UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura

Corso di Laurea in Ingegneria Informatica, Elettronica e delle Telecomunicazioni

Programmazione Genetica per l'ottimizzazione della Rappresentazione Latente di pattern

GP-based pattern Embedding optimization

Relatore:

Prof. Stefano Cagnoni

Tesi di Laurea di:

Federico Brandini

ANNO ACCADEMICO 2022-2023

A chi mi ha dato tutto, alla mia bisnonna Dinetta

Indice

Introduzione **1**

1 Calcolo Evolutivo e Programmazione Genetica **3**

1.1 Evolution modelling 4

1.1.1 Popolazione 6

1.1.2 Fitness function 7

1.1.3 Selezione 9

1.1.4 Riproduzione 11

1.1.5 Condizione di Terminazione 14

1.2 Programmazione Genetica 14

1.2.1 Encoding 16

1.2.2 Popolazione iniziale 17

1.2.3 Fitness function 18

1.2.4 Operatori 19

1.2.5 Run 21

2 Descrizione del Problema e Progettazione **22**

2.1 Problema di Classificazione 22

2.2 Classificatore stand-alone 23

2.2.1 Overfitting e Condizione di Terminazione 28

2.2.2 Elitarismo 29

2.3 Embedding e Coevoluzione 30

2.3.1 Valutazione dell'Embedding 32

2.3.2 Elitarismo esteso e Sopravvivenza 33

3 Implementazione **37**

3.1 Strumenti impiegati 37

3.2 Realizzazione 38

3.2.1 Modulo `_cfg_params.py` 39

3.2.2 Singole evoluzioni 42

3.2.3 Coevoluzione 49

3.3 Generalizzazione 52

4 Risultati **53**

4.1 Statistiche di Testing 53

4.2 Risultati raggiunti 54

4.3 Grafici di Learning 56

4.4 Classificatore completo 59

Conclusioni **62**

Bibliografia **63**

Introduzione

Per iniziare, conviene innanzitutto inquadrare l'ambito matematico in cui ci troviamo, ossia il cuore di questa tesi. Un **problema di ottimizzazione** è un problema per il quale siamo interessati a ricercare una soluzione ottimale, relativamente ad un certo obiettivo. Per esempio, supponiamo di dover ricercare il percorso migliore che ci consenta di giungere in B partendo da A; non necessariamente il percorso migliore deve essere quello più breve; per le nostre esigenze, quello migliore potrebbe essere quello che, in media, risulti più pianeggiante, con poco dislivello. La Matematica risolve questo genere di problemi rappresentandone gli obiettivi come funzioni, per le quali si ricercano quei valori delle variabili indipendenti che massimizzano o minimizzano i valori delle medesime (*i.e.* i punti di massimo e minimo globali). Un approccio simile è possibile quando lo spazio di ricerca delle soluzioni risulta essere denso, continuo e nel quale sia possibile applicare l'operatore di derivata (o di gradiente, per funzioni di più variabili); tuttavia, quando ci si muove in spazi di oggetti discreti, discontinui e non numerici, solitamente l'approccio visto in precedenza risulta particolarmente ostico da applicare. Il Calolo Evolutivo (*Evolutionary Computation, EC*), di cui la GP è una specializzazione, può essere uno strumento utile proprio in questi casi. Come da nome, questa tecnica di risoluzione si basa sui concetti teorici inerenti l'evoluzione biologica darwiniana, secondo i quali gli individui di una popolazione, attraverso mutazioni casuali e la proliferazione (in cui vengano favoriti i migliori), costituiranno generazioni di nuovi individui sempre più adatti all'ambiente che li circonda. Dopo un numero consistente di generazioni, probabilisticamente,

questi caratteri si estenderanno nella maggior parte della popolazione. Nel nostro caso, gli individui non saranno organismi viventi, ma possibili soluzioni per il nostro problema di ottimizzazione. Il concetto sarà evolvere la popolazione iniziale di individui per un certo numero di generazioni, attraverso le quali la "capacità di adattamento" (*i.e.* la **fitness**, come vedremo) della popolazione tenderà verso il valore ottimo. In questa tesi, l'obiettivo sarà realizzare un **classificatore generico** di pattern con l'uso della GP, mediante una ricerca nello spazio dei possibili classificatori, e di migliorarne le prestazioni attraverso una **rappresentazione latente** (o **embedding**) dei dati in ingresso al classificatore. Nel Capitolo 1 introdurremo i concetti principali relativi all'EC e agli Algoritmi Genetici (*Genetic Algorithm, GA*), con particolare riferimento alla GP; nel Capitolo 2 verrà presentata l'architettura del sistema, partendo dal modello di classificatore binario senza embedding (*stand-alone*), per poi giungere all'introduzione dell'embedding dei dati di input; nel Capitolo 3 verranno presentati gli strumenti utilizzati per l'implementazione, per poi passare al codice (in linguaggio Python); infine, nel Capitolo 4 analizzeremo i risultati ottenuti con qualche considerazione finale.

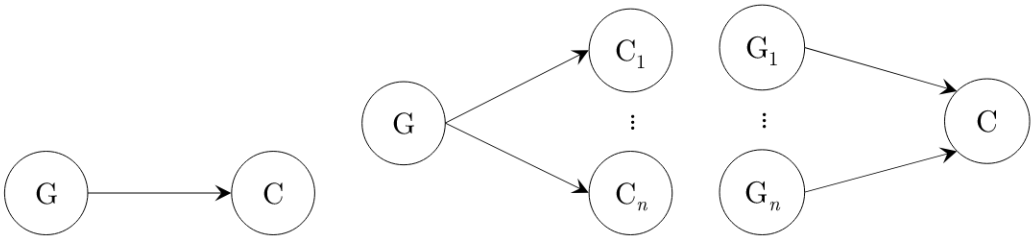
Capitolo 1

Calcolo Evolutivo e Programmazione Genetica

L'**evoluzione** è letteralmente il processo di miglioramento delle capacità di adattamento di un sistema, in relazione all'ambiente in cui risiede. Questo termine rientra in più ambiti distinti, come quello cosmico, chimico, stellare, planetario e biologico; è in particolare su quest'ultimo che si fonda la teoria del Calcolo Evolutivo (*Evolutionary Computation, EC*). Anche se esistono molteplici interpretazioni e teorie legate al concetto di evoluzione biologica, il primo che ne diede una formalizzazione fu Jean-Baptiste Lamarck nel 1809 con la sua teoria ***dell'uso e del disuso***, secondo cui gli individui mantengono e migliorano le caratteristiche che hanno un'effettiva utilità (*uso*), mentre perdono quelle non altrettanto utili (*disuso*). Una seconda teoria altrettanto importante riguarda la **selezione naturale**, affrontata nel 1859 da Charles Darwin nel suo libro *L'origine delle Specie*; secondo questa visione, gli individui con i migliori tratti e le migliori caratteristiche si adattano con maggiore probabilità all'ambiente, trasmettendo di conseguenza tali tratti alla prole. Queste due teorie sono entrambe tutt'ora accettate per descrivere e modellare il processo evolutivo.

1.1 Evolution modelling

Il primo passo per la definizione di un algoritmo evolutivo riguarda la rappresentazione degli individui nello spazio di ricerca, la cosiddetta fase di **encoding**. Se ragionassimo come in natura, un individuo potrebbe essere rappresentato da un suo **cromosoma**, ossia una lunga stringa di informazioni codificate in **geni**. Una qualsiasi forma in cui si può presentare un gene è detta **allele**. È importante distinguere le caratteristiche codificate all'interno del cromosoma (*i.e.* il **genotipo**) dalle caratteristiche somatiche e comportamentali che l'individuo mostra all'esterno (*i.e.* il **fenotipo**). Intuitivamente, ci si attende che un gene influenzi un unico carattere del fenotipo e che un carattere del fenotipo sia influenzato da un unico gene del genotipo; tuttavia, questa situazione accade raramente. Solitamente, infatti, ci si ritrova con un gene che influenza più caratteri (*pleiotropia*) e con un carattere influenzato da più geni (*poligenia*). Per esempio, non esiste il "gene del colore degli occhi", ma esistono più geni che influenzano parzialmente tale carattere. In realtà, i geni non sono i soli ad influenzare il fenotipo di un individuo, ma anche lo stesso ambiente ne è coinvolto (*i.e.* le relazioni/interazioni fra l'individuo e l'ambiente). Nel nostro caso, un cromosoma sarà un insieme di variabili



(a) Caso raro (b) Pleiotropia (c) Poligenia

Figura 1.1: Influenza del genotipo sul fenotipo

che identificherà una possibile soluzione ottima, un gene sarà una fra queste variabili e l'allele un valore da assegnarle. Negli Algoritmi Genetici (*Genetic Algorithm, GA*), i cromosomi vengono rappresentati da **vettori** in $\{0,1\}^n$, mentre nella Programmazione Genetica (*Genetic Programming, GP*) vengo-

no impiegati i cosiddetti **alberi sintattici** (dei quali discuteremo più avanti). Come nel caso del DNA, nella GA gli individui vengono rappresentati attraverso una loro codifica (il genotipo), che traduce le loro caratteristiche (il fenotipo). Risulta quindi necessario definire una corrispondenza tra individuo e fenotipo, e un'altra tra fenotipo e genotipo. In una rappresentazione

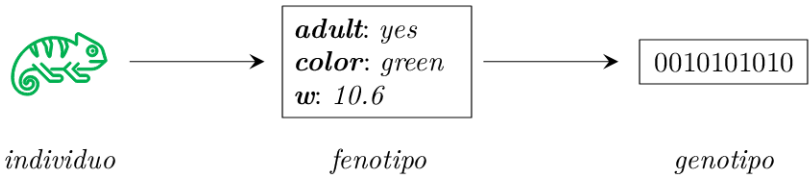


Figura 1.2: Encoding di un individuo reale in un cromosoma

genotipica possiamo trovare tre diversi tipi di variabili, per ognuno dei quali è necessaria un'operazione di trasformazione in una stringa di bit, che andrà a comporre parzialmente il cromosoma dell'individuo. In caso di variabile

- **binaria**, verrà creata una corrispondenza tra i due valori possibili della variabile con i due valori booleani 0 e 1, trasformandola in un bit (*e.g.* nell'esempio la variabile *adult* corrisponde al primo bit del cromosoma, quello più a sinistra);
- **nominale**, verrà creata una corrispondenza fra i k valori possibili per quella variabile e una codifica binaria con un numero di bit pari a $\lceil \log_2 k \rceil$ (*e.g.* nell'esempio la variabile *color*, suppondendo 8 possibili colori, corrisponde ai 3 bit successivi al primo);
- **densa**, è necessaria una funzione $\phi : \Delta \to \{0,1\}^d$ che permetta di approssimare il valore con un numero di bit fissato d; solitamente si lavora con la *distanza di Hamming* (*e.g.* nell'esempio la variabile w corrisponde agli ultimi 6 bit del vettore).

