nell'identificare e assegnare una categoria o una classe a ciascun segnale, viene effettuata in base alle loro caratteristiche o a pattern distintivi.

Nel contesto della medicina e della sanità, la classificazione dei segnali svolge un ruolo vitale nella diagnosi e nel monitoraggio di malattie e disfunzioni. Ad esempio, l'analisi dei segnali ECG consente agli esperti medici di riconoscere e distinguere tra diversi tipi di aritmie o anomalie del ritmo cardiaco, fornendo una diagnosi accurata e consentendo trattamenti mirati.

Oltre che nella medicina, la classificazione dei segnali ha un impatto significativo in altri ambiti come l'elaborazione del linguaggio naturale, il riconoscimento vocale, la sicurezza e il riconoscimento di pattern. Ad esempio, attraverso la classificazione dei segnali audio, è possibile identificare parole o fonemi specifici, consentendo la trascrizione automatica o il riconoscimento dei comandi vocali. Nell'ambito della sicurezza e della sorveglianza, la classificazione dei segnali può rilevare eventi sospetti o comportamenti anomali da sensori o sistemi di videosorveglianza, contribuendo così alla protezione delle persone e degli ambienti.

L'intelligenza artificiale e l'apprendimento automatico hanno rivoluzionato il campo della classificazione dei segnali, aprendo nuove prospettive e con-

sentendo progressi significativi. Grazie all'impiego di algoritmi e modelli di machine learning, è possibile analizzare segnali di rilevante durata e dimensione e di individuare al loro interno pattern complessi che potrebbero essere difficili da rilevare manualmente o attraverso metodi tradizionali.
Una delle forze dell'apprendimento automatico nella classificazione dei segnali risiede nella sua capacità di generalizzare le conoscenze acquisite durante il processo di addestramento. Dopo aver appreso da un insieme di dati di addestramento etichettati, il modello può essere utilizzato per classificare nuovi dati che non sono mai stati visti prima. Questa generalizzazione consente di estendere l'applicazione dei modelli di classificazione dei segnali a nuovi scenari o a nuovi dati di test, fornendo previsioni e risultati affidabili.

Descrizione e obiettivo dell'elaborato

L'elaborato descrive lo studio e lo sviluppo di un algoritmo basato sulla programmazione genetica modulare per la classificazione dei segnali.
Il progetto ha due obiettivi. Il primo è utilizzare la programmazione genetica (GP) per effettuare una preelaborazione dei segnali al fine di migliorare le prestazioni di un classificatore random forest. Il secondo obiettivo è realizzare un approccio modulare basato sulla GP per realizzare questa pre-elaborazione. In particolare, si intende implementare una versione di GP modulare che possa semplificare l'implementazione su FPGA del sistema, rendendo più compatte sia la ricerca che la descrizione degli alberi del sistema grazie all'utilizzo di moduli che implementano funzioni di alto livello.
La programmazione genetica rientra nel campo degli algoritmi evolutivi ed è una tecnica di ottimizzazione ispirata ai principi dell'evoluzione biologica. Consiste nel far evolvere una popolazione iniziale di individui, ognuno rappresentante una funzione che risolve un problema dato, ottenendo generazione dopo generazione soluzioni sempre migliori attraverso la selezione dei migliori individui e l'applicazione degli operatori di crossover e mutazione. La selezione si basa sulla valutazione del grado di adattamento di ciascun in-

dividuo tramite una funzione di fitness che misura il livello di soddisfazione rispetto agli obiettivi desiderati. In questo caso specifico, l'obiettivo è trovare una trasformazione del segnale che permetta di ottimizzare le prestazioni di un classificatore Random Forest applicato ai dati trasformati. Il programma genetico rappresentato da ogni individuo viene utilizzato per trasformare il training set, che addestra il modello di classificazione Random Forest, e il validation set, che valuta il modello addestrato.
Come esempio di applicazione di questo modello, il training set e validation set sono stati generati a partire dai dataset Digit e da un dataset contenente segnali ECG, di cui parlerò nel dettaglio nei capitoli successivi.
La metrica di valutazione usata è la F1-score, che rappresenta la fitness dell'individuo e deve essere massimizzata nel corso delle iterazioni.
L'aspetto chiave del progetto è l'approccio modulare che viene implementato analizzando gli individui della popolazione e selezionando i sottomoduli di profondità 1 e 2 più frequenti. Tra questi sottomoduli vengono selezionati quelli con una fitness più elevata, che vengono poi inseriti nel set delle primitive utilizzate per generare gli individui delle popolazioni successive.
Per agevolare l'utente nell'esecuzione dell'algoritmo è stata sviluppata un'interfaccia grafica che consente di impostare i parametri principali, come il numero di generazioni, il numero di individui della popolazione, la dimensione del kernel, la massima profondità degli alberi dei programmi genetici e in numero di iterazione dell'algoritmo genetico. Inoltre, l'utente può caricare il dataset su cui desidera lavorare e avviare l'esecuzione dell'algoritmo. Una volta completata l'esecuzione, i risultati saranno disponibili in un file di testo specifico. Questo consente all'utente di esaminare e valutare i risultati in modo semplice e ordinato.
Nell'elaborato verrà affrontata la teoria alla base dell'algoritmo, saranno presentati gli strumenti utilizzati, come le librerie e i dataset, verrà spiegato dettagliatamente l'algoritmo realizzato e, infine, saranno presentati e discussi i risultati ottenuti.

Capitolo 1

Evolutionary Computation

Il calcolo evolutivo (EC) è un campo dell'informatica che si propone di imitare i processi dell'evoluzione naturale al fine di risolvere problemi di ingegneria, ottimizzando una funzione obiettivo.[1]
L'EC comprende diversi modelli computazionali che simulano tali processi, con il concetto chiave della sopravvivenza del più adatto. L'idea di base è attuare un processo iterativo, chiamato evoluzione artificiale, che, a partire da un insieme iniziale di soluzioni molto approssimate o generate casualmente, porta a soluzioni di un dato problema che migliorano nel corso delle generazioni fino a quando non convergono ad un ottimo.

1.1 Algoritmo Evolutivo

Gli algoritmi evolutivi sono una tecnica dell'EC che simula l'evoluzione naturale sfruttando concetti come cromosomi, geni, fitness, selezione, crossover e mutazione per affrontare in modo efficiente e automatizzato problemi di ottimizzazione e ricerca.

1.1.1 Descrizione dei termini chiave

- **Cromosoma**: rappresenta la struttura genetica di un individuo, ovvero l'insieme dei geni che ne determinano le caratteristiche. Nel contesto

degli algoritmi evolutivi, il cromosoma rappresenta la soluzione di un problema ed è costituito da una sequenza di elementi corrispondenti ai geni.

- **Gene**: rappresenta l'unità di base del cromosoma, ovvero la porzione di DNA che codifica una specifica caratteristica dell'individuo. Nel contesto degli algoritmi evolutivi, il gene codifica una parte della soluzione del problema.

- **Fitness function**: rappresenta la funzione che viene utilizzata per valutare la bontà delle soluzioni proposte dagli algoritmi evolutivi. In altre parole, la fitness function assegna un punteggio alle soluzioni, in base alla loro capacità di risolvere il problema in questione.

- **Selezione**: rappresenta la fase dell'algoritmo evolutivo in cui si scelgono gli individui migliori, ovvero quelli con punteggio di fitness più elevato, per generare i nuovi individui della popolazione.

- **Crossover**: rappresenta la fase dell'algoritmo evolutivo in cui si combinano due cromosomi di due individui diversi per generare uno o più nuovi individui.

- **Mutazione**: rappresenta la fase dell'algoritmo evolutivo in cui si modificano casualmente alcuni geni del cromosoma, al fine di aumentare la diversità genetica della popolazione e di evitare la convergenza prematura verso soluzioni subottimali.

1.1.2 Algoritmo evolutivo generico

Gli algoritmi evolutivi EA usano una popolazione di individui (cromosomi) dove gli individui con le migliori capacità di sopravvivenza hanno maggiore probabilità di riprodursi e la prole viene generata combinando parte dei genitori o mutando un individuo della popolazione. La capacità di sopravvivenza di un individuo viene misurata da una funzione di fitness che riflette i vincoli

e gli obiettivi del problema che bisogna risolvere. L'evoluzione attraverso la selezione naturale di una popolazione di individui scelti a caso porta a definire un algoritmo evolutivo come una ricerca stocastica di una soluzione ottima a un dato problema all'interno dello spazio dei possibili cromosomi che la codificano.
Un tipico algoritmo evolutivo è composto dai seguenti passaggi:

1. Inizializzazione della popolazione di soluzioni
2. Valutazione della fitness di ogni membro della popolazione
3. Inizio di un ciclo che termina quando si soddisfa un criterio di arresto. In questo ciclo viene selezionato un sottoinsieme della popolazione in base alla fitness, da cui viene generata una nuova popolazione con gli operatori di riproduzione e in cui viene valutata la fitness.

Il processo di ricerca evolutiva è influenzato dalle seguenti componenti principali di un algoritmo evolutivo: codifica delle soluzioni possibili, funzione di fitness, popolazione con la sua strategia di inizializzazione, operatori di selezione, operatori di riproduzione e, infine, la condizione di terminazione. I diversi modi in cui vengono implementate le componenti dell'algoritmo evolutivo portano a diversi paradigmi di calcolo evolutivo:

- **algoritmi genetici**: modellano l'evoluzione genetica
- **programmazione genetica**, basata su algoritmi genetici in cui gli individui sono programmi rappresentati come alberi sintattici
- **programmazione evolutiva**, derivata dalla simulazione del comportamento adattivo nell'evoluzione
- **strategie di evoluzione**, con lo scopo di modellare parametri strategici che controllano la variazione dell'evoluzione
- **evoluzione differenziale**, come gli algoritmi genetici ma con un differente meccanismo di riproduzione

- **evoluzione culturale**, che modella l'evoluzione della cultura di una popolazione e il modo in cui la cultura influenza l'evoluzione genetica e fenotipica degli individui.

- **co-evoluzione**, dove individui inizialmente "stupidi" evolvono attraverso la cooperazione, o la competizione tra loro, acquisendo le caratteristiche necessarie per sopravvivere.

Tra queste diverse classi di algoritmi evolutivi, quella rilevante per questo lavoro di tesi è la programmazione genetica.

1.2 Algoritmo Genetico

Gli algoritmi genetici (GA) operano sulle codifiche delle soluzioni per ottenere nuove soluzioni. Le soluzioni sono rappresentate come stringhe di simboli (cromosomi) e vengono manipolate attraverso operatori genetici come la selezione, il crossover e la mutazione.
L'algoritmo genetico richiede la definizione di un meccanismo di decodifica da genotipo a fenotipo per produrre una soluzione valutabile. Il genotipo rappresenta il codice genetico di un individuo all'interno dell'algoritmo genetico, ovvero la sua rappresentazione come stringa di simboli, ma l'obiettivo finale dell'algoritmo genetico è quello di trovare una soluzione efficace a un determinato problema, e questa soluzione deve essere rappresentata come fenotipo, ovvero la manifestazione fisica dei caratteri codificati nel genotipo. Pertanto, è necessario definire un meccanismo di decodifica che possa tradurre il genotipo in una forma fenotipica valutabile dall'algoritmo, in modo che la sua efficacia possa essere valutata e confrontata con le altre soluzioni presenti nella popolazione.
L'algoritmo genetico di base inizia con la generazione di una popolazione casuale di individui, ognuno rappresentato da un cromosoma; successivamente si entra in un ciclo in cui si decodifica ogni cromosoma dell'individuo, si valuta la fitness di ogni individuo e si selezionano quelli più adatti per la

riproduzione attraverso gli operatori genetici. Questo processo di selezione, crossover e mutazione porta alla creazione di una nuova generazione di individui con caratteristiche sempre più adatte alla soluzione del problema.

1.2.1 Componenti di un algoritmo genetico

1.2.1.1 Rappresentazione (il cromosoma)

In natura, gli organismi hanno determinate caratteristiche che influenzano la loro capacità di sopravvivere e di riprodursi. Queste caratteristiche sono rappresentate da lunghe stringhe di informazioni contenute nei cromosomi dell'organismo. I cromosomi sono strutture di molecole di DNA intrecciate compatte, che si trovano nel nucleo delle cellule organiche. Ogni cromosoma contiene un gran numero di geni, dove un gene è l'unità di eredità. I geni determinano molti aspetti dell'anatomia e della fisiologia e individuo ha una sequenza unica di geni. Una forma alternativa di un gene viene definita allele.

Nel contesto dell'EC, ogni individuo rappresenta una soluzione candidata a un problema di ottimizzazione e le caratteristiche di un individuo sono rappresentate da un cromosoma, denominato genoma. Queste caratteristiche si riferiscono alle variabili del problema di ottimizzazione, per le quali si cerca un'assegnazione ottima. Ogni variabile che deve essere ottimizzata viene definita gene e l'assegnazione di un valore dal dominio consentito della variabile corrispondente viene definita allele.

Un passo importante nella progettazione di un algoritmo evolutivo (EA) è trovare una rappresentazione appropriata delle soluzioni candidate (cioè i cromosomi), poiché la rappresentazione codifica i parametri che caratterizzano le diverse soluzioni. La rappresentazione è strettamente correlata al tipo di problema che si vuole risolvere: la maggior parte degli EA rappresenta le soluzioni come vettori di un tipo di dati specifico, più spesso stringhe binarie o bit (fenotipo). Un'eccezione è la programmazione genetica (GP) in cui gli individui sono rappresentati da un albero.

1.2.1.2 Funzione di fitness

Per determinare la capacità di sopravvivenza di un individuo di un EA, viene utilizzata una funzione matematica per quantificare quanto è buona la soluzione rappresentata da un cromosoma. La funzione di fitness rappresenta la funzione obiettivo che deve misurare l'efficacia di una soluzione e soddisfare le seguenti ipotesi fondamentali:

1. Esiste una misura Q per la qualità di una soluzione.
2. Q è positivo e deve essere massimizzato
3. La Q di un individuo è la sua fitness

La funzione di fitness riveste un ruolo fondamentale poiché viene comunemente utilizzata dagli operatori evolutivi, come la selezione, il crossover e la mutazione. Ad esempio, gli operatori di selezione tendono a scegliere gli individui più adatti per il crossover, mentre tendono a selezionare quelli meno adatti per la mutazione.

1.2.1.3 Popolazione

Una popolazione è un multi-insieme, cioè un insieme che ammette la presenza di più copie dello stesso elemento (individuo, cioè un punto nello spazio di ricerca). Il primo passo nell'applicazione di un EA per risolvere un problema di ottimizzazione è generare una popolazione iniziale. Il modo standard di generare una popolazione iniziale consiste nell'assegnare un valore casuale dal dominio consentito a ciascuno dei geni di ciascun cromosoma. L'obiettivo della selezione casuale è garantire che la popolazione iniziale sia una rappresentazione uniforme dell'intero spazio di ricerca.
La dimensione della popolazione iniziale ha conseguenze in termini di complessità computazionale e capacità di esplorazione. Un gran numero di individui aumenta la diversità, migliorando così le capacità di esplorazione della popolazione. Tuttavia, maggiore è il numero di individui, maggiore è la complessità computazionale per generazione: il tempo di esecuzione per

generazione aumenta, ma può darsi che siano necessarie meno generazioni per individuare una soluzione accettabile. Una piccola popolazione, invece, rappresenterà una piccola parte dello spazio di ricerca. Sebbene la complessità temporale per generazione sia bassa, l'EA potrebbe aver bisogno di più generazioni per convergere rispetto a una popolazione numerosa.

1.2.1.4 Selezione

La selezione è uno dei principali operatori negli EA e l'obiettivo principale degli operatori di selezione è di enfatizzare le soluzioni migliori. La selezione negli algoritmi genetici ha tre fasi principali:

- Selezione dei genitori: in questa fase, gli individui migliori vengono selezionati dalla popolazione corrente per agire da genitori nella fase successiva di riproduzione.
- Riproduzione: dopo la fase di selezione dei genitori, gli individui selezionati vengono combinati attraverso operatori di crossover e/o mutazione per creare la progenie.
- Selezione di una nuova popolazione: alla fine di ogni generazione viene selezionata una nuova popolazione di soluzioni candidate che costituisca la popolazione nella generazione successiva, cercando di garantire la sopravvivenza dei migliori individui.

Tra i possibili operatori di selezione, quelli più comunemente utilizzati sono:

- **Proportional Selection**: ad ogni individuo viene assegnata una probabilità di essere selezionato proporzionale alla sua fitness. La selezione proporzionata alla fitness viene implementata facilmente, ma potrebbe portare a problemi quali la convergenza prematura, che si riscontra quando la fitness di un individuo è molto superiore alla fitness media della popolazione e quindi viene selezionata ripetutamente generando una popolazione uniforme mediocre, e la stagnazione, quando tutti

gli individui hanno una fitness simile (probabilità uguale) rendendo la strategia di ottimizzazione una ricerca casuale.

- **Rank Selection**: gli individui vengono ordinati in base alla fitness e a ciascun individuo viene attribuita una posizione in classifica (rank): viene definita una funzione di distribuzione di probabilità, decrescente con il rank, indipendente dai valori di fitness perché dipende solo dal posizionamento degli individui nella popolazione. Sarà computazionalmente più pesante, ma in questo modo non possiamo avere convergenza prematura, dato che nessun individuo avrà probabilità di selezione molto più elevata di qualsiasi altro, e nessuna stagnazione perché la distribuzione di probabilità non cambia e quindi non potrà mai essere uniforme.