Delineato l'encoding degli individui, possiamo introdurre l'algoritmo di evoluzione per la risoluzione del nostro problema di ottimizzazione. Si parte da

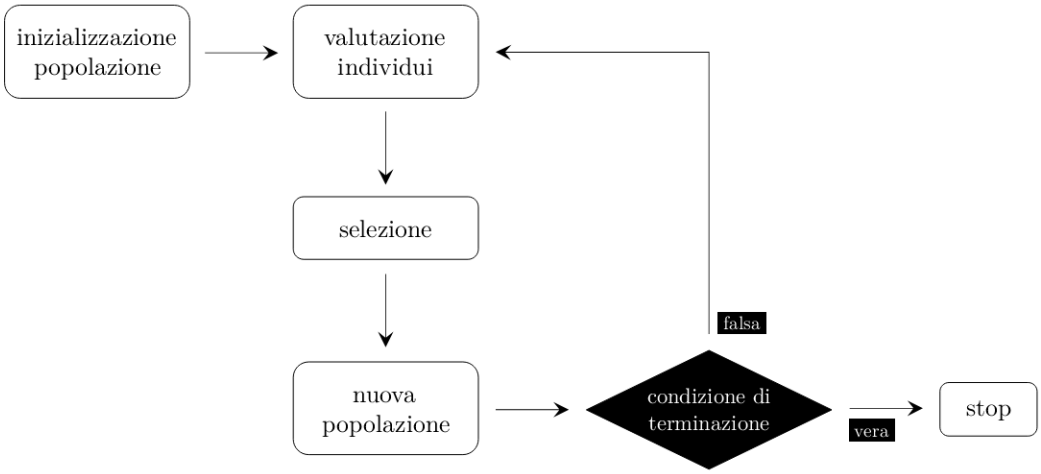


Figura 1.3: Algoritmo generico di evoluzione

una popolazione iniziale di individui da valutare, si selezionano gli individui "migliori", attraverso ulteriori operazioni si genera una nuova popolazione da rivalutare e così via, fino al verificarsi di una condizione di terminazione. Analizziamo nel dettaglio ognuno dei passi dell'algoritmo generico di evoluzione.

1.1.1 Popolazione

Inizialmente, potremmo definire il concetto di popolazione semplicemente come "un *insieme* di individui o cromosomi". In questo caso, però, non è possibile sapere se un individuo compare più volte all'interno della medesima popolazione; per questo motivo è necessario estendere il concetto di *insieme* con quello di *multiinsieme*.

Definizione 1.1. *Un* ***multiinsieme*** *è una coppia* (A, m)*, dove* A *è un insieme e* $m : A \to \mathbb{N} \setminus \{0\}$ *una funzione che associa ad ogni elemento di* A *la molteplicità di tale elemento.*

Se $P = (A, m)$ è un multiinsieme, chiamiamo

- **size** il numero di elementi in P contati con la loro molteplicità, ossia

$$\text{size}(P) = \sum_{a \in A} m(a); \tag{1.1}$$

- **diversity** il numero di elementi in P, senza molteplicità, ossia

$$\text{diversity}(P) = \mid A \mid . \tag{1.2}$$

Osserviamo che un insieme è un caso particolare di multiinsieme, nel quale $m \equiv 1$ su A e i concetti di *size* e *diversity* sono equivalenti alla comune *cardinalità*. Quindi, una **popolazione** è un *multiinsieme* di individui o cromosomi.

Per cominciare, è necessario inizializzare una popolazione iniziale di cromosomi; solitamente, la si inizializza in modo casuale e uniforme. L'obiettivo è rendere la distribuzione della popolazione il più possibile uniforme (spazialmente), così da popolare tutte le regioni dello spazio cromosomico. Una size alta genera maggiore diversità all'interno della popolazione, tuttavia aumenta la complessità temporale dell'algoritmo evolutivo; fissarla troppo bassa significa sicuramente abbassare il livello di complessità, ma anche non rappresentare al meglio lo spazio di ricerca. In quest'ultimo caso, l'algoritmo evolutivo potrebbe portarsi ad una condizione di *convergenza prematura*, ossia indicare come soluzione migliore al problema un individuo ben lontano dall'effettiva soluzione ottima.

1.1.2 Fitness function

Esattamente come in natura un individuo sopravvive se possiede una buona capacità di adattamento al suo ambiente, le nostre possibili soluzioni devono poter "sopravvivere" solo se effettivamente costituiscono una buona soluzione per il nostro problema di ottimizzazione. Risulta quindi necessario definire una funzione f che valuti tale bontà: la **fitness function**. Indicando con

X_{gen} lo spazio di tutti i possibili cromosomi, definire una funzione di valutazione direttamente su X_{gen} potrebbe risultare ostico in alcuni casi. In primo luogo, riportiamo la rappresentazione genotipica in una forma più comprensibile all'esterno, cioè quella fenotipica; per far ciò, definiamo la funzione di **decodifica**

$$\Phi : X_{gen} \to X_{fen}, \tag{1.3}$$

che associ ad ogni cromosoma, il corrispondente fenotipo, appartenente allo spazio di tutti i possibili fenotipi X_{fen}. Una volta passati alla rappresentazione decodificata, il passo successivo è definire una funzione di valutazione (**obiettivo**)

$$\Psi : X_{fen} \to \mathbb{R}, \tag{1.4}$$

che associa ad ogni fenotipo la corrispondente valutazione reale. Infine, per rappresentare la valutazione su una scala più interpretabile, scegliamo di comprimere la valutazione sui numeri reali positivi, definendo la funzione di **scaling**

$$\Upsilon : \mathbb{R} \to \mathbb{R}^+. \tag{1.5}$$

Mettendo insieme i pezzi, la funzione di fitness è data dalla composizione delle tre funzioni definite, ossia

$$f : X_{gen} \to \mathbb{R}^+, f = \Upsilon \circ \Psi \circ \Phi. \tag{1.6}$$

La versione di fitness appena definita è quella più completa; nulla vieta di eliminare alcuni passaggi nella composizione, se questi non dovessero servire (*e.g.* se la rappresentazione utilizzata è già comprensibile (possiamo parlare già di fenotipo), allora basta $f : X_{fen} \to \mathbb{R}^+, f = \Upsilon \circ \Psi$). In generale, se la fitness di un individuo dipende esclusivamente dall'individuo stesso, allora f è detta **assoluta**, altrimenti **relativa**. Inoltre, possiamo classificare i diversi tipi di problema di ottimizzazione a seconda del tipo di fitness adottata. Un problema è

- **unconstrained** (o **non vincolato**) se f descrive esclusivamente l'obiettivo;

- **constrained** (o **vincolato**) se f contiene una penalità per diversificare le soluzioni con pari valore dell'obiettivo (*e.g.* se l'obiettivo è determinare il percorso più breve da A a B, se se ne dovessero trovare due di pari lunghezza, allora potremmo scegliere quello con minor dislivello, penalizzando la fitness dell'altro);

- **multi-objective** se f si può scrivere come una somma pesata di n sotto-obiettivi, cioè $f = \sum_{i=1}^{n} w_i f_i$, con $w_i \in \mathbb{R}^+, \forall i = 1, ..., n$;

- **dynamic/noisy** se f dipende anche dal tempo e contiene una componente di rumore gaussiano.

Finora abbiamo definito la fitness function come una funzione nei reali positivi; questo perché f è pensata (non obbligatoriamente) come una funzione da minimizzare. In maniera del tutto equivalente si potrebbe pensare di definire una nuova fitness function $\widetilde{f}$ da massimizzare. Uno dei modi è banalmente porre $\widetilde{f} = -f$.

1.1.3 Selezione

La selezione è la fase di scelta degli individui migliori della popolazione e dovrebbe replicare la selezione naturale darwiniana. Più in generale, la **selezione** è un processo che restituisce un *sottomultiinsieme* S della popolazione P.