- **Tournament Selection**: per selezionare ogni individuo, viene scelto un sottoinsieme casuale della popolazione di dimensioni predefinite e viene selezionato il migliore. Con la Tournament Selection si crea il "mating pool" (piscina di accoppiamento) da cui vengono poi estratti i genitori con probabilità uniforme. I vantaggi sono quelli della rank selection, ma evita l'ordinamento.

- **Elitist Selection**: viene selezionata almeno una copia degli individui migliori, in questo modo non vengono perse le soluzioni buone, ma potrebbe causare convergenza prematura. I migliori individui vengono mantenuti nella Hall of Fame, una "statistica" esterna all'algoritmo che consente di conservare i migliori individui apparsi in tutto il processo evolutivo.

1.2.1.5 Operatori di riproduzione

Gli operatori di riproduzione sono utilizzati negli algoritmi genetici per generare nuove soluzioni dalla combinazione di due o più soluzioni esistenti. I due operatori di riproduzione più comuni sono il crossover (incrocio) e la mutazione. Il crossover è il processo di creazione di uno o più nuovi individui

attraverso la combinazione di materiale genetico selezionato casualmente da due o più genitori. Come la riproduzione sessuata in natura, il crossover genera nuovi individui il cui codice genetico deriva in parte da un genitore e in parte dall'altro. Ad esempio, nel caso di una soluzione rappresentata da una stringa binaria, il crossover può essere eseguito scegliendo un punto casualmente e scambiando le sezioni di destra o di sinistra (One Point Crossover), oppure vengono eseguiti due "tagli" e le sezioni interne o esterne vengono scambiate (Two Point Crossover).

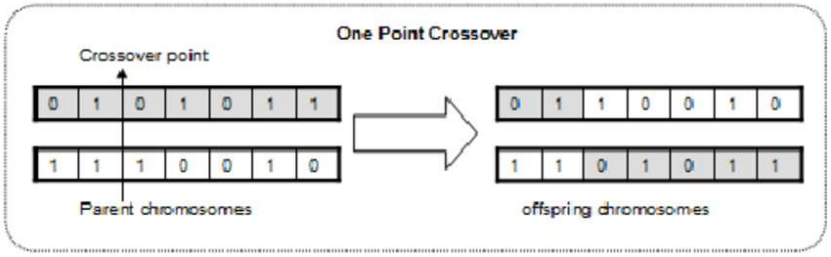

Figura 1.1: Esempio di One Point Crossover [2]

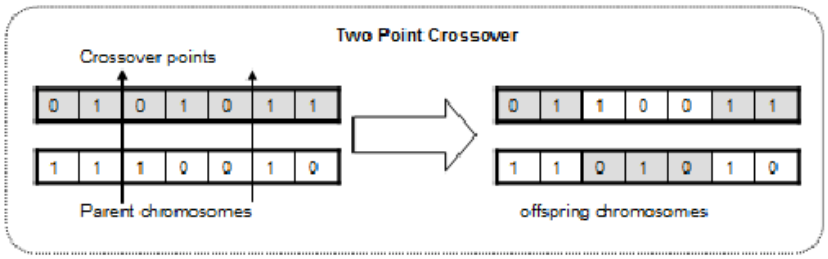

Figura 1.2: esempio di Two Point Crossover [3]

La mutazione è il processo di modifica casuale dei valori dei geni in un cromosoma. L'obiettivo principale della mutazione è introdurre nuovo materiale genetico nella popolazione, aumentando così la diversità genetica e cercando di esplorare regioni dello spazio delle soluzioni che non sono raggiungibili tramite la selezione e la crossover. La mutazione è spesso implementata come una piccola perturbazione di un singolo gene, come ad esempio invertire il valore binario di un bit o aggiungere o sottrarre un piccolo valore a un gene numerico.

1.2.2 Condizione di arresto

Gli operatori evolutivi vengono applicati in modo iterativo in un EA fino a quando non viene soddisfatta una condizione di arresto. Solitamente, la condizione di arresto più comune e semplice consiste nel limitare il numero di generazioni che l'algoritmo può eseguire o nel limitare il numero di valutazioni della funzione di fitness. Essendo però il processo evolutivo un processo di ottimizzazione in cui è difficile stabilire a priori la velocità e l'accuratezza con cui la popolazione tenderà verso soluzioni ottimali, spesso vengono imposti altri criteri di terminazione:

- Si può terminare l'algoritmo quando non si osserva alcun miglioramento per un certo numero di generazioni consecutive, che può essere rilevato monitorando la fitness dell'individuo migliore.
- Si può terminare l'algoritmo se non vi è alcun miglioramento significativo in un determinato intervallo di tempo.
- Si può terminare l'algoritmo quando non vi è alcun cambiamento nella popolazione o quando è stata trovata una soluzione accettabile (fitness maggiore di una soglia prefissata).

1.3 Programmazione genetica

Nel campo dell'intelligenza artificiale uno degli obiettivi più rilevanti è far si che i computer risolvano automaticamente i problemi. La programmazione genetica (GP) è una tecnica di calcolo evolutivo (chiamato anche algoritmo evolutivo) che risolve automaticamente i problemi senza richiedere all'utente di conoscere o specificare la forma o la struttura della soluzione.[4]
Nella programmazione genetica facciamo evolvere una popolazione di programmi per computer: generazione dopo generazione, GP trasforma stocasticamente popolazioni di programmi in nuove popolazioni di programmi auspicabilmente migliori valutando quanto bene funziona un programma eseguendolo e poi confrontando il suo comportamento con un comportamento

ideale. Questo confronto viene quantificato per dare un valore numerico chiamato fitness. I programmi che si comportano bene vengono scelti per essere riprodotti e produrre nuovi programmi per la generazione successiva.
Le principali operazioni genetiche utilizzate per creare nuovi programmi da quelli esistenti sono

- Crossover: La creazione di un programma figlio ottenuto combinando in modo casuale parti scelte casualmente da due programmi genitori selezionati.
- Mutazione: La creazione di un nuovo programma figlio ottenuto modificando in modo casuale una parte scelta casualmente di un programma genitore selezionato.

1.3.1 Elementi base

Nella programmazione genetica i programmi vengono rappresentati come alberi sintattici. Le variabili e le costanti del programma sono le foglie dell'albero e vengono chiamate "terminali", mentre le operazioni aritmetiche sono i nodi interni e vengono chiamati "funzioni". L'insieme di terminali e funzioni determinano l'insieme delle primitive di un sistema GP. Le primitive GP possono accettare anche un numero variabile di argomenti (arità), ma ormai è comune usare un numero fisso di argomenti delle funzioni.

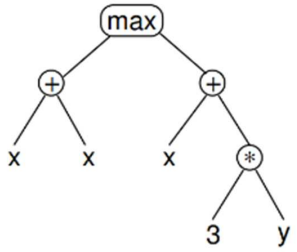

Figura 1.3: Esempio di albero sintattico GP che rappresenta il programma $max(x+x, x+3y)$

Ad esempio, in figura 1.3 viene rappresentata la funzione $max(x + x,$

$x + 3y$). Le variabili e le costanti $(x, y, 3)$ sono le foglie dell'albero, ovvero i terminali, mentre le operazioni aritmetiche $(+, *, max)$ sono nodi interni, chiamati funzioni.

1.3.2 Inizializzazione della popolazione

Solitamente gli individui nella popolazione iniziale vengono generati casualmente; ci sono diversi approcci per farlo: i principali sono il metodo full, il metodo grow e una combinazione dei due nota come ramped half-and-half. In questi algoritmi gli individui iniziali sono generati in modo da non superare una profondità massima, ovvero il numero di archi che servono per arrivare al terminale più profondo partendo dal nodo radice.
Nel **metodo full** vengono generati alberi in cui tutte le foglie si trovano alla stessa profondità, ma questo non implica che tutti gli alberi iniziali abbiano un numero identico di nodi (dimensione dell'albero). Questo accade quando tutte le funzioni dell'insieme delle primitive hanno un'arità uguale. Il metodo consiste nel prendere a caso i nodi dall'insieme di funzioni fino a quando non si raggiunge la profondità massima dell'albero; a quel punto possono essere scelti solo terminali.
Nel **metodo grow** si possono creare alberi di dimensioni varie e i nodi vengono selezionati dall'intero insieme delle primitive fino a quando un ramo non raggiunge il limite di profondità; a quel punto possono essere aggiunti solo terminali.
Il **metodo Ramped Half-and-Half** consiste nel creare metà della popolazione iniziale con il metodo full e l'altra metà con il metodo grow. In questo metodo vengono usati una gamma di limiti di profondità (ramped) per garantire la generazione di alberi di diverse dimensioni e forme. Non è necessario che la popolazione iniziale sia del tutto casuale. Se si conosce qualcosa sulle proprietà probabili della soluzione desiderata, si possono usare alberi che hanno queste proprietà per generare la popolazione iniziale.

1.3.3 Funzione di fitness

Nell'ambito della programmazione genetica, la funzione di fitness viene utilizzata per valutare la qualità di ogni individuo della popolazione e determinare quali individui hanno maggiori probabilità di sopravvivere e trasmettere i loro geni alle generazioni successive. La funzione di fitness può essere definita in base alle esigenze specifiche del problema da risolvere. In genere, viene valutata la capacità dell'individuo di produrre una soluzione che rispetti le specifiche richieste del problema e di farlo in modo efficace. Ad esempio, per un algoritmo di classificazione, la funzione di fitness potrebbe valutare la precisione dell'individuo nella classificazione dei dati di input, o come nel nostro caso la F1score.
L'ottimizzazione della funzione di fitness è uno dei principali obiettivi della programmazione genetica. Ciò significa che l'algoritmo cerca di modificare i geni degli individui della popolazione in modo da migliorare la funzione di fitness e, di conseguenza, la qualità delle soluzioni. In questo modo, l'algoritmo genetico può convergere verso una soluzione ottimale per il problema in questione.

1.3.4 Selezione

Gli operatori genetici nella programmazione genetica vengono applicati a individui che sono selezionati probabilisticamente in base alla loro fitness. Il metodo più comunemente utilizzato per selezionare gli individui in GP è la **selezione a torneo**, in cui un certo numero di individui, scelti a caso da una popolazione, vengono confrontati tra loro e il migliore viene scelto come genitore.
Un altro metodo è la **selezione proporzionale alla fitness**, ma si può utilizzare qualsiasi meccanismo di selezione standard degli algoritmi evolutivi.

1.3.5 Crossover e mutazione

La programmazione genetica si discosta in modo significativo dagli altri algoritmi evolutivi nell'implementazione degli operatori di crossover e mutazione. Il crossoover più utilizzato è il **subtree crossover** (figura 1.4), dove viene scelto un punto di crossover casuale in entrambi i genitori e vengono creati i figli sostituendo il sottoalbero con radice nel punto di crossover nella copia del primo genitore con una copia del sottoalbero con radice nel punto di crossover del secondo genitore. Le copie vengono utilizzate per evitare di interrompere gli individui originali.

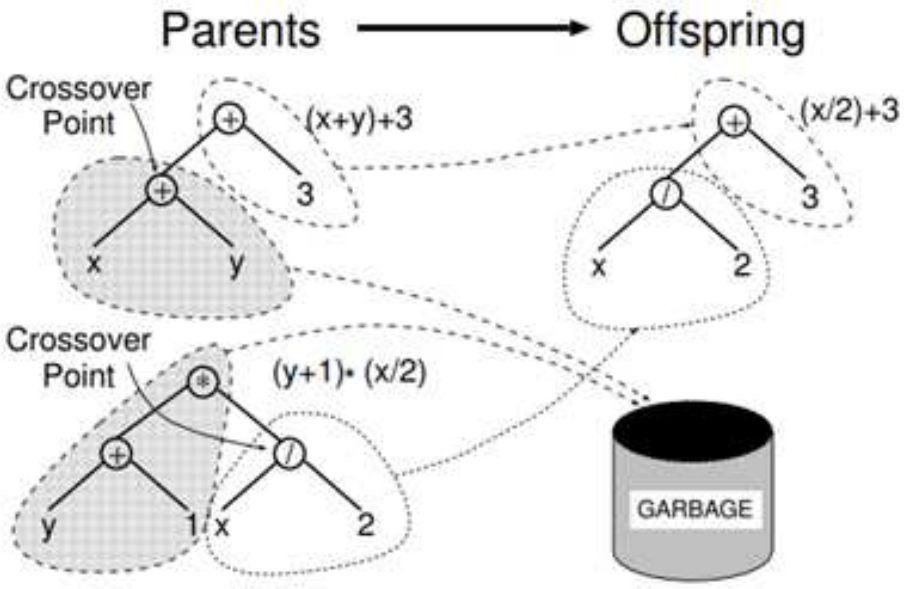

Figura 1.4: Esempio di subtree crossover

La forma di mutazione più comunemente utilizzata in GP è chiamata **subtree mutation** (figura 1.5) e seleziona casualmente un punto di mutazione in un albero e sostituisce il sottoalbero con radici in quel punto con un sottoalbero generato casualmente. La mutazione del sottoalbero viene talvolta implementata come crossover tra un programma e un programma casuale appena generato; questa operazione è nota anche come "headless chicken".

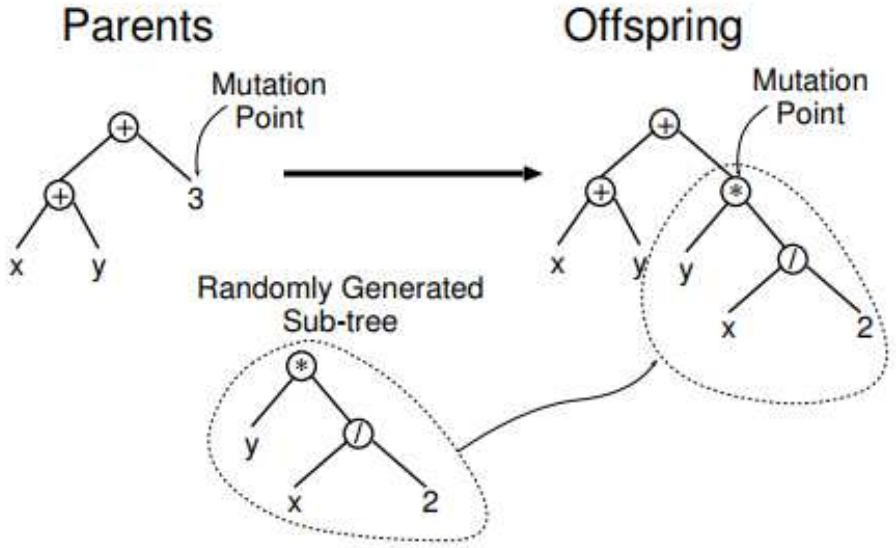

Figura 1.5: Esempio di subtree mutation

Un altro tipo di mutazione è la **mutazione puntiforme**, dove viene selezionato un nodo casuale e la primitiva memorizzata in quel nodo viene sostituita con un'altra primitiva casuale della stessa arità presa dall'insieme delle primitive. Se non esistono altre primitive con quell'arità, non succede nulla a quel nodo.

Capitolo 2

Strumenti utilizzati

In questa tesi, sono state utilizzate diverse librerie per implementare l'algoritmo. Tra queste, le più importanti sono DEAP [5] per l'algoritmo genetico, Tkinter [6] per l'interfaccia grafica, Scikit-learn [7] per l'addestramento e la valutazione del modello, Matplotlib per la visualizzazione dei grafici e la libreria Pickle [8] per salvare l'individuo migliore al termine dell'algoritmo, così da poterlo richiamare in altri script.

2.1 DEAP (Distributed Evolutionary Algorithms in Python)

DEAP è un framework Python che consente di implementare algoritmi evolutivi per la risoluzione di problemi di ottimizzazione. L'algoritmo utilizza la valutazione della fitness degli individui per selezionare i migliori candidati. La flessibilità di DEAP deriva dalla possibilità di creare i propri tipi di dati utilizzando il modulo creator, che consente di adattare l'algoritmo alle esigenze specifiche del problema da risolvere. Il metodo creator prevede la funzione "create" che accetta due argomenti: il nome della classe e la classe base. Tutti gli attributi successivi sono quelli della classe.
Per implementare l'algoritmo genetico (come riportato in figura 2.1), è necessario creare una classe FitnessMax con l'attributo weights impostato su

1 per indicare che si tratta di un problema di massimizzazione o -1 per un problema di minimizzazione. Inoltre, viene creato anche la classe Individual, basata su "PrimitiveTree", una classe fornita da DEAP per la rappresentazione delle espressioni matematiche. L'individuo ha un attributo "fitness" impostato sulla classe "FitnessMax" appena creata. In questo modo, gli individui possono essere valutati e selezionati in base alla loro fitness.

```
from deap import base, creator
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)
```

Figura 2.1: Definizione dei tipi di dati utilizzati

2.1.1 Inizializzazione

Una volta definiti i tipi di dati necessari, è fondamentale inizializzarli con valori adeguati per poter avviare l'algoritmo evolutivo. In DEAP, questo processo viene semplificato grazie alla presenza di un contenitore chiamato "Toolbox", che include diversi strumenti tra cui gli inizializzatori.

```
import random
from deap import tools

IND_SIZE = 10

toolbox = base.Toolbox()
toolbox.register("attribute", random.random)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attribute, n=IND_SIZE)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Figura 2.2: Esempio di inizializzazione

Il codice in Figura 2.2 inizializza una popolazione di individui, in questo caso contenenti numeri casuali.
Il primo strumento che viene registrato nella Toolbox è "attribute", che definisce ogni elemento del cromosoma. In questo caso è la funzione random.
Il secondo strumento che viene registrato è "individual", che utilizza la funzione tools.initRepeat per creare un individuo come una lista di attributi di

dimensione "IND_SIZE". Questo individuo viene creato utilizzando la classe Individual precedentemente definita tramite il modulo creator.
Infine, la Toolbox registra "population", che utilizza la funzione tools.initRepeat per creare una popolazione di individui. Sarà quindi possibile la creazione di una popolazione iniziale tramite la chiamata alla funzione toolbox.population(). Questo codice può essere facilmente personalizzato per inizializzare individui con tipi diversi di dati o con strutture più complesse.

2.1.2 Set di primitive

In DEAP, le primitive e i terminali definiti dall'utente sono organizzati in un set di primitive.

```
pset = PrimitiveSet("main", 2)
pset.addPrimitive(max, 2)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.mul, 2)
pset.addTerminal(3)
pset.addEphemeralConstant(lambda: random.uniform(-1, 1))
```

Figura 2.3: Esempio set di primitive

Come si vede nell'esempio riportato in figura 2.3, la prima riga crea un set di primitive specificando il nome del set ("main") e il numero di argomenti della funzione principale cui si farà riferimento con gli alias ARG0,ARG1,...,ecc (parametro 2). Le tre righe successive aggiungono funzioni come primitive: il primo argomento è la funzione da aggiungere e il secondo argomento la sua arità (numero di input). I terminali costanti si possono aggiungere al set usando "addTerminal". Inoltre, possono essere aggiunte anche costanti effimere, ovvero terminali che incapsulano valori generati da una data funzione in fase di esecuzione permettendo di avere terminali di valori diversi.
Considerato l'approccio modulare utilizzato nell'algoritmo genetico di questo progetto di tesi, ad ogni iterazione verranno esaminati gli individui della popolazione ottenuta ed estratti i sottomoduli di profondità 1 e 2. Di questi sottomoduli ne verranno scelti N (impostato dall'utente tramite interfaccia

grafica) in base alla loro frequenza e la loro fitness e questi N verranno inseriti nel set di primitive e utilizzati come funzioni nelle iterazioni successive. In questo modo, il set di primitive verrà costantemente aggiornato e migliorato, aumentando la possibilità di trovare soluzioni ottimali al problema considerato in forma compatta.

2.1.3 Operatori

Il modulo tools di DEAP implementa molti degli operatori tipici degli algoritmi genetici e questi vengono registrati nella toolbox utilizzando un alias, per rendere gli algoritmi genetici indipendenti dal nome specifico dell'operatore.
Nel codice di esempio in figura 2.4, vengono registrati nella toolbox gli operatori di selezione (a torneo), crossover (a due punti) e mutazione (gaussiana) utilizzando gli alias. Inoltre, bisogna definire la funzione di valutazione del nostro algoritmo: è la parte più personale dell'algoritmo e restituisce un valore di fitness sotto forma di tupla in modo da essere iterabile. Nel codice riportato la funzione di valutazione "evaluate" restituisce la somma degli elementi dell'individuo.
Tutti questi operatori registrati nella toolbox possono essere utilizzati nei passi successivi dell'algoritmo genetico, in modo da creare, valutare e modificare la popolazione di individui.