Definizione 1.2. *Siano $P = (A_P, m_P)$ e $S = (A_S, m_S)$ due multiinsiemi. S è un **sottomultiinsieme** di P se $A_S \subseteq A_P$ e $m_S(a) \leq m_P(a), \forall a \in A_S$. In tal caso, scriviamo $S \subseteq P$.*

Nella pratica, questa operazione è svolta da un operatore di selezione che si occupa di scegliere gli individui in base a un'opportuna strategia. I diversi operatori di selezione vengono "valutati" da una misura, detta *selective pressure* (pressione selettiva) o *takeover time*. Viene definita come la velocità con la quale la presunta migliore soluzione occuperà l'intera popolazione

mediante l'applicazione ripetuta del solo operatore di selezione. Avere una maggiore pressione selettiva, significa che la popolazione diminuirà la propria diversità più rapidamente, limitandone la capacità esplorativa, con il rischio di un imminente convergenza prematura. Un valore basso di pressione selettiva indica che l'applicazione dell'operatore favorirà una diminuzione della diversità più graduale. Vediamo alcuni operatori di selezione (supponiamo che f sia una fitness function da massimizzare).

- **Random selection.** La probabilità di selezione è uniforme; cioè se $p_{x_i} = \mathbb{P}(x_i \text{ viene selezionato})$, per ogni molteplice individuo x_i di P, in questo caso
$$p_{x_i} = \frac{1}{\text{size}(P)}. \tag{1.7}$$
Questo significa che tutti gli individui hanno la stessa probabilità di essere selezionati/scartati. In effetti, è anche la strategia selettiva con la più bassa *selective pressure*.

- **Fitness-proportional selection.** Ogni individuo ha una probabilità di selezione proporzionale al proprio valore di fitness; ossia
$$p_{x_i} = \frac{f(x_i)}{\sum_{j=1}^{\text{size(P)}} f(x_j)} = \frac{f(x_i)}{\sum_{y \in A_P} (m \cdot f)(y)}. \tag{1.8}$$
Una canonica implementazione dell'operatore è chiamata *Roulette Wheel Selection*, nella quale ad ogni individuo viene associata una fetta di roulette di grandezza proporzionale al valore di fitness e, a seconda di dove si fermerà la pallina, verrà selezionato l'individuo corrispondente. Proprio per il suo forte legame con la fitness, tale operatore risulta avere un'alta pressione selettiva.

- **Tournament selection.** Fissato $n < \text{size}(P)$, si scelgono n molteplici individui di P e, fra di essi, si seleziona quello con il valore di fitness più alto. La pressione selettiva dell'operatore è proporzionale a n: i casi limite sono $n = 1$, in cui la strategia degenera in una *random selection*, e $n = \text{size}(P) - 1$, dove la pressione selettiva può superare quella della *fitness-proportional selection*.

- **Rank-based selection.** Gli individui x_i vengono ordinati per valori di fitness debolmente decrescenti, ossia vale $\bar{\imath} < \bar{\jmath} \implies f(x_{\bar{\imath}}) \geq f(x_{\bar{\jmath}})$. A questo punto, ad ogni individuo x_i viene assegnata una probabilità di selezione p_{x_i}, che dipende dalla sola posizione i dell'individuo nell'ordinamento.

1.1.4 Riproduzione

Una volta selezionati gli individui della popolazione secondo una specifica strategia, bisogna creare la nuova popolazione di individui che andranno a costituire la *generazione* successiva. La **riproduzione** è letteralmente il processo di creazione di prole a partire da genitori selezionati (fase precedente). A tal proposito, vengono impiegate due strategie: ricombinazione e mutazione.

Il **crossover** (o *ricombinazione*) è un operatore di riproduzione che consente di generare prole (*i.e.* nuovi individui) dalla ricombinazione di materiale genetico ottenuto da altri individui. Siccome gli individui selezionati sono solitamente quelli migliori, la ricombinazione di questi ultimi creerà nuovi individui che costituiranno una nuova popolazione con minore diversità (*diversity*); infatti, è necessario un secondo operatore per controbilanciare, in qualche modo, questo effetto: la mutazione (che vedremo in seguito). Possiamo classificare gli operatori di crossover sulla base del valore di *arity* (*i.e.* numero di operandi), ossia il numero di "genitori" coinvolti nella ricombinazione; un operatore può essere

- **asessuale** (*asexual*), se da un solo genitore viene generato un solo figlio;
- **sessuale** (*sexual*), se da due genitori vengono generati uno o due figli;
- **multi-recombination**, se da tre o più genitori vengono generati uno o più figli.

Una volta avvenuta la fase precedente di selezione, non è detto che tutti gli individui vengano coinvolti nella ricombinazione; anzi, questa casistica

è piuttosto rara. Per decidere quali individui coinvolgere, viene fissata una probabilità di crossover p_c (solitamenta alta) così definita:

$$p_c = \mathbb{P}(x_i \text{ è coinvolto in una ricombinazione}), \forall i = 1, ..., \text{size}(P). \quad (1.9)$$

Per le rappresentazioni *bitstring* (*i.e.* vettori in $\{0,1\}^n$), gli operatori di crossover sono solitamente tutti sessuali, generano esattamente due figli e vengono identificati da un vettore binario $m \in \{0,1\}^n$ della stessa dimensione n degli individui: la **maschera** (o **mask**). Questo vettore indica quali bit dei genitori vengono scambiati nella generazione dei due figli. Se x^1 e x^2 sono i genitori, i figli $\widetilde{x}^1$ e $\widetilde{x}^2$ si ottengono da

$$\widetilde{x}_k^1 = \begin{cases} x_k^1 & \text{se} \quad m_k = 0 \\ x_k^2 & \text{se} \quad m_k = 1 \end{cases} \text{ e } \widetilde{x}_k^2 = \begin{cases} x_k^2 & \text{se} \quad m_k = 0 \\ x_k^1 & \text{se} \quad m_k = 1 \end{cases} ,\forall k = 1, ..., n \quad (1.10)$$

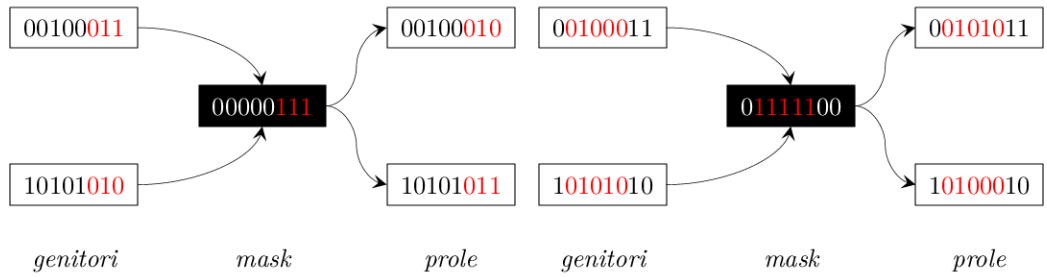
A seconda del tipo di maschera, si distinguono diverse strategie di crossover; vediamo quelle principali.

- **One-point crossover**, dove m è costituita da una prima serie di bit a 0, per finire con un'altra serie a 1 (come la maschera di rete) (*e.g.* con $n = 8$, 00000111); in questo caso è come se i genitori, nella creazione della prole, si scambiassero l'ultima parte del cromosoma (il bit da cui inizia la serie di 1 è scelto uniformemente campionando $\mathcal{U}(1, n-1)$).

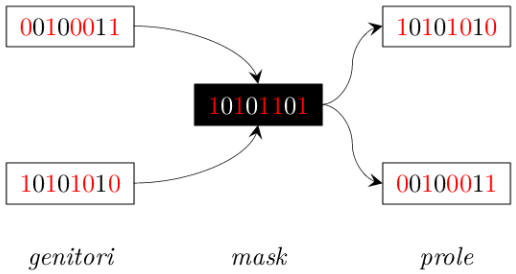
- **Two-point crossover**, dove m è costituita da una sola parte centrale contigua di bit a 1 (*e.g.* con $n = 8$, 01111100); è come se i genitori si scambiassero un segmento centrale di cromosoma (i bit di inizio e fine del segmento centrale vengono scelti campionando $\mathcal{U}(1, n)$).

- **Uniform crossover**, dove m è costituita da bit a 0 e bit a 1 con probabilità uniforme (*e.g.* con $n = 8$, 10101101); è come se i genitori si scambiassero diversi segmenti di cromosoma.

Un'altro operatore utilizzato nella fase di riproduzione è quello di mutazione. La **mutazione** è il processo di cambiamento dell'allele di un gene in un cromosoma. L'obiettivo principale di tale operatore, come avviene anche in



(a) One-point (b) Two-point



(c) Uniform

Figura 1.4: Crossover per rappresentazioni bitstring (caso $n = 8$)

natura, è aumentare la diversità nella popolazione; tuttavia, va applicato con parsimonia, in quanto si possono ledere le caratteristiche degli individui/soluzioni migliori. Per questa ragione la probabilità p_m di mutazione è scelta solitamente bassa.

Le prestazioni dell'algoritmo sono ovviamente influenzate dalla scelta di p_c e p_m. Solitamente, tale scelta avviene staticamente, ossia si stabiliscono i valori delle probabilità prima di procedere con l'esecuzione dell'algoritmo; tuttavia, è stato dimostrato che la variazione dei tassi di mutazione e ricombinazione aumenta le prestazioni per la ricerca degli individui migliori. Limitatamente alla mutazione, in alcuni casi, può essere utile avere un alto valore di p_m nelle prime generazioni, per aumentare la capacità esplorativa nella popolazione, e che decresce progressivamente all'aumentare del numero della generazione $t \geq 0$ (*i.e.* $p_m(t) \geq p_m(t+1)$). Un'altra strategia prevede di assegnare un va-

lore di p_m tanto più alto, quanto più è basso il valore di fitness dell'individuo (*i.e.* $p_m(x_i) \propto \frac{1}{f(x_i)}$); tale scelta dovrebbe consentire agli individui migliori di mantenere le proprie caratteristiche, evitando di peggiorare la fitness a causa di una mutazione, e agli individui peggiori di migliorare.