```
def evaluate(individual):
    return sum(individual),

toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evaluate)
```

Figura 2.4: Esempio operatori e funzioni di valutazione

2.2 Tkinter

Tkinter è una libreria di interfacce grafiche (GUI) per Python, che fornisce un'interfaccia per il toolkit GUI Tk. Tkinter è incluso nella distribuzione standard di Python e quindi non richiede alcuna installazione o configurazione aggiuntiva.
Il cuore di una interfaccia in Tkinter è un widget, ovvero un oggetto grafico che viene creato e manipolato tramite codice Python. Ci sono molti tipi di widget disponibili in Tkinter, tra cui finestre, pulsanti, caselle di testo, menù a tendina e molto altro. Per creare una finestra, si utilizza il metodo Tk() della classe Tkinter, mentre per aggiungere altri widget si utilizzano altri metodi come ad esempio Button() o Entry().
Inoltre, Tkinter include anche un sistema di gestione degli eventi, che permette di associare azioni a eventi specifici come il clic su un pulsante o la pressione di un tasto sulla tastiera.

2.2.1 Realizzazione dell'interfaccia grafica

In questo lavoro di tesi è stata progettata un'interfaccia grafica utilizzando il framework Tkinter a supporto dell'algoritmo genetico. Tramite questa interfaccia grafica l'utente può impostare i parametri dell'algoritmo genetico, scegliere il dataset su cui eseguire l'algoritmo e specificare il numero di volte che l'algoritmo deve essere eseguito.
Grazie all'uso di widget specifici, l'utente può facilmente inserire i parametri necessari e avviare l'esecuzione dell'algoritmo genetico premendo il pulsante "Esegui".
Durante l'esecuzione dell'algoritmo l'utente viene informato se questo è ancora in esecuzione oppure terminato.
Al termine dell'esecuzione, i risultati dell'algoritmo vengono salvati automaticamente in un file di testo per permettere una successiva analisi; inoltre, nell'interfaccia viene visualizzato un grafico che mostra l'andamento dell'algoritmo sui set di validazione e di test, generati a partire dal dataset inserito.

Questa visualizzazione dei risultati permette all'utente di valutare l'efficacia dell'algoritmo genetico o, nel caso di un problema di classificazione, l'eventuale insorgenza di overfitting.

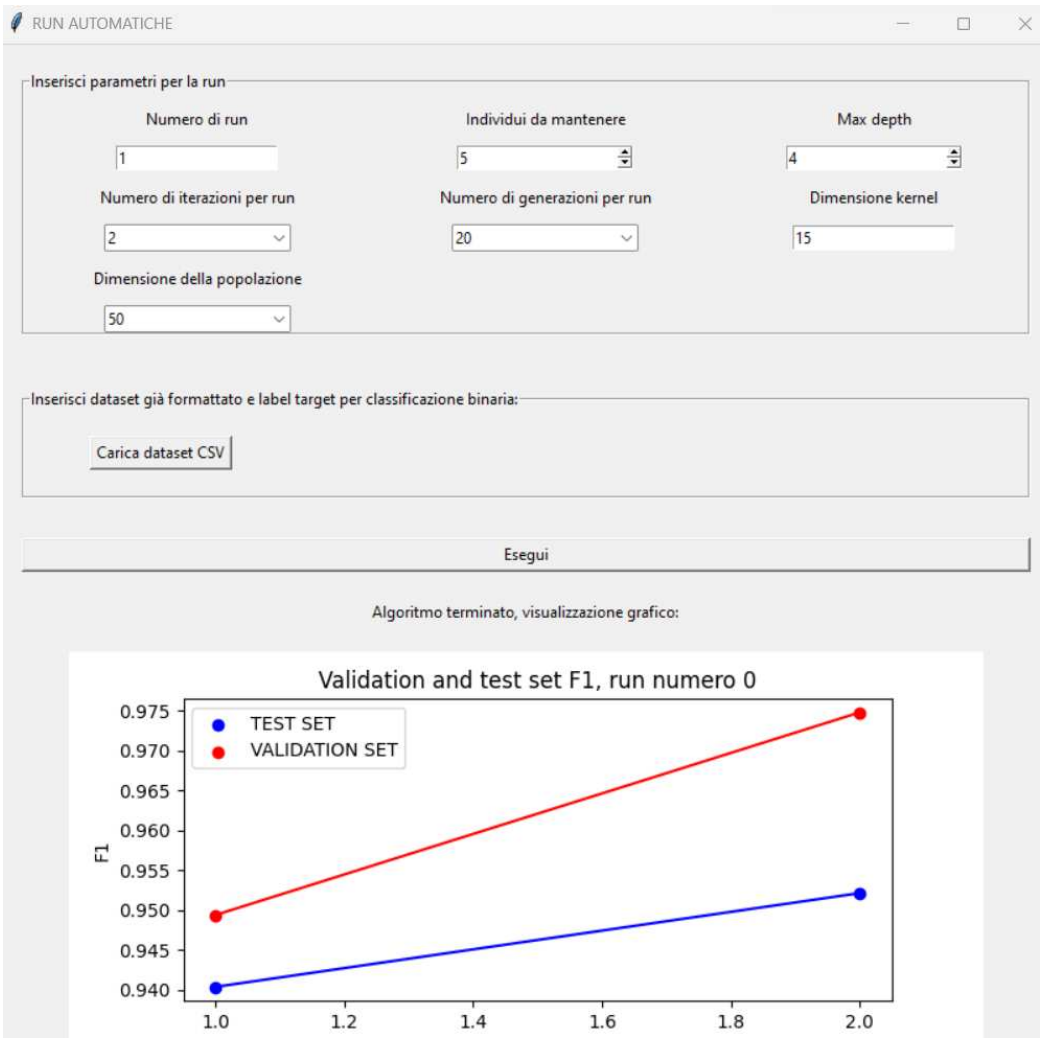

Figura 2.5: Interfaccia grafica con alcuni parametri di esempio e la visualizzazione finale dei risultati

2.3 Scikit-learn

Scikit-learn è una libreria di apprendimento automatico open source che supporta l'apprendimento supervisionato e non supervisionato. Fornisce inoltre vari strumenti per l'adattamento del modello, la preelaborazione dei dati, la selezione del modello, la valutazione del modello e molte altre utilità.
Nel mio progetto di tesi, ho utilizzato Scikit-learn per implementare un modello di classificazione basato sull'algoritmo Random Forest.
Random Forest è un algoritmo di apprendimento automatico di tipo ensemble che combina diversi alberi decisionali in una "foresta". Ogni albero decisionale viene addestrato su un sottoinsieme dei dati di training e vota per la classe di appartenenza dell'istanza da classificare. La classe che riceve il maggior numero di voti da parte degli alberi della foresta viene scelta come classe di appartenenza dell'istanza.
Ho realizzato il modello tramite la funzione di Scikit-learn "RandomForestClassifier" settando alcuni parametri come il numero di alberi della foresta e un seed per eliminare la componente stocastica del Random Forest. Questo modello è stato addestrato con la funzione "fit" usando il training set, mentre la predizione è stata fatta su validation set e test set con la funzione "predict".
In particolare, per ottenere training set, validation set e test set, ho suddiviso il dataset fornito dall'utente utilizzando la funzione di Scikit-learn "train_test_split". Per quanto riguarda la valutazione delle prestazioni del modello di classificazione, ho scelto di utilizzare come metrica di valutazione l'F1-score.
L'F1-score può essere interpretato come una media armonica della precision e della recall: la precision indica la percentuale di istanze positive che sono state correttamente classificate dal modello, mentre la recall indica la percentuale di istanze positive che sono state individuate dal modello. L'F1-score tiene conto di entrambe queste metriche e produce un valore che va da 0 a 1, essendo il suo valore migliore, che implica che sia precisione che recall siano uguali a 1.

La formula per il punteggio F1 è:

$$F1 = \frac{2 * (precision * recall)}{precision + recall} \tag{2.1}$$

Per ulteriori informazioni sulla qualità del mio modello, ho utilizzato anche la matrice di confusione. La matrice di confusione è una tabella che mostra il numero di previsioni corrette ed errate fatte dal modello per ciascuna classe. Se il problema è una classificazione binaria, possiamo suddividerla in 4 classi: veri positivi TP, falsi positivi FP, falsi negativi FN e veri negativi TN.
In base a questi parametri si può dare una definizione di precision e recall:

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

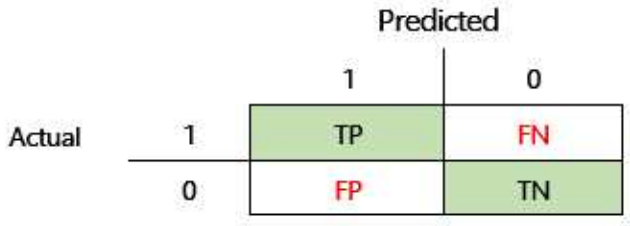

Figura 2.6: Matrice di confusione

2.4 Matplotlib

Matplotlib è una libreria completa per la creazione di visualizzazioni di dati statiche, animate e interattive in Python. Matplotlib è altamente personalizzabile e permette di controllare quasi ogni aspetto dei grafici, tra cui la dimensione, i colori, lo stile delle linee, i tipi di carattere, l'etichettatura degli assi e molto altro.

In questo progetto di tesi, ho utilizzato Matplotlib per realizzare molti grafici per visualizzare alcuni concetti principali. Ad esempio, tramite istogramma, si visualizza la frequenza con cui si presentano i moduli (figura 2.7 e 2.8), fondamentali per quest'approccio modulare della programmazione genetica. Inoltre, sono stati usati grafici a dispersione per rappresentare la fitness associata ai moduli più frequenti (figura 2.9).

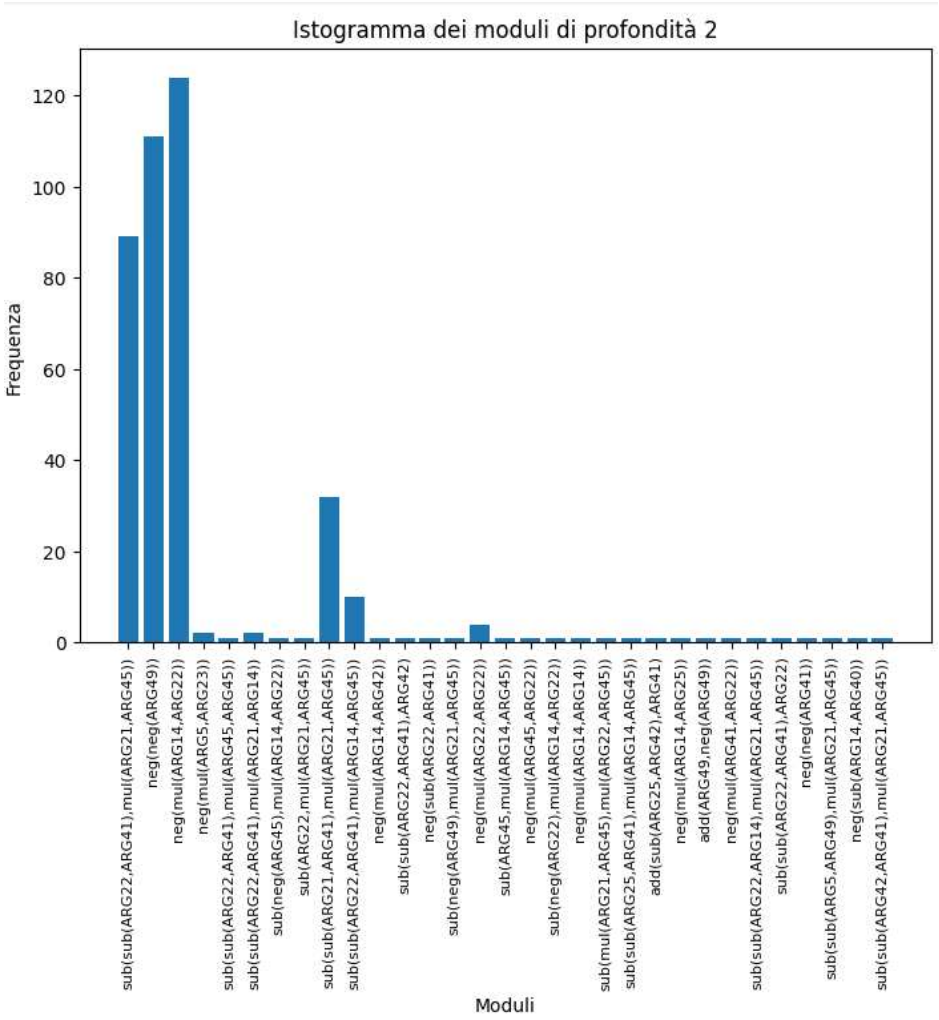

Figura 2.7: Istogramma delle frequenze di apparizione dei moduli di profondità 2 negli individui della popolazione

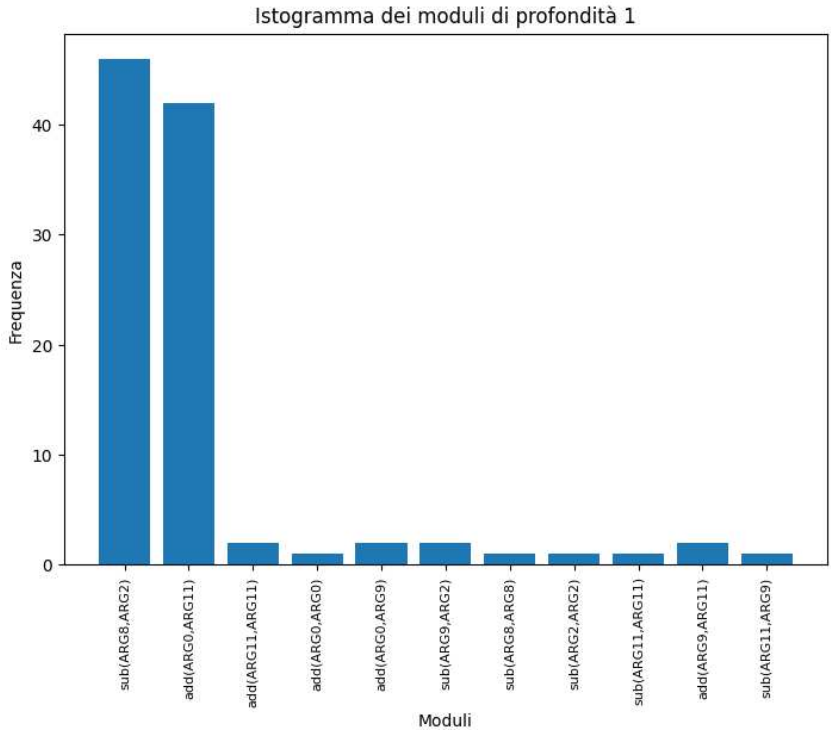

Figura 2.8: Istogramma delle frequenze di apparizione dei moduli di profondità 1 negli individui della popolazione

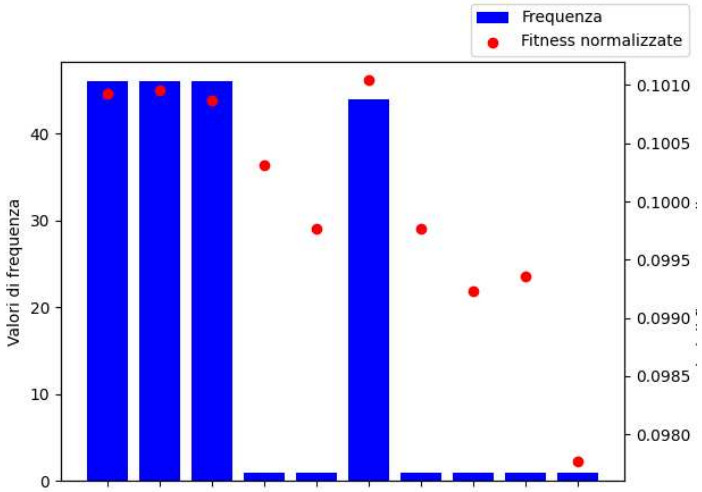

Figura 2.9: Istogramma delle frequenze dei moduli più frequenti, con i relativi valori di fitness

2.5 Pickle

Pickle è un modulo Python che consente di serializzare oggetti Python in un formato binario per consentirne la memorizzazione e il trasferimento tra diversi processi o computer. La libreria Pickle è particolarmente utile per salvare e caricare grandi quantità di dati strutturati, come ad esempio i modelli di machine learning o i dataset. Per serializzare una gerarchia di oggetti, è sufficiente chiamare la funzione dumps(). Analogamente, per deserializzare un flusso di dati, si chiama la funzione loads().
Ho utilizzato questa libreria per salvare il migliore individuo della popolazione finale dell'algoritmo genetico, in modo da poterlo richiamare in uno script esterno: una volta che ho una soluzione al mio problema devo fare in modo che sia utilizzabile in un qualunque programma che richieda di risolvere quel problema.
In questo script esterno, il migliore individuo sarà letto, compilato ed eseguito su un set di dati, stampando i risultati ottenuti. Tuttavia, avendo usato un approccio modulare, per poter compilare correttamente l'individuo, sarà necessario disporre anche del set di primitive relativo.
Di conseguenza, oltre a salvare il migliore individuo, sarà necessario salvare anche il pset, ovvero l'insieme completo di primitive disponibili per l'individuo migliore. In questo modo, si potrà garantire che l'individuo possa essere compilato correttamente e che i risultati siano affidabili. Per salvare il Primitive Set non è stato usato Pickle perché non supporta la serializzazione di funzioni e metodi, bensì Dill, una libreria molto simile a Pickle ma con funzionalità aggiuntive.

Capitolo 3

Dataset

Il capitolo precedente ha illustrato come la creazione dell'interfaccia grafica consenta all'utente di impostare i parametri più importanti dell'algoritmo genetico e di inserire il dataset su cui desidera lavorare. Il dataset deve essere già preformattato con la label nella prima colonna e i dati nelle colonne restanti. Una volta inserito il dataset, verrà suddiviso in training set, validation set e test set, che verranno utilizzati per l'addestramento e la valutazione del modello creato. Nella tesi sono stati utilizzati due dataset: il primo è composto da segnali forniti da Physikalisch-Technische Bundesanstalt (PTB) e usato per la classificazione dei segnali normali o anormali dell'elettrocardiogramma (ECG), mentre il secondo è il Dataset Digit, utilizzato per il riconoscimento di cifre scritte a mano.

3.1 PTB Diagnostic ECG Database

L'elettrocardiogramma o ECG è un esame diagnostico volto a registrare l'attività elettrica del cuore, al fine di valutarne lo stato di salute ed individuare diverse anomalie cardiache, patologie oppure aritmie. La registrazione dell'attività elettrica del cuore avviene attraverso elettrodi posti sulla pelle del paziente che rilevano le variazioni di tensione generate dalle contrazioni del muscolo cardiaco e producono un segnale variabile nel tempo. L'analisi di

questo segnale permette di rilevare varie anomalie cardiache o carenze nel flusso sanguigno attraverso il cuore. In questo modo, l'ECG rappresenta uno dei test diagnostici più comuni utilizzati per rilevare problemi cardiaci e monitorare la salute del cuore.
In questo progetto si è lavorato su segnali forniti da Physionet's ECG Database [9] fornito da Physikalisch-Technische Bundesanstalt (PTB) [10].
Il PTB Diagnostic ECG è una raccolta di 549 ECG di 294 pazienti, alcuni dei quali sani e altri affetti da diverse patologie cardiache. Ogni soggetto è rappresentato da uno a cinque record (ECG) e ogni registrazione comprende 15 segnali ad alta risoluzione misurati simultaneamente. Per la maggior parte di questi record ECG è presente un riepilogo clinico dettagliato, che include età, sesso, diagnosi e, ove applicabile, dati su anamnesi, farmaci e interventi, patologia coronarica, ventricolografia, ecocardiografia ed emodinamica.
La versione pre-elaborata di questo dataset è disponibile su Kaggle [11]. Per la mia tesi, ho utilizzato i file "ptbdb_abnormal.csv" e "ptbdb_normal.csv" in cui ad ogni segnale viene rispettivamente assegnata l'etichetta 1 (battito anormale) e 0 (battito normale). I due file CSV sono stati uniti in un unico file CSV mantenendo 3000 segnali per i battiti normali e 3000 segnali per i battiti anormali, ottenendo così un dataset bilanciato.

3.1.1 Pre-elaborazione dei segnali ECG

In questo progetto di tesi, i segnali del dataset sono stati pre-elaborati da Mohammad Kachuee, Shayan Fazeli e Majid Sarrafzadeh [12]. Gli autori hanno impiegato un metodo efficace per la pre-elaborazione dei segnali e l'estrazione dei battiti cardiaci da essi, ottenendo come risultato i file csv "ptbdb_abnormal.csv" e "ptbdb_normal.csv".
Il processo di estrazione dei battiti da un segnale ECG è stato svolto seguendo i seguenti passaggi: inizialmente, il segnale ECG è stato suddiviso in finestre di 10 secondi e ciascuna di esse è stata selezionata per l'analisi. L'analisi consiste nel normalizzare i valori di ampiezza del segnale in un intervallo tra zero e uno per facilitare la comparazione tra i segnali. Dopo di che, sono

stati identificati i massimi locali nel segnale ECG e applicando una soglia di 0.9 al valore normalizzato dei massimi locali, sono stati individuati i picchi R (il picco più alto del segnale ECG). Successivamente, è stata calcolata la mediana degli intervalli di tempo R-R, che rappresenta il periodo nominale del battito cardiaco per quella specifica finestra di 10 secondi. Questo valore rappresenta la durata media tra due battiti consecutivi.

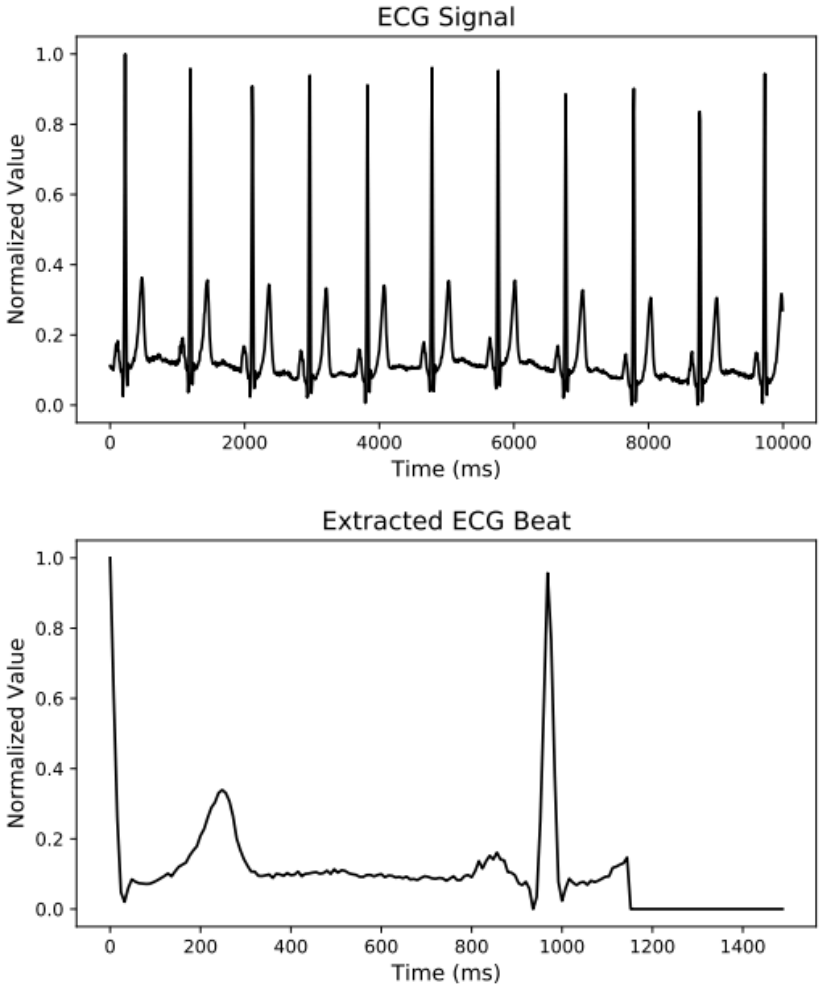

Figura 3.1: Un campione di 10 secondi del segnale ECG e un battito estratto da esso. Output pre-elaborato dal documento "ECG Heartbeat Classification: A Deep Transferable Representation".

Per ogni picco R individuato, viene selezionata una porzione del segnale che

corrisponde a 1.2 volte il valore del periodo nominale del battito cardiaco. Questa porzione del segnale contiene le informazioni specifiche relative a un singolo battito cardiaco. Infine, per mantenere una lunghezza fissa predefinita per tutte le porzioni selezionate, queste vengono riempite con zeri. Ciò aiuta a standardizzare le dimensioni delle porzioni e semplifica le successive elaborazioni e analisi dei battiti cardiaci.
Gli autori così facendo hanno fornito due file di dataset PTB ECG: uno contenente i battiti cardiaci anormali con la variabile di classe 1 e l'altro contiene i battiti cardiaci normali con la variabile di classe 0. Entrambi i file hanno 188 colonne, ma le letture anomale sono 10506 mentre quelle normali 4046. Per creare un unico file CSV, ho unito i dati mantenendo 3000 letture per ogni classe e inserendo la variabile di classe nella prima colonna. Il resto delle colonne rappresenta la lunghezza del segnale riempita con zeri per ottenere una lunghezza fissa.

3.2 Dataset Digit

Il dataset "Digit" contiene immagini di cifre da 0 a 9, scritte a mano da diversi autori. Ogni immagine è rappresentata da una matrice di pixel, in cui ogni pixel rappresenta l'intensità del colore o della scala di grigi. Queste immagini possono essere utilizzate per addestrare e testare algoritmi di riconoscimento delle cifre.
Il dataset "Digit" è parte della libreria Scikit-learn ed è composto da 1797 campioni, ognuno rappresentante un'immagine di una cifra ed etichettato con il corrispondente valore della cifra che rappresenta. Ciò rende il dataset adatto per compiti di classificazione, in cui l'obiettivo è predire correttamente la cifra rappresentata in una determinata immagine. In questo progetto di tesi, il dataset è stato importato ed elaborato in modo tale da ottenere un file CSV da utilizzare come dataset nell'algoritmo genetico.

3.2.1 Pre-elaborazione e caratteristiche dei dati

Dopo aver importato il dataset da Scikit-learn tramite la funzione "datasets.load_digits()", esso viene tradotto in un formato CSV in cui ogni riga rappresenta un'immagine che deve essere trasformata in un vettore di feature di lunghezza 64.
Per creare il file CSV è stata seguita una procedura semplice: la colonna iniziale del file CSV è stata dedicata al target delle varie immagini, rappresentando le cifre da 0 a 9, mentre le restanti colonne corrispondono ai pixel delle immagini, rappresentati come vettori di livelli di grigio.
Questa pre-elaborazione e rappresentazione dei dati sotto forma di file CSV consente di utilizzare il dataset "Digit" all'interno dell'algoritmo genetico realizzato.

3.3 Creazione di training set, validation set e test set

La preparazione dei dati che verranno utilizzati nell'addestramento di un modello di machine learning è fondamentale. In questo progetto di tesi, attraverso l'interfaccia grafica, l'utente può selezionare quale dataset utilizzare per addestrare il modello. Il dataset che seleziona deve essere un file CSV con un formato speciale: la prima riga deve contenere l'intestazione e la prima colonna deve contenere le etichette dei segnali.
Una volta inserito il dataset, questo viene importato nel nostro algoritmo leggendo le intestazioni delle colonne, le etichette (labels) dei segnali e i segnali stessi. Successivamente le etichette e i segnali vengono mescolati in modo sincronizzato, garantendo che l'ordine dei dati rimanga coerente tra le etichette e i corrispondenti segnali, evitando disallineamenti e assicurando una corretta associazione tra etichette e dati. Infine, il dataset viene suddiviso in tre parti: il set di addestramento (training set), il set di test (test set) e il set di validazione (validation set). Questa suddivisione è fondamentale per

valutare le prestazioni del modello.
Per ottenere una suddivisione equilibrata e rappresentativa del dataset, viene utilizzata la funzione train_test_split dalla libreria Scikit-learn. In particolare, il dataset viene suddiviso inizialmente in due parti: il 20% dei dati viene assegnato al test set, mentre il restante 80% viene utilizzato come set di addestramento. Successivamente, il set di addestramento viene ulteriormente suddiviso per ottenere il set di validazione. Questa suddivisione consente di monitorare l'andamento delle prestazioni del modello durante il processo di addestramento, fornendo un'indicazione sulla sua capacità di generalizzazione.
Una volta ottenuti i set di addestramento, test e validazione, è possibile utilizzarli per l'addestramento del modello e la valutazione delle sue prestazioni.

Capitolo 4

Implementazione dell'algoritmo

Una volta che l'utente, tramite interfaccia grafica, ha inserito i parametri necessari all'algoritmo di classificazione e il dataset su cui vuole lavorare, viene avviata l'esecuzione dell'algoritmo. In questo capitolo, verrà descritto l'algoritmo in dettaglio, illustrando l'inizializzazione del set di primitive e dei terminali, l'utilizzo del toolbox, l'algoritmo Easimple e la funzione di valutazione. Inoltre, verrà spiegato come viene effettivamente implementato l'approccio modulare.

4.1 Inizializzazione del set di primitive e di terminali

Per creare un algoritmo di programmazione genetica, il primo passo cruciale consiste nell'inizializzare il set di primitive e il set di terminali, come evidenziato nella figura 4.1.
Ogni primitiva nel set è caratterizzata da una funzione associata, che viene richiamata quando il nodo corrispondente viene valutato, e ha una specifica cardinalità, cioè il numero di parametri che accetta come input. Inizialmente il set di primitive è composto da funzioni elementari come addizione (operator.add), sottrazione (operator.sub), negazione (operator.neg), moltiplicazione (mul) e divisione protetta (protectedDiv). Sia l'operazione di mol-

tiplicazione che di divisione, prima di restituire un risultato, effettuano un controllo per gestire eventuali eccezioni e valori non validi. Queste funzioni rappresentano le operazioni matematiche di base che possono essere combinate per creare espressioni più complesse durante il processo di evoluzione. Inoltre, la funzione "addEphemeralConstant" viene utilizzata per aggiungere costanti effimere al set di primitive. Queste costanti vengono create utilizzando una funzione lambda che genera un valore casuale compreso tra -1 e 1.
Infine, le funzioni generate dal set di primitive richiederanno un numero di argomenti pari a KERNEL_SIZE per essere valutate correttamente.

```
pset = gp.PrimitiveSet("MAIN", KERNEL_SIZE)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(mul, 2)
pset.addPrimitive(protectedDiv, 2)
pset.addPrimitive(operator.neg, 1)
pset.addEphemeralConstant(f"rand101_{const}", lambda: random.randint(-1, 1))
```

Figura 4.1: Inizializzazione del set di primitive

4.2 Registrazione degli operatori evolutivi nel Toolbox

Il "Toolbox" fornisce un insieme di metodi per registrare e definire le operazioni fondamentali dell'algoritmo genetico.
Inizialmente, viene utilizzata la funzione "genHalfAndHalf" di DEAP per generare le espressioni degli individui nella popolazione iniziale. Questa funzione crea un'espressione utilizzando l'insieme di primitive (PrimitiveSet) che è stato creato precedentemente. La popolazione viene generata dividendola in due parti: una parte viene generata utilizzando il metodo "Full" (dove ogni foglia ha la stessa profondità), mentre l'altra parte viene generata utilizzando il metodo "Grow" (dove le foglie possono avere profondità diverse).

Inoltre, vengono specificate le profondità minima e massima per gli alberi delle espressioni. Questo significa che durante la generazione delle espressioni per gli individui, la profondità degli alberi sarà compresa tra la profondità minima e la profondità massima specificate.
Successivamente, vengono registrati i metodi per l'inizializzazione degli individui e delle popolazioni utilizzando le funzioni "initIterate" e "initRepeat" fornite da DEAP. Il metodo compile viene registrato utilizzando la funzione "gp.compile" per compilare le espressioni degli individui in funzioni eseguibili. Nel Toolbox viene registrata la funzione di valutazione ("evaluate") che rappresenta la metrica di fitness utilizzata per valutare la qualità degli individui. In questo caso, viene utilizzata la funzione "evalTrainingSet" che verrà spiegata in seguito. Successivamente, vengono registrati i metodi per la selezione, il crossover e la mutazione: viene utilizzata la selezione a torneo (selTournament), l'accoppiamento a un punto (cxOnePoint) e la mutazione uniforme (mutUniform) con la generazione di espressioni mutate utilizzando il metodo genFull.
Infine, la funzione "gp.staticLimit" viene utilizzata per decorare le operazioni di accoppiamento e mutazione nel Toolbox. La decorazione specifica che l'altezza dell'albero delle espressioni degli individui generati dalle operazioni di accoppiamento e mutazione non può superare il valore massimo MAX_DEPTH. In figura 4.2 viene riportato il codice corrispondente a quanto descritto.

```
toolbox = base.Toolbox()
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=MIN_DEPTH, max_=MAX_DEPTH)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("compile", gp.compile, pset=pset)

toolbox.register("evaluate", evalTrainingSet)
toolbox.register("select", tools.selTournament, tournsize=5)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=MIN_DEPTH, max_=MAX_DEPTH)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)

toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"), max_value=MAX_DEPTH))
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"), max_value=MAX_DEPTH))
```

Figura 4.2: Registrazione degli operatori nel Toolbox

4.3 Funzione di valutazione

La funzione di valutazione "evalTrainingSet" (figura 4.3) svolge un ruolo cruciale all'interno dell'algoritmo evolutivo, poiché è responsabile di valutare e assegnare un punteggio, chiamato fitness, a ciascun individuo della popolazione.
Il processo di valutazione inizia prendendo come input l'individuo da valutare, che rappresenta una possibile soluzione al problema affrontato dall'algoritmo evolutivo. In questo caso specifico, l'obiettivo è trovare una trasformazione del segnale che permetta di ottimizzare le prestazioni di un classificatore Random Forest applicato ai dati trasformati. Questo individuo viene utilizzato per generare un nuovo training set, che addestra il modello di classificazione Random Forest, e un nuovo validation set, che valuta il modello addestrato, partendo dal training set e validation set originali.

```
def evalTrainingSet(individual):
    #compilazione dell'individuo
    clf = gp.PrimitiveTree(individual)
    func = gp.compile(clf, pset)

    #generazione dei nuovi training set e validation set
    new_train_set=convolution(func,train_data,KERNEL_SIZE)
    new_val_set=convolution(func,data_val,KERNEL_SIZE)

    #addestramento e valutazione del modello
    f1_validation,l,p,rf=training_RF(new_train_set,train_labels, new_val_set, labels_val)

    # Calcolo della fitness con penalizzazione proporzionale al numero di nodi dell'albero
    num_nodes=len(individual)
    K = 10e-5
    fitness = f1_validation / (1 + K * num_nodes)

    return (fitness,)
```

Figura 4.3: Funzione di valutazione

Per poter utilizzare l'individuo, viene generato un programma eseguibile attraverso la compilazione dell'albero sintattico associato all'individuo. Questo programma è essenziale per passare da una rappresentazione ad albero a una forma di funzione valutabile. Il programma genetico dato dall'individuo compilato verrà utilizzato per effettuare la convoluzione con il training set e il va-

lidation set originali, usando KERNEL_SIZE come dimensione della finestra di convoluzione. Se consideriamo il training set, nella funzione "convolution" questo viene fatto scorrere e per ogni sua riga vengono considerate tutte le finestre di dimensione pari a KERNEL_SIZE. Il numero di finestre possibili in una riga del training set sarà pari a $len(riga) - KERNELSIZE + 1$. Per ogni finestra, i KERNEL_SIZE valori della riga considerati saranno i parametri della funzione generata compilando l'individuo, che accetta KERNEL_SIZE parametri per essere valutata correttamente. Il segnale che restituisce questa funzione farà parte del nuovo training set che si ottiene ripetendo l'operazione per tutte le finestre. Il ragionamento è analogo per ottenere il nuovo validation set.
Ottenuti il nuovo training set e il nuovo validation set, insieme alle rispettive etichette, vengono passati come parametri alla funzione "training_RF", che effettua l'addestramento e la valutazione del classificatore. (figura 4.4).

```python
#addestramento/valutazione random forest
def training_RF(X_data, X_labels, Y_data, Y_labels):
    try:
    # Convertire le etichette di classe in un formato adatto per la classificazione multi-classe
        le = LabelEncoder()
        le.fit(X_labels)
        X_labels_multi = le.transform(X_labels)
        Y_labels_multi = le.transform(Y_labels)

        seed = 40
        # salvare lo stato attuale del generatore di numeri casuali di numpy
        rng_state = np.random.get_state()
        # Creare il modello Random Forest per la classificazione multi-classe con lo stesso seed
        rf = RandomForestClassifier(n_estimators=50, random_state=seed)
        # ripristinare lo stato del generatore di numeri casuali di numpy
        np.random.set_state(rng_state)

        # Addestrare il modello
        rf.fit(X_data, X_labels_multi)
        # Valutare il modello usando il validation set per generalizzare meglio
        y_predictions = rf.predict(Y_data)

        # Calcolare la media degli F1-score per ogni classe
        f1_per_class = []
        for i in range(len(le.classes_)):
            y_true_i = [int(label == i) for label in Y_labels_multi]
            y_pred_i = [int(pred == i) for pred in y_predictions]
            f1_i = f1_score(y_true_i, y_pred_i, zero_division=0)
            f1_per_class.append(f1_i)
        mean_f1 = sum(f1_per_class) / len(f1_per_class)

    except Exception as e:
        # In caso di eccezione, impostiamo la fitness a zero
        mean_f1 = 0
        y_predictions = None
        rf = None
        Y_labels_multi = None

    return mean_f1 , Y_labels_multi, y_predictions, rf
```

Figura 4.4: Classificazione tramite Random Forest: addestramento e valutazione del modello

Il modello di classificazione usato è un Random Forest e prima di addestrarlo, le etichette di classe vengono convertite in un formato adatto per la classificazione multi-classe utilizzando l'oggetto LabelEncoder. Ciò garantisce che le etichette siano rappresentate come numeri interi univoci, necessari per l'addestramento corretto del modello. Inoltre, viene impostato un seed per il generatore di numeri random che garantisce che il processo di addestramento sia riproducibile. Ciò significa che, a parità di seed, il modello addestrato restituirà sempre gli stessi risultati.
Dopo l'addestramento, il modello Random Forest viene utilizzato per fare previsioni sul validation set. La metrica di valutazione è F1_score, che viene calcolata separatamente per ogni classe confrontando le etichette vere con le previsioni effettuate dal modello. In seguito viene effettuata la media degli F1_score di tutte le classi, che rappresenta sia la misura complessiva delle prestazioni del modello sia la fitness dell'individuo.
Alla fine della funzione di valutazione, alla fitness ottenuta viene applicata una penalizzazione proporzionale al numero di nodi dell'albero che rappresenta l'individuo valutato. Questo viene fatto, a parità di fitness, per favorire gli alberi più piccoli e limitare quindi la dimensione media degli alberi della popolazione che, altrimenti, tenderebbe a crescere.

4.4 Ciclo di iterazioni dell'algoritmo genetico

Una volta inizializzati pset e toolbox, viene creata una popolazione iniziale (figura 4.5) con un numero di individui pari al parametro corrispondente inserito dall'utente tramite interfaccia grafica.

```
pop = toolbox.population(n=N_POPOLATION)
```

Figura 4.5: Creazione della popolazione

Tramite interfaccia l'utente indica anche il numero di iterazioni, ovvero quante volte vuole eseguire l'algoritmo genetico. All'inizio della prima iterazione viene aggiunta alla popolazione una funzione identità, che sarà utile al ter-

mine delle iterazioni per sapere se l'approccio ha migliorato i risultati oppure no: se la funzione identità sarà l'individuo migliore al termine di tutto l'algoritmo allora l'approccio usato non ha portato a miglioramenti. Dalla seconda iterazione in poi verrà preservato nella popolazione l'individuo migliore dell'iterazione precedente (elitismo). Gli individui migliori vengono salvati in una struttura chiamata Hall-Of-Fame. L'elitismo è realizzato sostituendo casualmente individui della popolazione con il migliore o i migliori della precedente generazione.

4.4.1 Algoritmo genetico Easimple

Arrivati a questo punto viene eseguito il processo evolutivo attraverso l'algoritmo Easimple.

Parameters:	• **population** – A list of individuals. • **toolbox** – A `Toolbox` that contains the evolution operators. • **cxpb** – The probability of mating two individuals. • **mutpb** – The probability of mutating an individual. • **ngen** – The number of generation. • **stats** – A `Statistics` object that is updated inplace, optional. • **halloffame** – A `HallOfFame` object that will contain the best individuals, optional. • **verbose** – Whether or not to log the statistics.
Returns:	The final population
Returns:	A class:*~deap.tools.Logbook* with the statistics of the evolution

Figura 4.6: Parametri e valori di ritorno della funzione Easimple

L'algoritmo accetta una popolazione e la evolve usando il metodo varAnd(). In questo metodo vengono selezionati alcuni individui dalla popolazione esistente come genitori per generare nuovi individui mediante crossover (mating pool). Il crossover combina le caratteristiche degli individui genitori per creare nuove soluzioni che possono essere migliori delle soluzioni originali. Successivamente, alcuni degli individui ottenuti dal crossover vengono sottoposti a mutazione, che introduce piccole modifiche casuali nelle soluzioni per esplorare lo spazio delle possibili soluzioni in modo più ampio. L'algoritmo Easimple restituisce la popolazione ottimizzata e un Logbook con le statistiche dell'evoluzione.

```
evaluate(population)
for g in range(ngen):
    population = select(population, len(population))
    offspring = varAnd(population, toolbox, cxpb, mutpb)
    evaluate(offspring)
    population = offspring
```

Figura 4.7: Pseudocodice dell'algoritmo Easimple

Come anche riportato nello pseudocodice che descrive Easimple in figura 4.7, l'algoritmo in primo luogo valuta gli individui che non sono ancora stati esaminati, dopo di che entra in un ciclo generazionale nel quale viene applicata la procedura di selezione per sostituire interamente la popolazione parentale. In seguito, viene applicato il metodo varAnd() per produrre la popolazione della generazione successiva, vengono valutati i nuovi individui e calcolate le statistiche della popolazione. Infine, dopo N generazioni (parametro scelto dall'utente) l'evoluzione si arresta e viene restituita una tupla contenente la popolazione evoluta e il Logbook del processo.
In questo progetto di tesi è stata effettuata una piccola modifica all'algoritmo standard Easimple in modo tale da implementare l'elitismo tra una generazione e l'altra: l'individuo migliore della i-esima generazione viene inserito nella popolazione della generazione i+1. Dopo aver effettuato crossover e mutazione, un individuo della popolazione viene sostituito casualmente con l'individuo migliore della generazione precedente, assicurandoci che la fitness dell'individuo che verrà sostituito sia inferiore di quella dell'individuo migliore.

4.4.2 Approccio modulare: scelta degli individui da mantenere

Poiché la caratteristica principale di questo progetto di tesi è l'approccio modulare, dopo che l'algoritmo Easimple ha generato una nuova popolazione composta da individui valutati, è necessario effettuare una scelta. Pertanto occorre analizzare gli individui valutati per estrarre i sottomoduli di profondità 1 e 2 che li compongono.

Ad esempio, se abbiamo un individuo "$add(mul(1, ARG1), neg(sub(-1, ARG0)))$" (figura 4.8), i sottomoduli di profondità 1 saranno "$mul(1, ARG1)$" e "$sub(-1, ARG0)$", mentre di profondità 2 avremo solo $neg(sub(-1, ARG0))$ (figura 4.9).

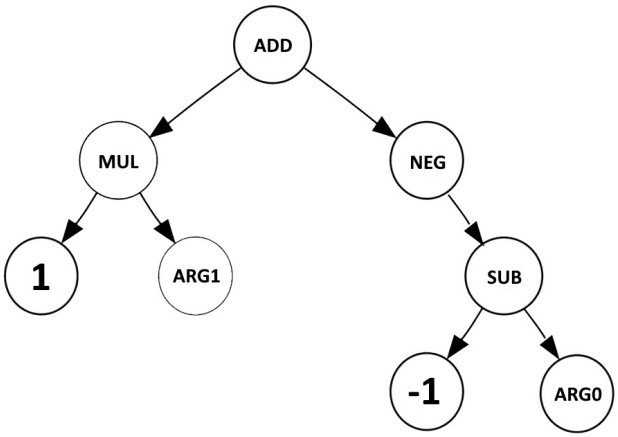

Figura 4.8: "$add(mul(1, ARG1), neg(sub(-1, ARG0)))$"

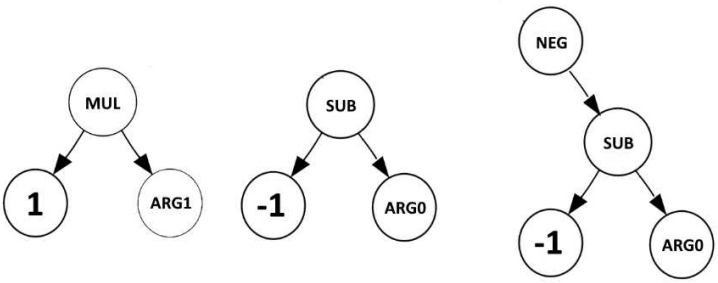

Figura 4.9: Sottomoduli che vengono estratti

Come mostra la figura 4.10, tramite la funzione "extraction" ogni individuo viene analizzato estraendo tramite espressioni regolari i moduli di profondità 1 e di profondità 2. Successivamente, ciascun sottomodulo viene salvato in un dizionario in base alla sua profondità, insieme a due valori: la frequenza con cui appare nei vari individui e la somma delle fitness degli individui in cui è presente.

```
#resistuisce i moduli/sottoalberi della popolazione
def get_modules(pop):
    my_dict1={}
    my_dict2={}
    for individual in pop:
        # funzione che estrae i moduli di profondità 1 e 2 da un albero(individuo) usando espressioni regolari
        module_depth1,module_depth2=extraction(str(individual))

        #per ogni modulo guardo se è già presente nel dizionario
        for m in module_depth1:
            if m not in my_dict1:
                #se non è ancora presente lo aggiungo
                my_dict1[m] = [1,individual.fitness.values[0]]
            else:
                # se è presente incremento la frequenza e la fitness
                my_dict1[m][0] += 1
                my_dict1[m][1] += individual.fitness.values[0]

        for m in module_depth2:
            if m not in my_dict2:
                my_dict2[m] = [1,individual.fitness.values[0]]
            else:
                my_dict2[m][0] += 1
                my_dict2[m][1] += individual.fitness.values[0]

    return my_dict1,my_dict2
```

Figura 4.10: Estrazione dei moduli di profondità 1 e 2 dagli individui della popolazione: la funzione restituisce due dizionari distinti

Una volta che si hanno questi due dizionari, si procede con la selezione dei sottomoduli da mantenere nelle iterazioni successive tramite la funzione "get_individual_to_keep" (figura 4.11) che prevede i seguenti passaggi:

1. I dizionari vengono ordinati in modo decrescente in funzione della frequenza di apparizione.

2. I 5 moduli più frequenti di entrambi i dizionari vengono inseriti in un terzo dizionario. In questo dizionario, al posto della somma delle fitness viene messa la media delle fitness, calcolato come $\frac{sommafitness}{frequenza}$. Inoltre viene effettuata la normalizzazione della media appena calcolata.

3. Questo dizionario viene ordinato in modo decrescente in funzione della media normalizzata delle fitness.

4. A questo punto, i moduli da mantenere nelle iterazioni successive sono i primi N moduli del dizionario creato, dove N è un parametro che decide l'utente tramite interfaccia grafica.