1.1.5 Condizione di Terminazione

Gli operatori evolutivi (*i.e.* selezione, mutazione e crossover) vengono applicati iterativamente finché non si verifica una **condizione di terminazione**. Per decidere quando far terminare l'algoritmo evolutivo, si può procedere per **limitazione** del numero di generazioni o del numero di volte in cui viene calcolata la fitness; tuttavia, potrebbe accadere che l'algoritmo termini ancor prima di aver trovato una soluzione che perlomeno si avvicini a quella ottima. Un'altra modalità per la scelta della terminazione riguarda i cosiddetti **criteri di convergenza**, ossia sfruttare una qualche "sentinella" che consenta di affermare che la popolazione si trovi in una situazione di *ristagno* (*i.e.* arresto del miglioramento). In base a questi criteri, si decide di terminare quando

- non si osservano miglioramenti nell'individuo migliore per un certo numero di generazioni;
- non si osservano miglioramenti significativi nell'intera popolazione;
- è stata trovata una soluzione accettabile (con un valore di fitness sopra o sotto una certa soglia);
- non si osservano miglioramenti significativi sul miglior valore di f finora ottenuto (nelle varie generazioni).

1.2 Programmazione Genetica

Finora abbiamo trattato la risoluzione di problemi di ottimizzazione mediante algoritmi genetici (GA), dove gli individui vengono rappresentati come vetto-

ri. Questo tipo di encoding può essere limitante in alcuni casi, specialmente quando si lavora con una popolazione nella quale gli individui rappresentano funzioni o programmi. Nella Programmazione Genetica (*Genetic Programming, GP*) gli individui sono proprio delle funzioni e vengono rappresentati attraverso **alberi sintattici** (*syntax trees*). I nodi foglia costituiscono gli **operandi** (chiamati anche **nodi terminali**), gli altri nodi sono gli **operatori** (*e.g.* nell'esempio, i nodi x, 2 e 5 sono nodi terminali, mentre $+$ e $*$ sono operatori). La struttura gerarchica dell'albero delinea le precedenze fra gli

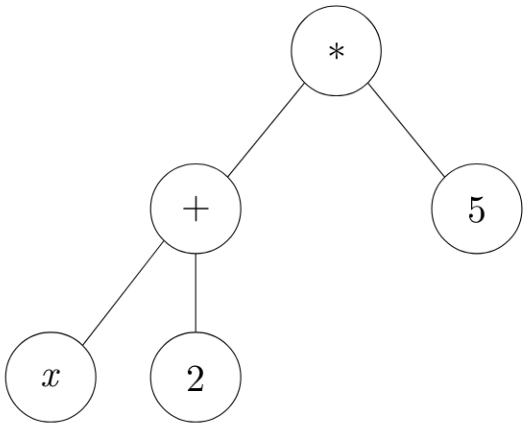


Figura 1.5: Un albero sintattico che rappresenta l'operazione $(x + 2) * 5$

operatori da applicare e, in base all'algoritmo di visita utilizzato, otteniamo espressioni regolari equivalenti, ma espresse secondo differenti notazioni. Se applichiamo un algoritmo di visita

- *pre-order*, allora otterremo un'espressione in notazione *prefissa*, dove l'operatore compare prima degli operandi (*e.g.* nell'esempio, $*$ $(+$ x $2)$ $5)$;
- *in-order*, allora otterremo un'espressione in notazione *infissa* (quella usata in matematica), dove l'operatore compare fra gli operandi (*e.g.* nell'esempio, $(x + 2) * 5$);
- *post-order*, allora otterremo un'espressione in notazione *postfissa*, dove l'operatore compare dopo gli operandi (*e.g.* nell'esempio, $(x\,2\,+)\ 5\ *$).

1.2.1 Encoding

Per generare un albero sintattico è necessario conoscere tutti i nodi che possono essere utilizzati e, fra questi, quali sono gli operandi e quali gli operatori.

Inizialmente definiamo un insieme di simboli terminali T, che chiameremo **terminal set**. In questo insieme verranno inclusi

- gli input esterni (o variabili) (*e.g.* x, y);
- le funzioni senza argomenti (le cosiddette *0-arity function*) (*e.g.* rand());
- le costanti (*e.g.* 5, -2);
- le costanti effimere (*Ephemeral Random Constants, ERC*), che assumono un valore casuale nel momento della generazione dell'albero e che viene trattato in seguito come costante (indicate con il simbolo $\Re$).

Un esempio di terminal set potrebbe essere $T = \{x, y, \text{rand}(), 5, -2, \Re\}$.

Definiamo, inoltre, un insieme di operatori F da applicare agli operandi, che chiameremo **function set**. Le funzioni cambiano a seconda della tipologia del problema da risolvere; tuttavia, per evitare situazioni in cui l'espressione regolare decodificata dall'albero risulti semanticamente errata (*e.g* l'espressione regolare infissa $\log(5-9)$, pur essendo sintatticamente corretta, è errata dal punto di vista semantico, in quanto la funzione logaritmica è definita solo in $\mathbb{R}^+$), spesso viene imposto il vincolo della **closure** (*chiusura*), che comprende le seguenti condizioni.

- *Type consistency.* Significa richiedere che una qualsiasi ricombinazione (crossover) produrrà un albero semanticamente corretto. A tal proposito, ogni funzione $f \in F$ deve essere *type-consistent*, ossia restituire valori dello stesso tipo degli argomenti. Solitamente viene previsto un meccanismo di conversione automatica fra tipi differenti (*e.g.* corrispondenza fra i valori booleani V/F e i valori numerici 1/0).

- *Evaluation safety.* Significa richiedere che i domini delle funzioni vengano estesi in modo tale da evitare situazioni di non definizione delle stesse (funzioni *protected*). Per esempio, la funzione di divisione $/(a, b)$ non è definita per $b = 0$; tuttavia, nessuno ci assicura che quell'input non possa comunque essere 0. Risulta quindi necessario "proteggere" $/(\cdot, \cdot)$ estendendola, ad esempio, in questo modo:

$$/_p(a, b) = \begin{cases} /(a, b) & \text{se} \quad b \neq 0 \\ 0 & \text{se} \quad b = 0 \end{cases} . \tag{1.11}$$

  L'operatore $/_p$ viene solitamente chiamato *divisione protetta*.

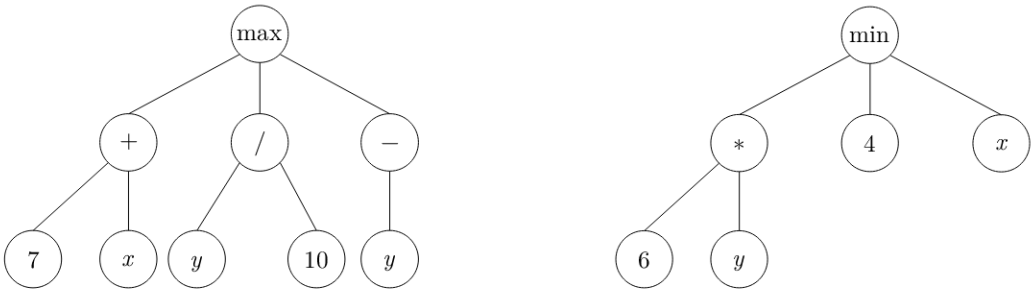
Un'altra importante e richiesta proprietà dell'intero **primitive set** (*i.e.* $T \cup F$) è la **sufficiency**, ossia la possibilità di esprimere una soluzione al problema mediante i nodi definiti. Per esempio, se l'obiettivo è trovare un individuo che calcoli la funzione $g(x) = e^{-x}$, il primitive set $p_1 = \{+, -, *, /, x\}$ non è sufficiente, mentre $p_2 = \{\exp(\cdot), /, x, 1\}$ lo è.

1.2.2 Popolazione iniziale

Per inizializzare il multiinsieme P della popolazione iniziale esistono due metodi, che prevedono entrambi che si fissi il valore massimo di profondità (*depth*) dell'albero d_{max}. (Indichiamo con d la profondità dell'albero corrente)

- **Full method.** Finché $d < d_{max}$, viene aggiunto opportunamente un nodo di F all'albero, altrimenti si aggiunge un nodo presente in T. Questo metodo permette di creare sempre alberi **bilanciati**, ossia alberi in cui i cammini che congiungono la radice ai nodi foglia hanno tutti la stessa lunghezza (in questo caso esattamente pari a d_{max}).

- **Grow method.** Finché $d < d_{max}$, viene aggiunto opportunamente un nodo del primitive set $F \cup T$ all'albero, altrimenti si aggiunge un nodo presente esclusivamente in T. Viceversa, questo metodo permette di generare anche alberi non bilanciati.



(a) Caso bilanciato (b) Caso non bilanciato

Figura 1.6: Alberi sintattici ($d_{max} = 2$)

In entrambi i metodi, la scelta dei nodi all'interno dei set è aleatoria e uniforme; questo significa che tutti i nodi dell'insieme prescelto hanno la stessa probabilità di essere estratti. Per modificare questo comportamento, basta estendere gli insiemi T e F a multiinsiemi, per poi aumentare la molteplicità dei nodi per i quali si vuole aumentare la probabilità di estrazione.

Solitamente, nel contesto pratico, la popolazione viene generata scegliendo di volta in volta, per ogni albero sintattico da generare, uno fra i due metodi appena descritti, con pari probabilità. Questa tecnica di inizializzazione è nota come **Ramped half-and-half initialization**.