```python
def get_individuals_to_keep(pop,n, modules_depth1,modules_depth2):
    #ordinamento in funzione della frequenza
    sorted_modules_1 = dict(sorted(modules_depth1.items(), key=lambda x: x[1][0], reverse=True))
    sorted_modules_2 = dict(sorted(modules_depth2.items(), key=lambda x: x[1][0], reverse=True))
    # Estrapola i primi 5 elementi (quelli che si presentano di più), in questo modo potrò mantenere al massimo 10 individui
    modules_freq = {}
    for key, value in list(sorted_modules_1.items())[:5]:
        value[1] /= value[0] #media delle fitness
        modules_freq[key] = value
    for key, value in list(sorted_modules_2.items())[:5]:
        value[1] /= value[0]
        modules_freq[key] = value

    # normalizzazione delle fitness
    somma_fitnesses_modules = sum(v[1] for v in modules_freq.values())
    for module in modules_freq:
        if somma_fitnesses_modules != 0:
            modules_freq[module][1] = modules_freq[module][1] / somma_fitnesses_modules
        else:
            modules_freq[module][1] = 0

    #visualizzazione del grafico con frequenze e fitness associate ai moduli
    view_hist_fitness_freq(modules_freq)

    #ordinamento in funzione della fitness
    sorted_modules_fitness = dict(sorted(modules_freq.items(), key=lambda x: x[1][1], reverse=True))

    #selezione degli N individui da mantenere
    individuals_to_keep = []
    for module, value in sorted_modules_fitness.items():
        if len(individuals_to_keep)<n:
            individuals_to_keep.append(module)
        else:
            break

    return individuals_to_keep
```

Figura 4.11: Estrazione dei moduli di profondità 1 e 2 dagli individui della popolazione: la funzione restituisce due dizionari distinti

Applicando questa metodologia, i moduli da mantenere vengono scelti sia in base alla loro frequenza sia in base al valore medio di fitness ad essi associato. Successivamente, questi moduli vengono compilati per renderli funzioni eseguibili e vengono inseriti nel set delle primitive. In questo modo, possono essere utilizzati nelle iterazioni successive per creare gli individui della popolazione.

4.5 Salvataggio dell'individuo migliore

Dopo aver completato tutte le iterazioni, oltre al salvataggio dei risultati in un file di testo, il miglior individuo viene salvato utilizzando la libreria Pickle (figura 4.12). Questo individuo rappresenta una soluzione ottimale per il problema in questione e, salvandolo in un file, può essere utilizzato in

qualsiasi programma che richieda la risoluzione di tale problema.
In questo progetto è stato creato uno script indipendente (figura 4.13) che legge l'individuo migliore salvato in uno specifico file. Successivamente, l'individuo viene eseguito su un insieme di dati e i risultati vengono stampati. Tuttavia, per utilizzare il miglior individuo nello script, è necessario compilarlo per trasformarlo in una funzione valutabile. Durante la compilazione, è richiesto anche il pset (Primitive Set) utilizzato nella creazione dell'individuo durante l'algoritmo evolutivo. Pertanto, al termine delle iterazioni, oltre al salvataggio dell'individuo, viene salvato anche il pset (figura 4.12). Dato che il pset è composto da primitive, viene utilizzata la sottolibreria Dill di Pickle per la sua serializzazione.
Inoltre, per l'esecuzione dello script indipendente, saranno necessari anche il nome assegnato alle costanti effimere e il valore KERNEL_SIZE utilizzato. Per questo motivo, vengono salvati in un file di testo separato (figura 4.12) e richiamati nello script (figura 4.13).

```
now = datetime.datetime.now()
current_time = now.strftime("%Y-%m-%d_%H-%M-%S")

with open(f"best_individual_{current_time}.pickle", "wb") as f:
    pickle.dump(hof[N_ITERATIONS-1][0], f)

with open(f"pset_{current_time}.pkl", "wb") as p:
    dill.dump(pset, p)

with open(f"parameters_{current_time}.txt", "w") as r:
    r.write(str(const)+"\n")
    r.write(str(KERNEL_SIZE)+"\n")
```

Figura 4.12: Salvataggio dell'individuo, del pset e dei parametri in file identificati dalla data attuale

In conclusione, al termine di ogni esecuzione dell'algoritmo, si avranno i tre seguenti file (i nomi sono generici e ognuno sarà caratterizzato dalla data attuale):

1. **'best_individual.pkl'**: il file contenente l'individuo migliore serializzato utilizzando la libreria Pickle. Questo file rappresenta la soluzio-

ne ottimale al problema considerato e può essere utilizzato in altri programmi.

2. **pset.pkl**:il file contenente il pset serializzato utilizzando la libreria Dill di Pickle. Il pset è composto da primitive e rappresenta l'insieme di funzioni e operatori utilizzati nella creazione dell'individuo migliore durante l'algoritmo evolutivo.

3. **parameters.txt**: il file fornisce i parametri aggiuntivi necessari per l'esecuzione dello script indipendente. Le informazioni includono il nome assegnato alle costanti effimere e il valore di KERNEL_SIZE.

Questi tre file verranno usati per riprodurre l'ambiente in cui l'individuo migliore è stato generato e per eseguire lo script indipendente utilizzando l'individuo compilato e i parametri corretti.

```
#lettura dei parametri, del pset e dell'individuo
with open("parameters.txt", "r") as r:
    const=int(r.readline())
    KERNEL_SIZE=int(r.readline())

pset = gp.PrimitiveSet("MAIN", KERNEL_SIZE)
pset.addEphemeralConstant(f"rand101_{const}", lambda: random.randint(-1, 1))

with open("pset.pkl", "rb") as f:
    pset = dill.load(f)

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)


with open("best_individual.pickle", "rb") as f:
    best_individual = pickle.load(f)

best_individual = creator.Individual(best_individual)

#set di dati arbitrario
data=test_data[20:40]
labels=test_labels[20:40]

#funzione di valutazione
f1=eval(best_individual,data,labels,pset)
#stampa dei risultati
print(f1)
```

Figura 4.13: Script esterno in cui vengono letti i file necessari. La funzione di valutazione "eval" è la stessa discussa in precedenza

4.6 Esempio di esecuzione

In questo paragrafo è simulata l'esecuzione dell'algoritmo passo per passo, mostrando i risultati ottenuti.

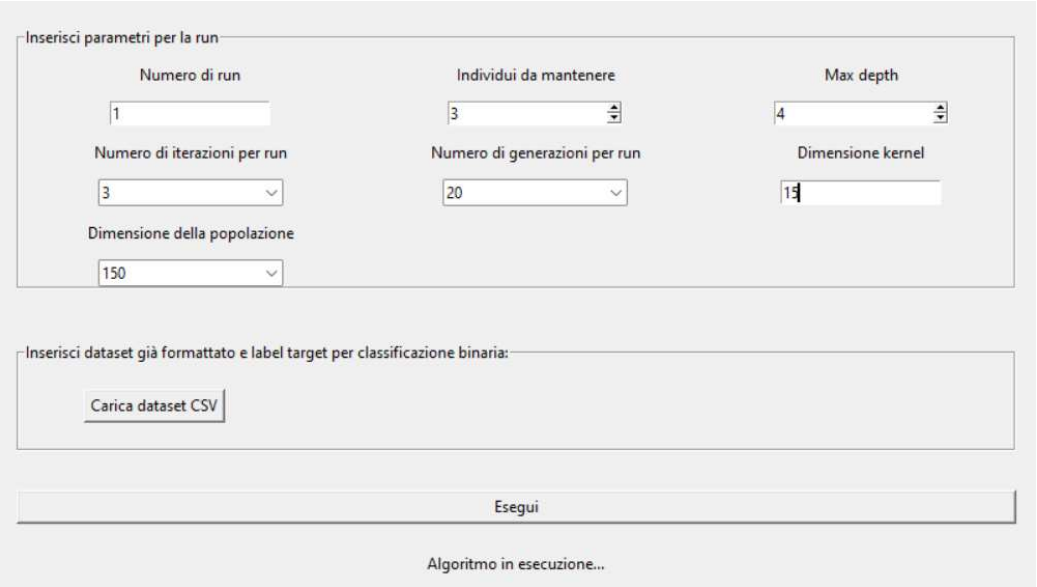

Figura 4.14: Passo 1: Interfaccia in cui vengono settati parametri e dataset

1. **Avvio dell'interfaccia grafica** (figura 4.14) : L'utente inserisce i parametri dell'algoritmo e il dataset su cui lavorare. Cliccando sul tasto esegui viene mandato in esecuzione l'algoritmo genetico con i parametri e dataset impostati. La figura mostra l'interfaccia e il messaggio di informazione sullo stato dell'algoritmo.

2. **Esecuzione dell'algoritmo**: Durante l'esecuzione dell'algoritmo sono stampate a terminale le statistiche e i valori dell'F1 score.
   Ad ogni fine iterazione, viene valutato l'individuo migliore calcolando l'F1 score sul validation set e sul test set.
   Il risultato della valutazione è visualizzato anche tramite la matrice di confusione (esempio in figura 4.15).
   Inoltre viene visualizzato un grafico che riporta sia la frequenza che la

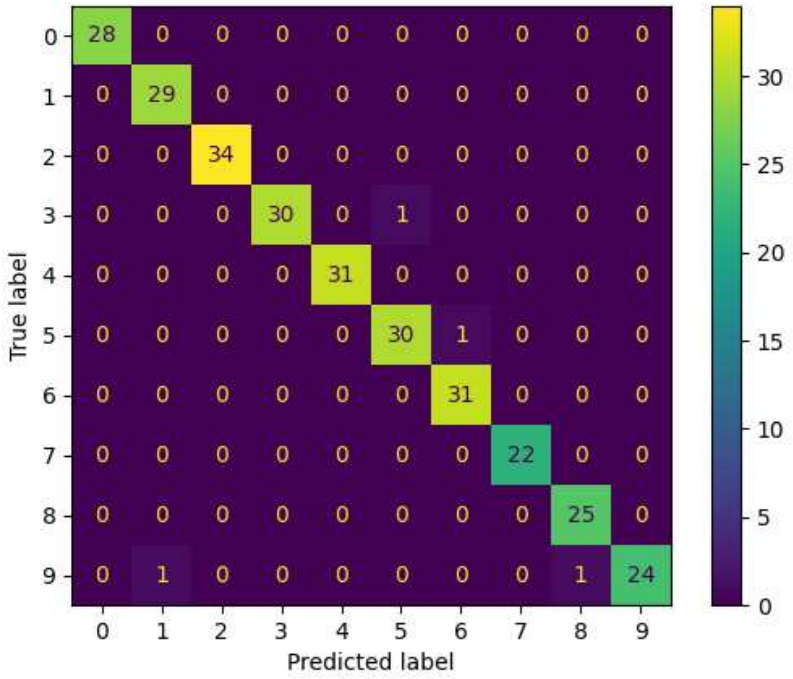

Figura 4.15: Passo 2: visualizzazione della matrice di confusione calcolata su test set o su validation set

fitness media normalizzata dei moduli tra cui selezionare gli individui da mantenere nell'iterazione successiva (figura 4.16).
In questo modo si può vedere se i moduli più frequenti sono anche quelli con fitness migliore.

3. **Fine esecuzione**: Al termine dell'esecuzione viene salvato l'individuo migliore, il pset e i parametri necessari per eseguire lo script esterno. Inoltre, vengono salvati i risultati in un file di testo. In questo file di testo sono riportati i parametri usati, le statistiche di ogni iterazione e i valori dell'F1 score su test set e validation set di ogni iterazione (esempio in figura 4.17).

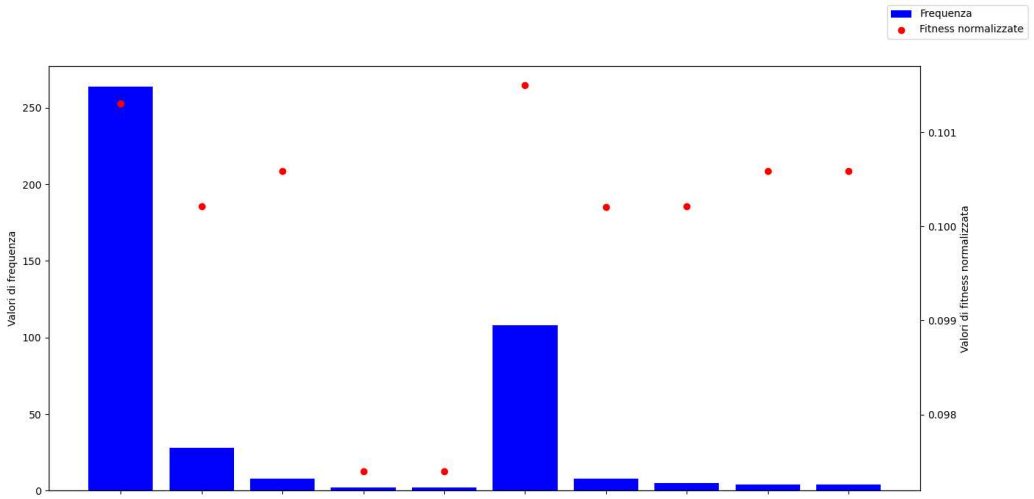

Figura 4.16: Passo 2: Visualizzazione istogramma delle frequenze e delle fitness associate ai moduli

Statistiche iterazione 1:

		fitness						size					
gen	nevals	avg	gen	max	min	nevals	std	avg	gen	max	min	nevals	std
0	149	0.947776	0	0.98703	0.800493	149	0.024498	24.8667	0	31	8	149	5.27215
1	76	0.968586	1	0.98703	0.929277	76	0.00988811	23.3533	1	31	7	76	6.30993
2	92	0.975865	2	0.98703	0.876924	92	0.0115074	19.52	2	31	2	92	7.0083
3	92	0.978204	3	0.987697	0.931016	92	0.00922469	17.3067	3	30	2	92	6.54212
4	91	0.979016	4	0.987697	0.948643	91	0.00964942	15.8667	4	30	4	91	7.90583
5	76	0.980983	5	0.987697	0.948693	76	0.00920599	18.1067	5	30	2	76	10.0851
6	91	0.983492	6	0.988212	0.946565	91	0.00820577	25.3	6	30	6	91	6.61488
7	97	0.982369	7	0.988311	0.948334	97	0.00989287	23.92	7	30	8	97	4.90241
8	87	0.98169	8	0.988311	0.944037	87	0.0111524	20.76	8	28	6	87	3.50177
9	86	0.982917	9	0.988409	0.930676	86	0.0103955	18.9133	9	29	5	86	2.93924
10	78	0.983776	10	0.988508	0.925635	78	0.00966471	17.5133	10	26	5	78	2.59932
11	62	0.98545	11	0.988903	0.952195	62	0.00718801	16.2733	11	19	5	62	2.09729
12	90	0.983783	12	0.988903	0.90169	90	0.0115817	15.1867	12	30	3	90	2.61632
13	92	0.977063	13	0.989101	0.0182892	92	0.0812292	13.4733	13	31	3	92	3.29383
14	105	0.983367	14	0.989298	0.944243	105	0.00950215	11.9333	14	31	5	105	3.27753
15	80	0.984411	15	0.989298	0.8777	80	0.0127401	10.6267	15	28	3	80	2.89493
16	88	0.984591	16	0.989496	0.854113	88	0.0129932	9.31333	16	29	3	88	2.58492
17	80	0.986124	17	0.989496	0.96057	80	0.00667994	8.02667	17	15	3	80	1.92855
18	87	0.984668	18	0.989496	0.943786	87	0.00888441	6.96	18	31	3	87	3.35237
19	71	0.985677	19	0.989496	0.944907	71	0.00699744	5.57333	19	23	3	71	2.05701
20	93	0.983697	20	0.989496	0.960799	93	0.0079022	5.04	20	9	3	93	1.01902

F1 on test set dell'iterazione 1: 0.9631157938033713

Figura 4.17: Passo 3: Esempio di statistiche di una esecuzione, salvate nel file di testo a fine algoritmo

4.7 Altri approcci e differenze con l'algoritmo implementato

Inizialmente è stato sviluppato un altro algoritmo che presenta alcune differenze dall'approccio di cui abbiamo precedentemente parlato.
Innanzitutto, anziché inizializzare il primitive set con KERNEL_SIZE argomenti, viene utilizzato un singolo argomento. Inoltre, nella funzione di

valutazione, anziché applicare direttamente il programma genetico generato dall'individuo a una finestra del segnale (facendo poi scorrere la finestra), viene generato un kernel utilizzando l'individuo che successivamente viene convoluto con ogni segnale (funzione "get_kernel" riportata in figura 4.18).

```
def get_kernel(individual, kernel_size: int):
    func = toolbox.compile(expr=individual)
    kernel = []
    for x in range(0, kernel_size + 1):
        kernel.append(func(x))
    kernel=np.array(kernel)
    return kernel

def evalTrainingSet(individual):
    kernel = get_kernel(individual, KERNEL_SIZE)
    new_train_set = convolution(kernel, train_data)
    new_val_set = convolution(kernel, data_val)
    f1_validation=training_RF(new_train_set,train_labels, new_val_set, labels_val,target)
    return (f1_validation,)
```

Figura 4.18: Funzione di valutazione nell'approccio iniziale, in cui viene generato il kernel e fatta la semplice convoluzione con i segnali

Un'altra differenza significativa è che vengono estratti tutti i sottomoduli degli individui, non solo quelli di profondità 1 e 2. Questa estrazione viene effettuata trasformando l'individuo in un formato particolare in modo tale da poter usare la libreria NLTK per estrarre i vari sottomoduli con la funzione ParentedTree.
Una volta estratti tutti i sottomoduli presenti nella popolazione si considerano solo i 5 più frequenti.
Lo scopo dell'algoritmo è inserire nel set delle primitive N individui che presentano almeno uno di questi 5 sottomoduli più frequenti e che siano il più diversi possibili tra di loro.
L'algoritmo per selezionare individui più diversi tra di loro applica la tecnica di PCA e di clustering k-means (figura 4.19). La PCA (Principal Component Analysis) è una tecnica di riduzione della dimensionalità che identifica le principali componenti dei dati, mentre k-means è un algoritmo di clustering che raggruppa i dati in cluster simili, in modo da selezionare in ciascun cluster un individuo il più diverso possibile dagli altri.

In sostanza, dopo aver selezionato gli individui che presentano almeno uno dei 5 sottomoduli più frequenti, l'algoritmo applica su questi la PCA e il clustering k-means per selezionare gli N individui più diversi tra loro da inserire nel set delle primitive, in modo tale da poterli riutilizzare nelle iterazioni successive.

```python
#Selects n individuals to keep, as different as possible among individuals that have the most frequent modules as subtrees
def get_individuals_to_keep(pop, n, modules,  cntTree, cntBest):
    pop_kernels = []
    population_with_modules_freq=[]
    #aggiungo a pop_kernel solo il kernel degli individui della popolazione che presentano almeno uno dei moduli più frequenti,
    # salvandomi questi individui dentro a population_with_modules_freq
    for p in pop:
        p_modules= extract_modules(str(p),  cntTree, cntBest)
        for module in modules:
            if module in p_modules:
                population_with_modules_freq.append(p)
                pop_kernels.append(get_kernel(p, KERNEL_SIZE))
                break
    pop_kernels = list(set(tuple(row) for row in pop_kernels))
    pop_kernels = np.array(pop_kernels)
    # using PCA to extract the two principal components from the kernels, reducing each one to a point (x, y)
    pca = PCA(n_components=2)
    pca.fit(pop_kernels)
    pop_kernels = pca.transform(pop_kernels)
    # applying KMeans to find the clusters
    try:
        kmeans = KMeans(n_clusters=N_CLUSTERS, random_state=0, n_init="auto").fit(pop_kernels)
    except ValueError:
        print("Warning: fewer samples than clusters, reducing number of clusters to be equal to the number of samples")
        kmeans = KMeans(n_clusters=len(pop_kernels), random_state=0, n_init="auto").fit(pop_kernels)

    #kmeans = KMeans(n_clusters=N_CLUSTERS, random_state=0, n_init="auto").fit(pop_kernels)
    kernel_labels = kmeans.labels_
    individuals_to_keep = []
    for i in range(N_CLUSTERS):
        kept = 0
        trovato=False
        for j in range(len(individuals_to_keep)):
            if(str(population_with_modules_freq[i])==str(individuals_to_keep[j])):
                trovato=True
                break

        if(trovato==False):
            for j in range(len(kernel_labels)):
                if kernel_labels[j] == i:
                    kept += 1
                    individuals_to_keep.append(population_with_modules_freq[i])
                    if kept == N_IND_FOR_CLUSTER:
                        break
        # if one or more clusters were empty, select some random individuals from the population
        if kept < N_IND_FOR_CLUSTER:
            for i in range(N_IND_FOR_CLUSTER - kept):
                individuals_to_keep.append(random.choice(pop))