1.2.3 Fitness function

Esattamente come nella GA, anche nella GP è necessario definire una funzione di fitness $f : X_{gen} \to \mathbb{R}^+$ che valuti la bontà di una possibile soluzione ottima al problema (un albero sintattico). Inizialmente, l'albero $x_i \in X_{gen}$ viene decodificato nella sua espressione regolare sintatticamente corretta $\Phi(x_i) \in X_{fen}$, ossia il programma. Quest'ultimo viene eseguito più volte, per calcolarne le prestazioni sulla base di un qualche obiettivo, ottenendo una valutazione riassuntiva $\Psi(\Phi(x_i)) \in \mathbb{R}$, per la quale, infine, si esegue l'eventuale scaling $\Upsilon(\Psi(\Phi(x_i))) \in \mathbb{R}^+$. Esistono diversi modi per valutare le prestazioni della funzione eseguita, quali

- tempo di esecuzione per arrivare al *target state* (*i.e.* il risultato dell'espressione);
- il livello di *accuracy* (*i.e.* riconoscere bene un certo pattern);
- la distanza fra il risultato desiderato e il target dell'individuo.

Se i programmi hanno una o più variabili di input, come solitamente accade, può risultare complicato valutarli. La soluzione si identifica nell'eseguire più volte la funzione, con differenti valori delle variabili; ogni configurazione con cui si avvia un programma è detta *fitness case*.

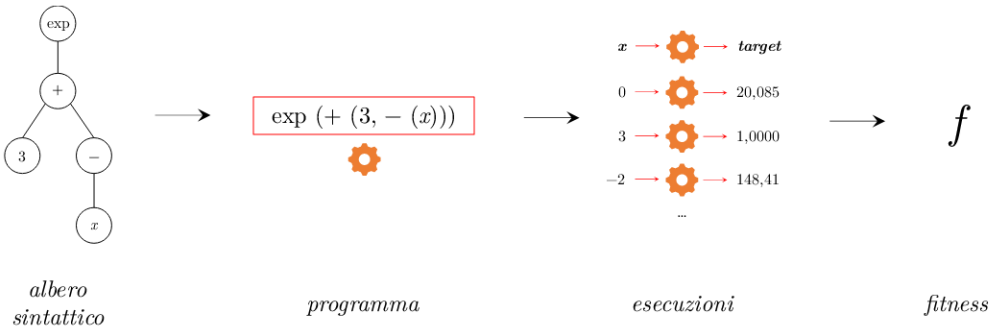


Figura 1.7: Processo di calcolo del valore di fitness per un individuo

1.2.4 Operatori

Gli operatori utilizzati in GP sono analoghi a quelli della GA. Per quanto riguarda la **selezione**, in linea di principio, qualsiasi strategia può essere utilizzata, anche se, solitamente, viene impiegata la *fitness-proportional selection*, la *tournament selection* o la *over selection*. In quest'ultimo caso, gli individui della popolazione corrente vengono ordinati per valori di fitness debolmente decrescenti e, in seguito, divisi in due parti (non necessariamente uguali); la nuova popolazione sarà costituita per l'80% da individui della prima parte e il restante 20% da individui della seconda.

Come tecnica di **ricombinazione**, viene scelto il *subtree crossover*, ossia un operatore di sexual crossover che, partendo da due alberi sintattici, ne genera altri due scambiando un sottoalbero fra i genitori prescelti.

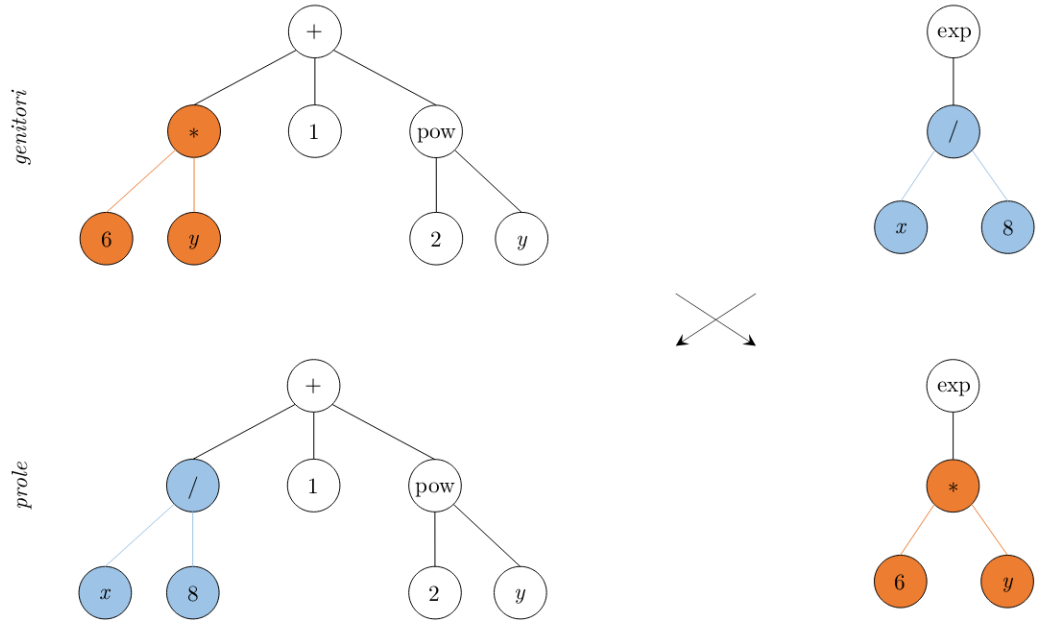


Figura 1.8: Subtree crossover

Come operatore di **mutazione**, ne viene impiegato uno fra

- *subtree mutation*, ossia la sostituzione di un sottoalbero con un altro generato casualmente;
- *headless chicken*, ossia un *subtree crossover* fra l'albero mutante e un altro albero generato casualmente;
- *point mutation*, ossia la sostituzione di un nodo con uno del primitive set (nel caso si tratti di un nodo di F, l'arietà di quest'ultimo dovrà coincidere con quella del nodo rimosso).

In generale, gli operatori di mutazione e ricombinazione possono generare alberi di profondità maggiore rispetto a quelli di partenza. Questo fenomeno

è chiamato **bloat** e, a lungo andare, può portare ad avere una popolazione costituita da alberi sintattici sempre più profondi, causando rallentamenti sia in fase di decodifica che in esecuzione. Per ovviare a questo problema, solitamente si introduce una componente di penalità alla fitness (constrained problem) oppure, più semplicemente, si evita di salvare le modifiche effettuate sull'individuo se, ad esempio, la sua profondità risulta maggiore di d_{max}. Se un individuo riesce a passare alla generazione successiva rimanendo inalterato, si parla di semplice **sopravvivenza** (*survival*); inoltre, se si impone la sopravvivenza degli individui migliori, allora si parla di **elitarismo** (*elitism*).

1.2.5 Run

L'ultimo passo riguarda l'impostazione dei parametri e l'azionamento dell'algoritmo di GP. I principali parametri da impostare sono

- il valore di probabilità di crossover p_c;
- il valore di probabilità di mutazione p_m;
- l'eventuale numero massimo di generazioni g_{max};
- il numero di run dell'algoritmo r.

Fatto ciò, l'algoritmo può essere avviato. Al termine di questo, avviene la designazione della soluzione migliore (*solution designation*), per la quale viene restituito l'individuo migliore (*best-so-far individual*) o gli individui migliori nelle varie generazioni, memorizzati all'interno di una lista chiamata **hall of fame** (in quest'ultimo caso è necessario indicarne la dimensione).

Capitolo 2

Descrizione del Problema e Progettazione

In questo capitolo presentiamo gli obiettivi della tesi, partendo dal modello del classificatore stand-alone, per giungere all'introduzione di un embedding dei dati in ingresso per migliorarne le prestazioni.

2.1 Problema di Classificazione

Un **problema di classificazione** è un problema di ottimizzazione, per il quale si ricerca la migliore funzione $c : \Gamma \to \{0, ..., k-1\}$, con $k \in \mathbb{N} \setminus \{0, 1\}$, che consenta di associare ad ogni oggetto $w \in \Gamma$ la propria classe $c(w)$, fra le k classi possibili. La funzione c è detta **classificatore**. Solitamente, come nel nostro caso, gli oggetti da classificare vengono rappresentati come vettori (**patterns**) di numeri reali; per questa ragione, poniamo $\Gamma = \mathbb{R}^n$. In questa tesi, la ricerca del classificatore migliore avviene attraverso la GP, evolvendo una popolazione di candidati classificatori; tuttavia, anziché ricercare direttamente $c(w)$ per k classi (operazione piuttosto onerosa), l'idea è quella di evolvere separatamente k popolazioni di classificatori, dove ogni popolazione $i \in \{0, ..., k-1\}$ è composta da candidati classificatori binari (2 sole classi), per determinare se un pattern appartiene, o meno, alla classe i. Tale approc-

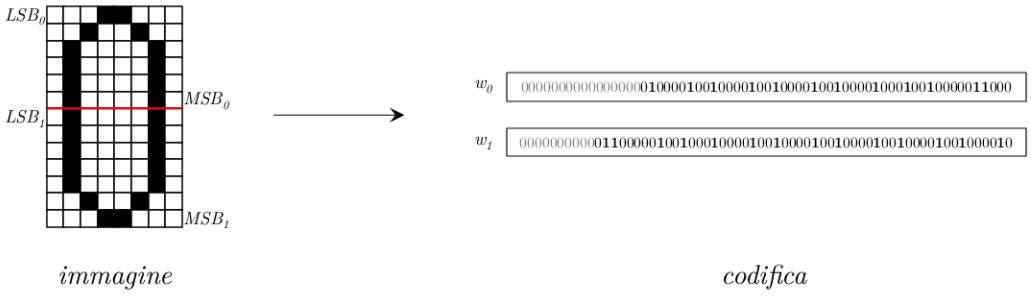
cio prende il nome di *one-vs-all*.