    return individuals_to_keep
```

Figura 4.19: Funzione per decidere quali individui inserire nel pset

Capitolo 5

Risultati

Questo capitolo presenta i risultati ottenuti dall'implementazione dell'algoritmo basato sulla programmazione genetica modulare per la classificazione dei segnali presenti nel dataset Digit e nel dataset PTB delle aritmie. Lo scopo principale di questo studio è valutare se l'approccio modulare possa migliorare la classificazione dei segnali rispetto a due approcci alternativi quali l'applicazione di un classificatore Random Forest direttamente al set di dati originali e l'utilizzo della forma più tradizionale di programmazione genetica, in cui ogni individuo rappresenta un programma genetico utilizzato per generare un nuovo set di dati ma senza l'introduzione dei moduli nel set delle primitive.
L'utilizzo di un'interfaccia grafica ha permesso di eseguire l'algoritmo con diverse combinazioni di parametri. Gli utenti possono impostare i seguenti parametri dell'algoritmo: dimensione del kernel, numero di generazioni, numero di iterazioni dell'algoritmo evolutivo, massima profondità dell'albero, numero di individui nella popolazione e numero di sottomoduli da inserire ad ogni iterazione nel set delle primitive.
Dopo aver condotto diverse esecuzioni sperimentali, è emerso che l'utilizzo di valori relativamente piccoli per la dimensione del kernel ha portato a risultati migliori nella classificazione dei segnali. Al contrario, un aumento della dimensione del kernel ha generato valori di prestazione inferiori.

Inoltre, aumentare il numero di individui nella popolazione e il numero di generazioni ha spesso portato a miglioramenti nelle prestazioni. Tuttavia, è importante notare che ciò comporta un aumento dei tempi di esecuzione dell'algoritmo.
L'aumento della profondità dell'albero non ha influenzato significativamente i risultati, mentre il numero di iterazioni massimo usato è stato 3 per evitare che la dimensione degli individui modulari diventasse eccessivamente grande. Nelle seguenti tabelle dei risultati questi due parametri, profondità massima dell'albero e numero delle iterazioni, non saranno riportati poiché i loro valori saranno rispettivamente 4 e 3.
La prima iterazione rappresenta la semplice applicazione della programmazione genetica, mentre la seconda e la terza iterazione presentano l'approccio modulare.
Per quanto riguarda l'approccio modulare, è stato osservato che il numero minimo di sottomoduli da mantenere durante l'evoluzione dell'algoritmo dovrebbe essere superiore o uguale a tre al fine di ottenere risultati significativi. L'obiettivo è mostrare i risultati relativi a quattro diversi approcci e confrontarli al fine di determinare se l'approccio modulare abbia portato a miglioramenti.
I quattro approcci analizzati sono:

- L'applicazione diretta del classificatore Random Forest ai segnali dei dataset originali.
- La programmazione genetica con una singola iterazione
- Il primo approccio alla programmazione genetica modulare in cui i moduli sono interi individui selezionati tramite il classificatore k-means
- L'approccio di base di questa tesi, in cui la modularità è ottenuta da moduli di profondità 1 e 2 estratti dagli individui migliori.

5.1 Risultati sul dataset PTB delle aritmie

Di seguito vengono riportate delle tabelle riportanti i risultati più significativi ottenuti lavorando sul dataset fornito da PTB composto da una serie di segnali ECG e classificati in battiti normali e battiti anormali (aritmia).
Il primo confronto, riportato nella tabella in figura 5.1, viene fatto tra i due approcci alla programmazione genetica modulare, notando che il lavoro su cui si basa questa tesi porta a risultati migliori rispetto al primo approccio utilizzato. Questa tabella riporta alcune esecuzioni per mostrare che, indipendentemente dalla combinazione dei parametri utilizzati, i risultati di F1_score sono costantemente inferiori per il primo approccio.

Kernel_size	Population_size	N° generazioni	N° sottomoduli	F1 test	F1 val	F1 test 1°	F1 val 1°
5	50	20	10	0.9139	0.92852	0.911	0.919
10	50	20	10	0.9241	0.9316	0.91	0.928
35	50	20	10	0.9203	0.9277	0.85	0.86
50	50	20	3	0.9178	0.9145	0.83	0.85
65	150	20	5	0.917	0.923	0.84	0.90
65	50	20	5	0.891	0.9239	0.84	0.85
125	50	20	3	0.846	0.9059	0.735	0.782

Figura 5.1: Confronto tra primo e secondo approccio alla programmazione genetica modulare

La tabella in figura 5.2 riporta le esecuzioni effettuate evidenziando i casi in cui l'approccio modulare ha prodotto risultati migliori rispetto alla programmazione genetica standard. Osservare risultati di F1_score più elevati dalla seconda iterazione in poi indica che l'approccio modulare apporta miglioramenti rispetto alla programmazione genetica standard. Al contrario, se i risultati non cambiano dopo la prima iterazione, significa che la programmazione genetica modulare non porta a miglioramenti né a peggioramenti.
Come si può notare non è sempre vero che l'approccio modulare funzioni meglio che lo standard, però ci sono state molte esecuzioni che hanno portato a benefici.

Kernel-size	Moduli da mantenere	N Generazioni	Pop_size	N ITERAZIONI	F1 test GP	F1 val GP	F1 test GPM	F1 val GPM
5	5	20	50	2	0.86972	0.90949	0.86972	0.90949
5	10	20	150	2	0.89639	0.93737	0.89639	0.93737
5	10	20	50	3	0.9045	0.91985	0.91393	0.92852
5	5	20	150	3	0.90702	0.93244	0.90702	0.93244
10	3	20	50	3	0.89999	0.9281	0.89999	0.9281
10	5	20	50	3	0.92142	0.95968	0.92142	0.95968
10	5	20	150	3	0.89997	0.9419	0.89997	0.9419
10	10	20	50	3	0.9241	0.93168	0.9241	0.93168
15	3	20	50	3	0.924	0.9247	0.92848	0.93745
15	5	20	50	3	0.89948	0.9241	0.89948	0.9241
15	5	30	150	3	0.91284	0.96418	0.91284	0.96418
15	10	20	50	3	0.89995	0.93303	0.89995	0.93303
15	3	20	150	3	0.89883	0.91954	0.89883	0.91954
25	5	20	50	2	0.8785	0,9553	0.8892	0.9776
35	10	20	50	2	0.92038	0.92774	0.92038	0.92774
35	3	20	50	2	0.90706	0.95086	0.90706	0.95086
35	5	20	150	2	0.89285	0.94639	0.89285	0.94639
35	3	20	50	2	0.8677	0.9104	0.87857	0.92392
35	5	20	50	2	0.89913	0.9459	0.89913	0.9459
50	3	20	50	3	0.87	0.885	0.9178	0.91456
50	5	20	50	3	0.90342	0.87491	0.90342	0.87491
50	10	20	50	2	0.88211	0.90765	0.88887	0.91497
50	5	20	50	2	0.87102	0.9277	0.87847	0.94134
50	3	20	50	2	0.87	0.885	0.888	0.889
50	3	20	150	3	0.91428	0.94639	0.91428	0.94639
51	10	30	50	2	0.91062	0.92398	0.91062	0.92398
65	5	20	150	2	0.917	0.923	0.917	0.923
65	5	20	50	2	0.88662	0.9096	0.89177	0.92398
65	10	20	50	2	0.90177	0.90887	0.90177	0.90887
85	5	20	50	2	0.89006	0.91126	0.9142	0.92392
125	3	20	50	3	0.84619	0.90593	0.84619	0.90593
125	10	20	50	2	0.90166	0.90987	0.90166	0.90987
125	10	20	50	2	0,92	0,93	0,92	0,93
125	7	20	150	2	0.86422	0.96428	0.86422	0.96428

Figura 5.2: Esperimenti svolti. In giallo vengono evidenziate le esecuzioni in cui la programmazione genetica modulare migliora i risultati di F1_score rispetto a programmazione genetica standard

Le dimensioni del kernel che portano la programmazione genetica modulare ad avere risultati maggiori rispetto alla standard sono quelle con valori medi: le esecuzioni in cui la modularità migliora i risultati sono quelle con valori di kernel intorno a 50. I risultati di F1 sono abbastanza simili al variare delle dimensioni del kernel, anche se si osservano valori in media più elevati con kernel relativamente piccoli.
Per quanto riguarda il miglioramento di F1 rispetto al classificatore Random Forest, la tabella in figura 5.3 mostra, usando gli stessi test set e validation set, il confronto tra F1_score ottenuto con programmazione genetica standard (GP), F1_score ottenuto con programmazione genetica modulare (GPM) e F1_score ottenuto dal classificatore Random Forest sui dati originali.

Kernel size	Pop size	N° generazioni	N° sottomoduli	F1 test GP	F1 val GP	F1 test GPM	F1 val GPM	F1 test RF	F1 val RF
3	50	20	3	0.8744	0.9411	0.8744	0.941	0.8707	0.895
5	150	20	5	0.892	0.95	0.892	0.95	0.889	0.905
10	50	20	10	0.9142	0.9464	0.9142	0.946	0.885	0.888
15	50	20	3	0.9247	0.924	0.9284	0.937	0.8889	0.888
15	150	20	3	0.903	0.928	0.903	0.928	0.86	0.88
15	50	30	5	0.899	0.919	0.899	0.919	0.889	0.864
20	50	20	3	0.9249	0.9374	0.9249	0.9374	0.899	0.91
25	50	20	5	0.8785	0.9553	0.8892	0.977	0.878	0.906
25	50	20	10	0.896	0.937	0.896	0.937	0.894	0.892
35	50	20	3	0.8677	0.9104	0.8785	0.923	0.8499	0.901
50	50	20	3	0.87	0.885	0.888	0.889	0.882	0.856
55	50	20	10	0.863	0.932	0.863	0.932	0.888	0.883
60	50	20	5	0.877	0.9193	0.877	0.919	0.90	0.87
85	50	20	5	0.907	0.914	0.914	0.923	0.89	0.87

Figura 5.3: Confronto tra GP, GPM e RF. In arancione sono evidenziate alcune esecuzioni in cui l'approccio modulare (GPM) è migliore rispetto sia alla programmazione genetica standard (GP) che al Random Forest (RF). In blu sono riportate alcune esecuzioni in cui GPM è uguale a GP, ma migliore rispetto a RF. Mentre quelle in bianco sono esecuzioni in cui GPM non porta ad alcun miglioramento.

Dai risultati presentati nella tabella della figura 5.3, si può notare che l'applicazione della programmazione genetica porta quasi sempre a risultati migliori rispetto all'utilizzo di un Random Forest sui dati originali, soprattutto con valori della dimensione del kernel relativamente piccoli. Ciò suggerisce che l'uso dell'algoritmo genetico può migliorare la capacità di classificazione dei segnali. Questo è anche evidenziato dal fatto che, al termine dell'algoritmo, l'individuo migliore identificato non è mai la funzione di identità inserita all'inizio dell'algoritmo.
La funzione di identità nella convoluzione non altera i dati a cui viene applicata, mantenendo quindi i dati originali. Questo dimostra che l'approccio utilizzato produce risultati superiori all'applicazione del Random Forest sui dati originali.
L'approccio modulare non evidenzia un miglioramento sistematico rispetto alla programmazione genetica standard, tuttavia, sono stati osservati risultati superiori in diverse esecuzioni come si osserva anche dalla tabella 5.2.

5.2 Risultati sul dataset Digit

Di seguito vengono presentate le tabelle con i risultati più significativi ottenuti lavorando sul dataset Digit.
Prima di tutto si può evidenziare come i risultati del primo approccio utilizzato, anche in questo caso, sono inferiori rispetto all'approccio modulare alla base di questo lavoro di tesi.
Nella tabella in figura 5.4 viene presentato il confronto dei due metodi da cui emerge che, indipendentemente dalla combinazione dei parametri utilizzati, i risultati di F1_score sono costantemente inferiori per il primo approccio.

Kernel size	Population size	N° generazioni	N° sottomoduli	F1 test	F1 val	F1 test 1°	F1 val 1°
15	50	20	10	0.96565	0.9833	0.94	0.95
15	150	40	5	0.971	0.989	0.94	0.97
20	50	20	5	0.97009	0.9895	0.95	0.97
20	150	20	3	0.96355	0.9926	0.95	0.96
25	50	20	5	0.96864	0.9795	0.93	0.94
35	150	20	5	0.9526	0.9701	0.87	0.89
50	50	20	2	0.9317	0.9599	0.62	0.67

Figura 5.4: Confronto dei risultati tra primo e secondo approccio alla programmazione genetica modulare.

La tabella in figura 5.5 riporta le esecuzioni effettuate evidenziando i casi in cui l'approccio modulare ha prodotto risultati migliori rispetto alla programmazione genetica standard. Come si può notare non è sempre vero che l'approccio modulare funzioni meglio rispetto all'approccio standard, però ci sono state molte esecuzioni che hanno portato a miglioramenti.
I risultati migliori si ottengono con una dimensione del kernel né troppo piccola né troppo grande. Infatti, le esecuzioni che hanno la dimensione del kernel nel range tra 15 e 25 hanno fornito risultati più elevati, mentre quelle con kernel maggiori hanno portato a risultati inferiori evidenziando che la dimensione del kernel è sicuramente il parametro che influenza maggiormente il risultato.
Si può osservare che l'aumento della popolazione in alcune esecuzioni porta

a miglioramenti, come anche l'aumento delle generazioni, anche se in questo caso, dopo un certo numero di generazioni il valore di F1 non aumenta ulteriormente.

Kernel-size	Moduli da mantenere	N Generazioni	Pop_size	F1 test GP	F1 val GP	F1 test GPM	F1 val GPM
5	2	20	50	0.9535	0.9869	0.9535	0.9869
5	5	20	50	0.9627	0.9902	0.9627	0.9902
5	10	20	50	0.9522	0.9712	0.9602	0.9894
10	3	20	50	0.9578	0.9837	0.9578	0.9837
10	5	30	150	0.9835	0.9967	0.9835	0.9967
10	3	20	150	0.9752	0.9822	0.9752	0.9822
10	10	30	150	0.9488	0.9856	0.9488	0.9856
15	3	20	50	0.9655	0.9702	0.9773	0.9711
15	3	20	150	0.9571	0.9862	0.9571	0.9862
15	3	20	150	0.9839	0.9884	0.9839	0.9884
15	5	40	150	0.9654	0.9812	0.971	0.989
15	5	20	50	0.9737	0.9897	0.9737	0.9897
15	5	20	150	0.9614	0.9824	0.9614	0.9824
15	10	20	50	0.9599	0.9723	0.9656	0.9833
15	3	20	150	0.9631	0.9899	0.9631	0.9899
20	2	20	50	0.9601	0.9728	0.9601	0.9728
20	5	20	50	0.9622	0.9809	0.97	0.9895
20	10	20	50	0.9704	0.9836	0.9704	0.9836
20	10	20	150	0.9638	0.9773	0.9638	0.9773
20	3	20	150	0.9602	0.9722	0.9635	0.9926
25	2	20	50	0.9665	0.9792	0.9665	0.9792
25	5	20	50	0.9612	0.9708	0.9686	0.9795
25	5	50	300	0.972	0.9851	0.972	0.9851
25	10	20	50	0.9504	0.9865	0.9504	0.9865
25	5	20	150	0.9561	0.9906	0.9561	0.9906
25	3	30	150	0.9626	0.9759	0.9626	0.9759
35	3	20	50	0.9553	0.9682	0.9553	0.9682
35	5	20	50	0.9535	0.9893	0.9535	0.9893
35	10	20	50	0.9562	0.9699	0.9562	0.9699
35	3	20	50	0.9393	0.968	0.9393	0.968
35	5	20	150	0.9489	0.9655	0.9526	0.9701
35	10	20	150	0.9481	0.9825	0.9481	0.9825
45	3	20	150	0.9676	0.9753	0.9676	0.9753
45	2	20	50	0.9167	0.9583	0.9167	0.9583
45	5	20	50	0.9306	0.9561	0.9306	0.9561
45	10	20	50	0.9088	0.9586	0.9088	0.9586
50	2	20	50	0.9245	0.9532	0.9317	0.9599
50	5	20	50	0.9421	0.9229	0.9421	0.9229
50	10	20	50	0.9375	0.9533	0.9375	0.9533
50	2	20	150	0.9195	0.9759	0.9195	0.9759

Figura 5.5: Esecuzioni di GP modulare. In giallo vengono evidenziate le esecuzioni in cui la programmazione genetica modulare migliora i risultati di F1_score rispetto alla programmazione genetica standard.

Per quanto riguarda invece il miglioramento rispetto al classificatore Random Forest, la tabella in figura 5.6 mostra, usando gli stessi test set e validation set, il confronto tra F1_score ottenuto con GP, con GPM e con il classificatore Random Forest sui dati originali.

Kernel size	Pop size	N° generazioni	N° sottomoduli	F1 test GP	F1 val GP	F1 test GPM	F1 val GPM	F1 test RF	F1 val RF
5	50	20	10	0.95821	0.98829	0.96026	0.98944	0.95788	0.98074
5	150	40	10	0.95724	0.9930	0.95724	0.9930	0.9701	0.9713
10	50	20	3	0.9578	0.98379	0.9578	0.98379	0.9633	0.9802
10	150	30	5	0.98356	0.99676	0.98356	0.99676	0.9722	0.9779
15	150	20	3	0.95718	0.98621	0.95718	0.98621	0.9521	0.9633
15	50	20	5	0.9736	0.98212	0.9736	0.98212	0.97	0.9773
15	50	20	10	0.9551	0.97638	0.9602	0.97966	0.9655	0.9755
15	50	20	3	0.96587	0.97095	0.97735	0.97115	0.9718	0.96576
20	50	20	5	0.96895	0.96968	0.96895	0.96968	0.973	0.9719
25	300	50	5	0.97206	0.9851	0.97206	0.9851	0.968	0.972

Figura 5.6: Confronto di F1_score di alcune esecuzioni con i tre approcci differenti. In arancione sono evidenziate le esecuzioni in cui l'approccio modulare (GPM) è migliore rispetto sia alla programmazione genetica standard (GP) che al Random Forest (RF). In blu sono riportate alcune esecuzioni in cui GPM è uguale a GP, ma migliore rispetto a RF. Quelle in grigio sono esecuzioni in cui GPM è migliore rispetto a GP, ma peggiore rispetto a RF. Quelle in bianco sono esecuzioni in cui GPM non porta alcun miglioramento.

I risultati riportati nella tabella 5.6 evidenziano che, rispetto ai risultati ottenuti utilizzando il dataset di segnali ECG, la programmazione genetica mostra meno frequentemente un miglioramento nella misura F1 rispetto all'applicazione del Random Forest sui dati originali. Questo può essere attribuito al fatto che l'algoritmo implementato è progettato per operare su segnali, mentre nel caso del dataset Digit i dati consistono in immagini che vengono trasformate in formato monodimensionale.
Nonostante i risultati della programmazione genetica non siano sempre migliori rispetto al Random Forest, in media la programmazione genetica ottiene risultati superiori al Random Forest in 6 casi su 10, mantenendo comunque una performance simile a RF quando si verifica un peggioramento.
L'approccio modulare non evidenzia un miglioramento sistematico rispetto alla programmazione genetica standard; tuttavia, sono stati osservati risultati migliori in diverse esecuzioni come si osserva anche dalla tabella 5.5.

5.3 Visualizzazione dei risultati

Di seguito vengono riportate le matrici di confusione, di un esempio di esecuzione, relative al classificatore Random Forest sul validation set e sul test set originali (Figura 5.7 per dataset su ECG, figura 5.8 per dataset Digit) e all'approccio della programmazione genetica (Figura 5.9 per dataset su ECG, figura 5.10 per dataset Digit), in cui il programma genetico rappresentato da ogni individuo viene utilizzato per generare il nuovo set di dati. Queste figure offrono una rappresentazione visiva dei risultati ottenuti in un'esecuzione in cui la GP mostra un miglioramento rispetto all'applicazione del classificatore RF sui dati originali.

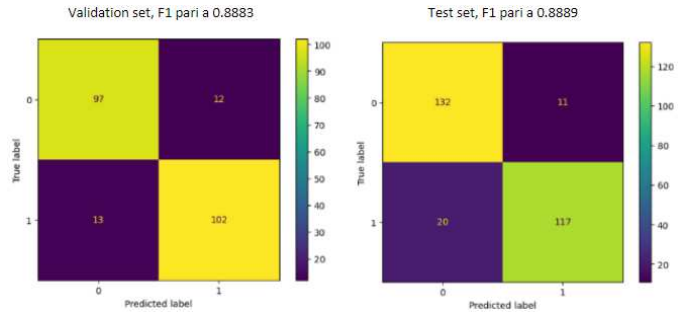

Figura 5.7: Matrici di confusione relative alla predizione del RF su validation set e test set originali del dataset di segnali ECG.

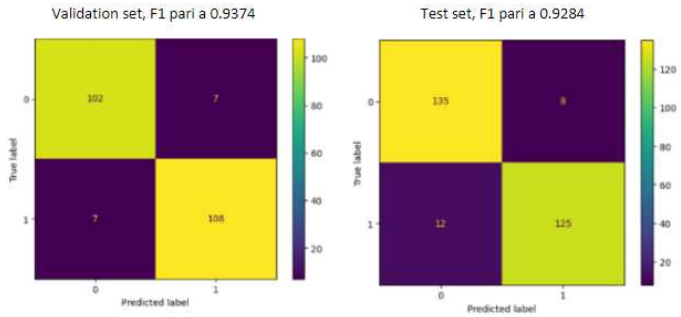

Figura 5.8: A sinistra la matrice di confusione, del dataset di segnali ECG, relativa alla predizione del RF su validation set ottenuto con GP, mentre a destra quella sul test set ottenuto con GP.

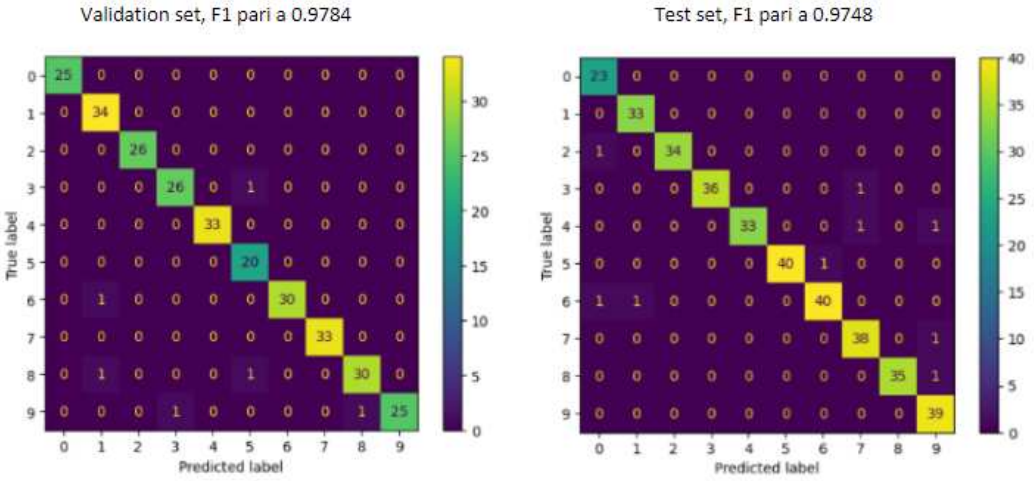

Figura 5.9: Matrici di confusione relative alla predizione del RF su validation set e test set originali del dataset Digit.

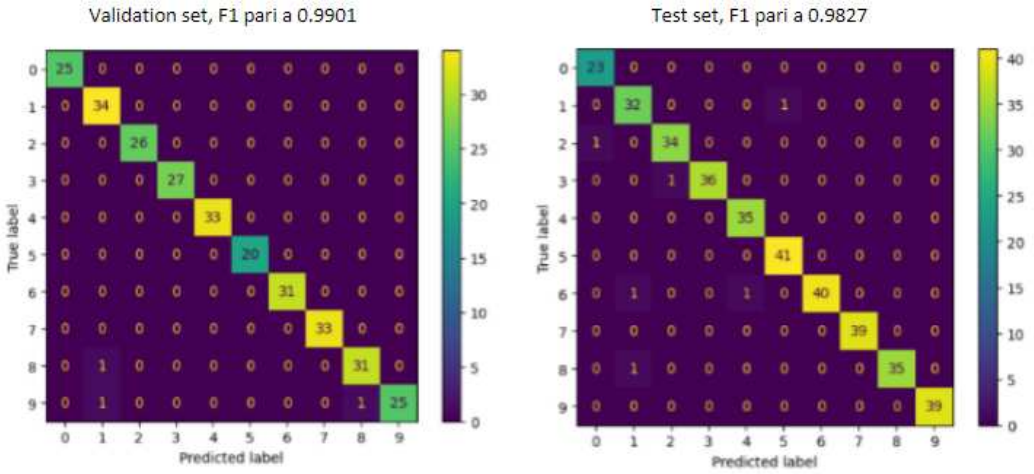

Figura 5.10: A sinistra la matrice di confusione, del dataset Digit, relativa alla predizione del RF su validation set ottenuto con GP, mentre a destra quella sul test set ottenuto con GP.

Come si può notare dalle figure, le predizioni corrette sono maggiori nell'approccio con la programmazione genetica modulare che ottiene un valore di F1 score più elevato

Conclusioni

Alla luce dei risultati ottenuti e delle analisi condotte, è possibile trarre conclusioni riguardo all'efficacia della programmazione genetica modulare per la classificazione di segnali.
Gli obiettivi di questo progetto sono stati due. Il primo è stato utilizzare la programmazione genetica (GP) per effettuare una pre-elaborazione dei dataset al fine di migliorare le prestazioni di un classificatore Random Forest applicato direttamente sui segnali dei dataset originali. Il secondo obiettivo è stato realizzare un approccio modulare basato sulla GP per effettuare questa pre-elaborazione. In particolare, è stata implementata una versione di GP modulare che ha semplificato l'implementazione su FPGA del sistema, rendendo più compatte sia la ricerca che la descrizione degli alberi del sistema grazie all'utilizzo di moduli che implementano funzioni di alto livello.
I dataset specifici utilizzati sono stati il dataset Digit e il dataset PTB di segnali ECG.
Dai risultati riportati nel capitolo "Risultati", emerge chiaramente che l'applicazione del Random Forest ai dati originali può fornire buoni risultati, soprattutto per quanto riguarda il dataset Digit, e con un tempo di computazione più rapido rispetto alla programmazione genetica. Tuttavia, è importante notare che in alcune esecuzioni su identici set di allenamento, validazione e test, il Random Forest applicato ai dati originali ha ottenuto risultati inferiori in termini di F1-score rispetto alla programmazione genetica. Questo suggerisce che la programmazione genetica, tramite la trasformazione del set di dati originali, porta a un miglioramento nel processo di

classificazione rispetto al Random Forest applicato ai dati originali.
Inoltre, l'approccio modulare della programmazione genetica produce spesso miglioramenti rispetto alla GP standard, anche se non è stata sviluppata con tale obiettivo.
L'uso della programmazione genetica modulare potrebbe essere considerato anche in altri campi in cui si affrontano problemi complessi di classificazione. Tuttavia, è fondamentale tener conto delle sfide e delle limitazioni associate a questo approccio. Ad esempio, la scelta dei moduli appropriati e i parametri dell'algoritmo possono influenzare significativamente l'efficacia complessiva dell'approccio modulare. Inoltre, è necessaria un'attenta valutazione delle specifiche del problema e delle caratteristiche dei dati per determinare l'efficacia dell'approccio modulare.
In conclusione, questa ricerca ha contribuito a dimostrare che l'approccio modulare della programmazione genetica può offrire miglioramenti significativi nella classificazione dei segnali rispetto alla GP standard e all'uso diretto del Random Forest sui dati originali.

Sviluppi futuri

La ricerca futura potrebbe esplorare ulteriormente l'ottimizzazione dell'approccio modulare e l'individuazione di nuove strategie che combinano efficacemente i vantaggi della programmazione genetica con altri metodi di classificazione.
In particolare, ci sono alcune modifiche che potrebbero essere effettuate per provare a migliorare l'algoritmo. Per quanto riguarda l'ottimizzazione dell'approccio modulare, è possibile approfondire l'analisi dei programmi generati dai moduli da inserire nel set di primitive verificando se tali programmi sono effettivamente funzioni in cui compaiono le variabili di ingresso o se si tratta di costanti. In questo secondo caso, sarebbe opportuno inserire i moduli nel set di terminali anziché nel set delle primitive.
Un'altra modifica da effettuare potrebbe essere quella di ridurre la comples-

sità degli alberi generati mediante sostituzione mirate dei sottomoduli. In particolare, a partire dalla seconda iterazione, si potrebbe identificare la presenza di sottomoduli negli individui della popolazione iniziale che corrispondono a quelli scelti per essere inseriti nel set delle primitive nelle iterazioni precedenti. Quando tali sottomoduli vengono individuati, potrebbero essere sostituiti con la corrispondente funzione, evitando così l'occupazione inutile della profondità degli alberi. Questa strategia potrebbe contribuire a ridurre la complessità degli alberi generati e a migliorare le prestazioni del sistema complessivo.

Un'altra modifica da considerare potrebbe essere quella di ritornare al primo approccio e migliorare l'efficienza dell'estrazione dei sottomoduli di varie profondità, oppure estrarre solo quelli di profondità 1 e 2 e applicare il classificatore k-means per scegliere quali mantenere nelle iterazioni successive.

In conclusione, nella ricerca futura si dovrà lavorare sul processo di estrazione dei sottomoduli e sperimentare diverse strategie di selezione dei moduli da inserire nel set delle primitive, con l'obiettivo di identificare il metodo che possa garantire i migliori risultati in termini di prestazioni del sistema. Esplorare ulteriormente le opzioni prima descritte potrebbe contribuire a migliorare l'efficacia dell'approccio modulare della programmazione genetica nella pre-elaborazione dei dataset e nell'applicazione dei classificatori.

Bibliografia

[1] A.P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley, 2007.

[2] A novel crossover operator for genetic algorithms: Ring crossover - scientific figure on researchgate.

[3] A novel crossover operator for genetic algorithms: Ring crossover - scientific figure on researchgate.

[4] R. Poli, W.B. Langdon, N.F. McPhee, and J.R. Koza. *A Field Guide to Genetic Programming*. Lulu.com, 2008.

[5] DEAP Development Team. Deap documentation: Distributed evolutionary algorithms in python, 2021.

[6] Python Software Foundation. *Tkinter 8.6 Documentation*. Python Documentation, 3.10 edition, 2021. `https://docs.python.org/3/library/tkinter.html`.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Python Software Foundation. Python 3 documentation: Pickle - python object serialization. 2021.

[9] A. Goldberger, L. Amaral, L. Glass, J. Hausdorff, PC. Ivanov, R. Mark, and HE. Stanley. Physiobank, physiotoolkit e physionet: componenti di una nuova risorsa di ricerca per segnali fisiologici complessi. *Circulation*, 101(23):e215–e220, 2000.

[10] Physikalisch-Technische Bundesanstalt (PTB). Ptb homepage.

[11] Shayan Fazeli. Heartbeat sounds, 2016.

[12] Mohammad Kachuee, Shayan Fazeli, and Majid Sarrafzadeh. Ecg heartbeat classification: A deep transferable representation. *arXiv preprint arXiv:1805.00794*, 2018.

[13] KNIME. How to classify ecg signals with deep learning, s.d.