Nel nostro caso, i pattern da classificare sono dei caratteri (cifre, in particolare), ognuno dei quali è rappresentato all'interno di un immagine binaria di 8×13 pixel. Il passo successivo riguarda la **codifica** (o rappresentazione) dell'immagine in un vettore numerico; inizialmente, potremmo codificare la sequenza dei 104 pixel in una word da 104 bit. A questo punto, procediamo con due diverse segmentazioni della parola (*i.e. packed encoding*). In un primo caso, scegliamo di dividere la parola in due segmenti, rispettivamente di 48 e 56 bit, giacenti sui bit meno significativi di due word a 64 bit (patterns in un sottoinsieme di $\mathbb{N}^2$); in un secondo caso, scegliamo una suddivisione in quattro segmenti, i primi tre di 24 bit e il restante di 32, giacenti sui bit meno significativi di quattro word a 32 bit (patterns in un sottoinsieme di $\mathbb{N}^4$).

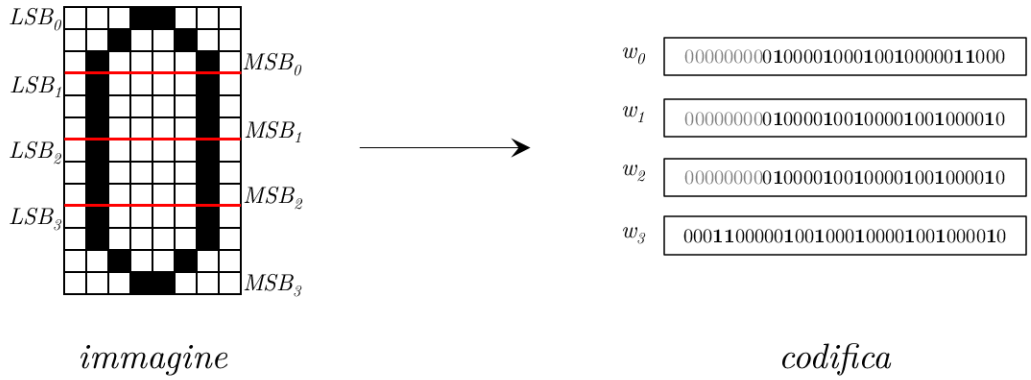
2.2 Classificatore stand-alone

Una volta delinata la strategia di codifica e, quindi, di rappresentazione dei pattern da classificare, l'obiettivo si identifica nel ricercare 10 classificatori (in 10 popolazioni differenti), uno per ogni cifra, che stabiliscano se la cifra rappresentata in un pattern sia, o meno, quella che tale classificatore deve individuare. Formalmente, se $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ è una cifra, l'intenzione (ideale) è quella di ricercare dei classificatori binari $c_i : \mathbb{N}^n \to \{\mathrm{V}, \mathrm{F}\}$ che, dato in input un pattern x, restituiscano il valore di verità $c_i(w)$ dell'enunciato "w rappresenta la cifra i". Utilizzando la prima tecnica di *packed encoding* avremmo $n = 2$ e $w = (w_0, w_1)$, mentre con la seconda $n = 4$ e $w = (w_0, w_1, w_2, w_3)$.

Partiamo definendo un opportuno **primitive set** per l'encoding degli alberi sintattici, che rappresentaranno i candidati classificatori nella popolazione. Iniziamo dal function set. La famiglia logica completa AND-OR-NOT valida



(a) Patterns con due word a 64 bit



(b) Patterns con quattro word a 32 bit

Figura 2.1: Codifica delle immagini in patterns

Funzione	Arietà	Note
AND	2	bitwise AND
OR	2	bitwise OR
NOT	1	bitwise NOT
NAND	2	bitwise NAND
NOR	2	bitwise NOR
XOR	2	bitwise XOR
dCSHt	1	$d \in \{L, R\}$, $t \in \{1, 2, 4\}$

Tabella 2.1: Function set

la *sufficiency* del primitive set; questo significa che qualunque sia la soluzione migliore al nostro problema, questa potrà essere sicuramente rappresentata. Tuttavia, se il function set fosse composto esclusivamente dai primi sei operatori bitwise, un bit di posizione p in un risultato, con $0 \leq p \leq l-1$ (con l numero di bit che compongono una word), dipenderebbe soltanto dai bit di posizione p presenti negli operandi, correndo il rischio di non poter raggiungere una buona soluzione. A tal proposito, decidiamo di inserire un operatore di *circular shift*, in grado di variare le dipendenze fra bit, conservando, allo stesso tempo, tutte le informazioni presenti in una word (circolarità). In realtà, decidiamo di inserire in totale sei operatori di shift circolare (CSH), al fine di effettuare traslazioni di 1, 2 e 4 bit, sia a destra (*right circular shift*) che a sinistra (*left circular shift*). Per quanto concerne la definizione dei simboli terminali, il terminal set sarà composto da tutte le word di input w_i che compongono un pattern, oltre all'inserimento di una *Ephemeral Random Constant* denominata "ERC".

Terminale	Tipo	Note
w_i	input	$\forall i \in \{0, ..., n-1\}$
ERC	$\Re$	intero senza segno a l bit (ossia ERC $\in [0, 2^{l-1}] \cap \mathbb{N}$)

Tabella 2.2: Terminal set

Fatto ciò, definiamo una **fitness function** $f : X_{gen} \to \mathbb{R}^+$, da minimizzare, per quantificare la capacità di un individuo nella classificazione dei pattern in ingresso. Essendo presenti delle variabili di input all'interno del terminal set, per poter determinare la fitness associata ad un individuo, è necessario eseguire il programma derivante dalla compilazione dell'albero sintattico più volte, con diverse configurazioni di tali input (*fitness cases*); nel nostro caso, contando il numero di falsi positivi FP e falsi negativi FN. Si ha un falso positivo, quando w non rappresenta la cifra i, ma l'individuo soggetto a va-

lutazione $x \in X_{gen}$, il cui scopo è quello del classificatore ideale c_i, restituisce $x(w) = \text{V}$, ossia che invece la rappresenta; mentre, si ha un falso negativo, quando w rappresenta la cifra i, ma $x(w) = \text{F}$ "afferma" il contrario. È dunque facile intuire che i dati sui quali i classificatori vengono valutati sono *labelled*, ossia etichettati con la classe di appartenenza, così da guidare l'intero processo di apprendimento (*Supervised Learning*). Siccome l'obiettivo dell'evoluzione è minimizzare il più possibile la funzione f, quest'ultima deve essere "costruita" in modo tale da restituire valori "bassi" sugli individui migliori e valori "alti" sugli individui peggiori. In realtà, poiché un individuo restituisce un valore a l bit (32 o 64, nel nostro caso), è come se al suo interno fossero presenti l classificatori che evolvono parallelamente, ma senza influire sulle prestazioni di calcolo della CPU (come invece accade per gli individui della popolazione), grazie alla potenza degli operatori bitwise; questa tecnica di evoluzione, che sfrutta il parallelismo intrinseco degli operatori bitwise, è chiamata **Sub-Machine-Code Genetic Programming**. Innanzitutto, per valutare ogni "sotto-classificatore" di un individuo, potremmo predisporre una funzione di fitness $f_{sub} : X_{gen} \times \{0, ..., l-1\} \to \mathbb{R}^+$ che associ al bit i-esimo del target dell'individuo x, il relativo valore di fitness $f_{sub}(x, i)$. Nel nostro caso, una funzione di fitness per il singolo bit potrebbe essere

$$f_{sub}(x, i) = \sqrt{50 \frac{FP_{x,i}^2 + FN_{x,i}^2}{(NP + NN)^2}}, \tag{2.1}$$

dove $FP_{x,i}$ e $FN_{x,i}$ sono rispettivamente il numero di falsi positivi e falsi negativi ottenuti sull'i-esimo bit del candidato classificatore x, mentre NP e NN rispettivamente il numero di positivi e negativi totali presenti nell'evaluation set (training set, per adesso). La fitness function dell'individuo x è presto definita: sarà semplicemente il minimo fra le fitness ottenute nei diversi bit del target, ossia

$$f(x) = \min\{f_{sub}(x, i) : 0 \leq i \leq l-1\}. \tag{2.2}$$

Infine, per evitare un possibile fenomeno di *bloat* (*i.e.* crescita incontrollata della dimensione degli alberi), introduciamo un vincolo al nostro problema

di ottimizzazione, inserendo una componente di penalità all'interno di f_{sub}, ossia

$$f_{sub}(x,i) = \sqrt{50\frac{FP_{x,i}^2 + FN_{x,i}^2}{(NP+NN)^2}} - 10^{-6} \cdot h_x, \tag{2.3}$$

dove h_x è il numero di nodi dell'albero sintattico x. In questo modo, a parità di valore dell'obiettivo, l'algoritmo preferirà l'albero sintattico con altezza minore.

Una volta delineata la funzione di fitness, possiamo passare alla scelta dei **parametri** e degli **operatori**.

Fase	Operatore	Note
selezione	tournament selection	sottomultiinsiemi di size pari a 7
ricombinazione	subtree crossover	
mutazione	subtree mutation	

Tabella 2.3: Operatori

Nome	Parametro	Valore
population size	size(P)	1000 individui (syntax trees)
probabilità di crossover	p_c	0.8
probabilità di mutazione	p_m	0.15

Tabella 2.4: Parametri

Infine, delineiamo una possibile **condizione di terminazione**. Per poter affermare la presenza di un ristagno nella popolazione, è necessario intravedere un rallentamento del miglioramento nei valori di fitness sugli individui della popolazione stessa. Potremmo assumere una condizione di ristagno (*i.e.*

terminazione dell'algoritmo) quando il minimo valore di f su P ($\min_P f$) non diminuisce per un certo numero di generazioni consecutive. A livello concettuale, stiamo sostanzialmente dicendo che l'algoritmo non è riuscito a determinare un individuo migliore di quello attuale per quel determinato numero di generazioni. Arrivati a quel punto, poi, dovremmo fare in modo che l'algoritmo restituisca proprio tale individuo migliore (anche se potrebbero essercene di più in P), ossia il miglior classificatore trovato fino ad ora (*best-so-far*). In realtà, la tecnica appena descritta non previene il fenomeno dell'**overfitting**; per questa ragione, è necessario apportare alcune modifiche a questa strategia.

2.2.1 Overfitting e Condizione di Terminazione

Durante l'evoluzione, nelle varie generazioni, gli individui della popolazione sono valutati da una funzione di fitness, che restituisce la valutazione migliore fra le prestazioni misurate sui diversi bit nel caso dei classificatori e le prestazioni ottenute sull'attuale classificatore migliore, al variare dell'embedding, nel caso di questi ultimi. Siccome l'ipotetica condizione di terminazione si basa sui migliori valori di fitness ottenuti sul training set, è come se si scegliesse l'individuo migliore sulla base della propria prestanza nella sola fase di allenamento (*training*), quindi sui soli dati su cui abbiamo ottimizzato il comportamento della popolazione. In alcuni casi, l'apprendimento può essere effettivamente efficace, ma in altri, come solitamente accade, degenera in una "memorizzazione" dei dati di input. In analogia con quanto affermato, è come se si testasse l'abilità di una classe di studenti di risolvere dei problemi con gli stessi identici esercizi svolti in classe; il docente non sarà in grado di distinguere gli studenti con alte albilità da quelli che hanno semplicemente imparato a memoria la risoluzione dei soli problemi svolti a lezione. Per evitare questa situazione, esattamente come il docente propone problemi differenti in fase di verifica, predisponiamo un altro set di dati (**validation set**), distinti da quelli del training set, per verificare l'effettivo apprendimento degli individui migliori (ottenuti sul training set). L'idea è quella di valutare

periodicamente l'individuo migliore della popolazione attraverso una sua valutazione con la stessa funzione di fitness utilizzata nell'evoluzione, ma con i dati del validation set, anziché del training set. Decidiamo di terminare il processo di evoluzione quando, per un certo numero di valutazioni consecutive (sul validation set) dell'individuo migliore ottenuto sul training set, non si ottengono valori di fitness migliori (in ogni caso, è comunque presente un limite al numero di generazioni). Limitatamente alla popolazione dei classificatori, anche la determinazione del bit migliore sul classificatore più prestante, attualmente presente nella popolazione, avverrà sulla base del validation set. Questa condizione di terminazione consente di fermare il processo di evoluzione poco prima dell'overfitting, così da massimizzare il più possibile le prestazioni dell'individuo. In una situazione più generale, dove la funzione di fitness deve essere minimizzata, il processo di evoluzione si articola in tre fasi (Figura 2.2):

1. **underfitting**, in cui la popolazione assume valori elevati di fitness sia sul training set che sul validation set;

2. **learning** (apprendimento), in cui, per diverse generazioni, la popolazione assume valori via via decrescenti sia sul training set che sul validation set;

3. **overfitting**, in cui le prestazioni sul training set migliorano o rimangono costanti, mentre quelle sul validation set via via peggiorano.

2.2.2 Elitarismo

La progettazione attuale dell'algoritmo contiene un piccolo problema. Se l'attuale individuo migliore all'interno della popolazione dovesse essere coinvolto in un'operazione di mutazione o di ricombinazione, questo potrebbe non essere più presente nella popolazione della generazione successiva. Il rischio è quello di perdere il possibile individuo migliore dell'algoritmo o di perdere, comunque, le sue caratteristiche genetiche che, in un'operazione di

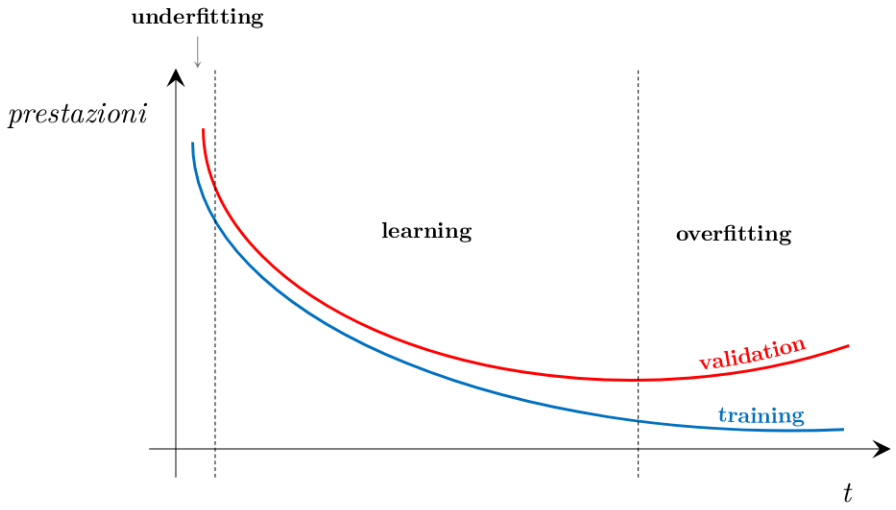


Figura 2.2: Underfitting, learning e overfitting

ricombinazione o mutazione, non è detto generino sempre individui con fitness non superiore a quella migliore. Per queste ragioni, si rende necessaria l'introduzione dell'**elitarismo** (o **elitismo**), ossia la sopravvivenza (*survival*) dell'individuo migliore presente nella popolazione (o più individui migliori). Ora, indicando con $P(t)$ la popolazione alla generazione t, si ha che $\min_{P(t)} f$ non cresce mai all'aumentare di t (al limite rimane uguale), ossia la funzione

$$\text{bestfit}(t) = \min_{P(t)} f \tag{2.4}$$

è tale che $\text{bestfit}(t) \geq \text{bestfit}(t+1), \forall t$, cioè è **debolmente decrescente**. A livello algoritmico, se dopo la fase di generazione della nuova popolazione, l'individuo migliore della generazione precedente non dovesse essere presente, tale individuo vi sarà inserito (rimuovendone uno casualmente e uniformemente dalla popolazione, così da mantenere $\text{size}(P(t))$ costante).

2.3 Embedding e Coevoluzione

In relazione all'obiettivo di questa tesi, per migliorare le prestazioni del classificatore, decidiamo di inserire come stadio intermedio fra l'emissione del

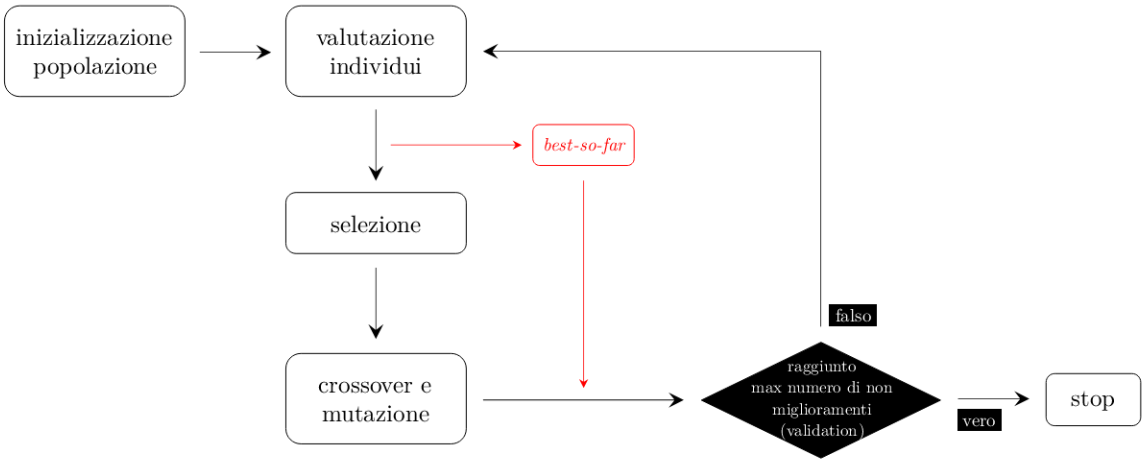
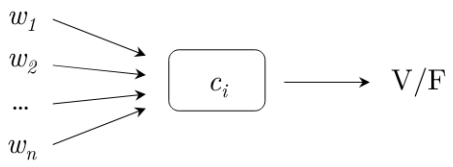
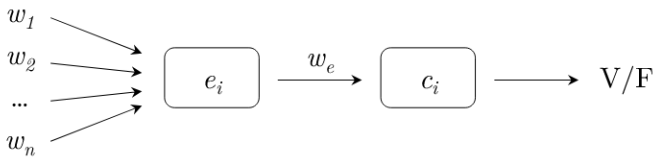


Figura 2.3: Algoritmo di evoluzione (l'elitarismo è in rosso)

pattern di input e il classificatore una funzione di embedding. Questo blocco riceve in input il pattern w e restituisce in output un nuovo pattern composto da un'unica feature w_e ottenuta tramite una qualche funzione $e_i(w)$, che intendiamo trovare (il pedice i indica l'embedding per il classificatore binario c_i per la label i). Ricordiamo che w è un vettore di quattro word a 32 bit oppure di due word a 64 bit, a seconda del meccanismo di *packed encoding* utilizzato. Il terminal set per gli alberi sintattici di classificazione verrà quindi modificato sostituendo le componenti w_i con la rappresentazione latente w_e. A questo punto l'obiettivo si identifica nella ricerca della giusta coppia $C_i = (e_i, c_i)$, dove ora $c_i : \mathbb{N} \to \{\mathrm{V}, \mathrm{F}\}$ e $e_i : \mathbb{N}^n \to \mathbb{N}$. L'idea è quella di evolvere separatemente una popolazione di candidati embedding e un'altra di candidati classificatori, per i quali il training set viene precedentemente trasformato secondo il miglior embedding trovato fino a quel punto. Inizialmente, la scelta dell'embedding deve per forza avvenire in modo casuale, cercando, comunque, di evitare la scelta di un embedding che, sicuramente, non potrà portare alcun beneficio (*e.g.* $e_i(w)$ costante). Una prima strategia di scelta potrebbe essere $e_i(w) = w_0$, così da creare variabilità fra le diverse rappresentazioni; un'altra potrebbe essere invece $e_i(w)$, la cui espressione analitica contiene effettivamente **tutte** le componenti w_j, per assicurare la



(a) Classificatore stand-alone



(b) Classificatore con embedding dell'input

Figura 2.4: Introduzione del sistema di embedding

dipendenza da un intero pattern (adotteremo quest'ultima politica). Successivamente, evolviamo una popolazione di classificatori P_c, al fine di decretare un buon classificatore per l'embedding ottenuto. Fissato tale classificatore, evolviamo una popolazione di embedding P_e per scegliere quello che meglio condensa le informazioni delle features per il classificatore. A questo punto si procede da capo, fissando, stavolta, l'embedding migliore ottenuto poc'anzi. Questo meccanismo è noto come **coevoluzione** e continua per un numero fissato di iterazioni. Il primitive set, la funzione di fitness e gli altri parametri di configurazione resteranno, di base, gli stessi già trattati per il classificatore stand-alone, con opportune modifiche del terminal set per c_i e della funzione di fitness per e_i.

2.3.1 Valutazione dell'Embedding

Per valutare un embedding della popolazione P_e, è necessario definire una funzione di fitness $f_e : X_{gen} \to \mathbb{R}^+$, da minimizzare, che associ valori "bassi" agli individui che portano buone prestazioni sul classificatore fissato c_i e "alti" sugli altri. L'idea è quella di trasformare il training set con l'individuo $x \in P_e$

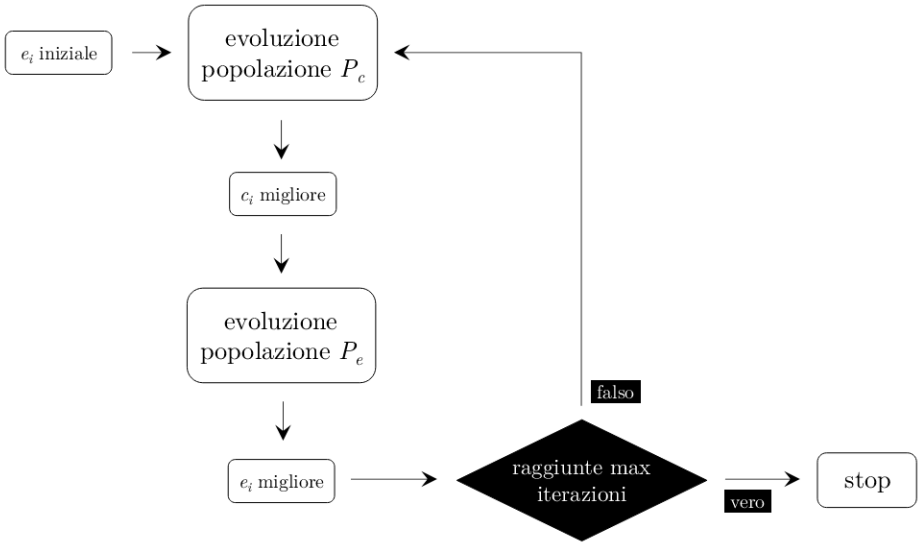


Figura 2.5: Algoritmo di coevoluzione

e, successivamente, valutare le prestazioni del classificatore c_i con la funzione di fitness f_{sub} (2.1), per il classificatore stand-alone, sul bit migliore di c_i. Concettualmente, quindi,

$$f_e(x) = f_{sub,x}(c_i, j_{best}), \tag{2.5}$$

dove il pedice x indica l'embedding utilizzato per la trasformazione del training set (o validation set, per la prevenzione dell'overfitting) e j_{best} il bit migliore di c_i.

2.3.2 Elitarismo esteso e Sopravvivenza

Siccome stiamo parlando di coevoluzione e non più di evoluzione, l'algoritmo descritto precedentemente pone un piccolo problema: ad ogni iterazione la popolazione viene reinizializzata da zero. Seguendo alla lettera questo schema, si rischia, nella maggior parte dei casi, di riottenere un albero sintattico con prestazioni o fitness paragonabili al precedente, sia per i classificatori che per gli embedding. Per ovviare a tale problematica, si potrebbe pensare di salvare l'individuo migliore della popolazione e, all'iterazione successiva,

di reinserirlo all'interno della nuova popolazione iniziale. Ricordando che le singole evoluzioni sono elitarie, ossia assicurano che l'individuo migliore (**sul training set**) sopravviva da una generazione all'altra limitatamente alla singola evoluzione, ora, grazie all'**estensione dell'elitarismo** fra due evoluzioni distinte (dello stesso tipo di individui), l'individuo migliore ottenuto, questa volta, **sul validation set** sopravvivrà anche fra due iterazioni della coevoluzione. Attenzione: questo **non** significa che la funzione bestfit(t) sia debolmente decrescente anche fra le evoluzioni nelle varie iterazioni; ossia, indicando con j il numero dell'iterazione,

$$\textbf{non è vero che } \text{bestfit}_j(t_1) \geq \text{bestfit}_{j+1}(t_2), \forall j, t_1, t_2. \tag{2.6}$$

La motivazione di non sussistenza della precedente disuguaglianza deriva semplicemente dal fatto che l'individuo migliore che sopravvive tra un'iterazione e quella successiva è determinato dal validation set, e non dal training set, come invece accade per l'elitarismo nella singola evoluzione. Creatosi un legame fra i processi evolutivi nelle varie iterazioni, è possibile estendere le condizioni di terminazione delle due evoluzioni a condizioni di terminazione dell'algoritmo coevolutivo (oltre a quella già presente). Sarà quindi necessario contare i non miglioramenti con un unico contatore su entrambe le evoluzioni, così da arrestare l'algoritmo nel caso in cui non si ottengano prestazioni migliori per un certo numero di generazioni, indipendentemente dalla popolazione (*e.g.* una parte di generazioni potrebbe riguardare la popolazione P_c, mentre l'altra P_e, senza che il conteggio riparta da zero fra un'evoluzione e l'altra). Grazie a questi due cambiamenti, la struttura dell'algoritmo coevolutivo assume l'aspetto di Figura 2.5. In realtà, per non perdere le buone caratteristiche genetiche degli individui dell'ultima popolazione $P_j(t_{last})$ ottenuta dall'evoluzione precedente, anziché garantire l'inserimento del solo individuo migliore sul validation set $x_{best,j}$ nella popolazione iniziale $P_{j+1}(0)$ dell'evoluzione successiva, sarebbe più opportuno assicurare la **sopravvivenza** di un'intera porzione $S_j \subseteq P_j(t_{last})$ di individui. Ordinando gli individui nella popolazione $P_j(t_{last})$ per valori debolmente crescenti di fitness (valutata sul training set), identifichiamo i diversi quartili della popolazione

con $P_j^q(t_{last})$, dove $q \in \{1,2,3,4\}$; sia $s \in [0,1]$ la percentuale di popolazione che sopravvive (0.6, nel nostro caso). Invece, la porzione di popolazione che sopravvive sarà

$$S_j = \bigcup_{q=1}^{4} S_j^q, \tag{2.7}$$

dove S_j^q è il sottoinsieme di $P_j^q(t_{last})$ formato dai $\lfloor \frac{10}{5-q} \cdot s \cdot \text{size}(P_j^q(t_{last})) \rfloor$ individui migliori di quest'ultimo; in questo modo S_j sarà composto dagli individui migliori del primo quartile per il 40%, del secondo per il 30%, del terzo per il 20% e del quarto per il 10%. Nel caso $x_{best,j}$ non dovesse essere stato inserito, si provvede di conseguenza (l'individuo migliore sul validation deve sopravvivere). Siccome appena utilizzato, per completezza, forniamo anche il concetto di unione fra multiinsiemi.

Definizione 2.1. *Siano $S_1 = (A_1, m_1)$ e $S_2 = (A_2, m_2)$ due multiinsiemi. Chiamiamo **unione** di S_1 e S_2 il multiinsieme*

$$S_1 \cup S_2 = (A_1 \cup A_2, m), \text{ dove } m(a) = \begin{cases} m_1(a) & se \quad a \in A_1 \setminus A_2 \\ m_2(a) & se \quad a \in A_2 \setminus A_1 \\ m_1(a) + m_2(a) & se \quad a \in A_1 \cap A_2 \end{cases}. \tag{2.8}$$

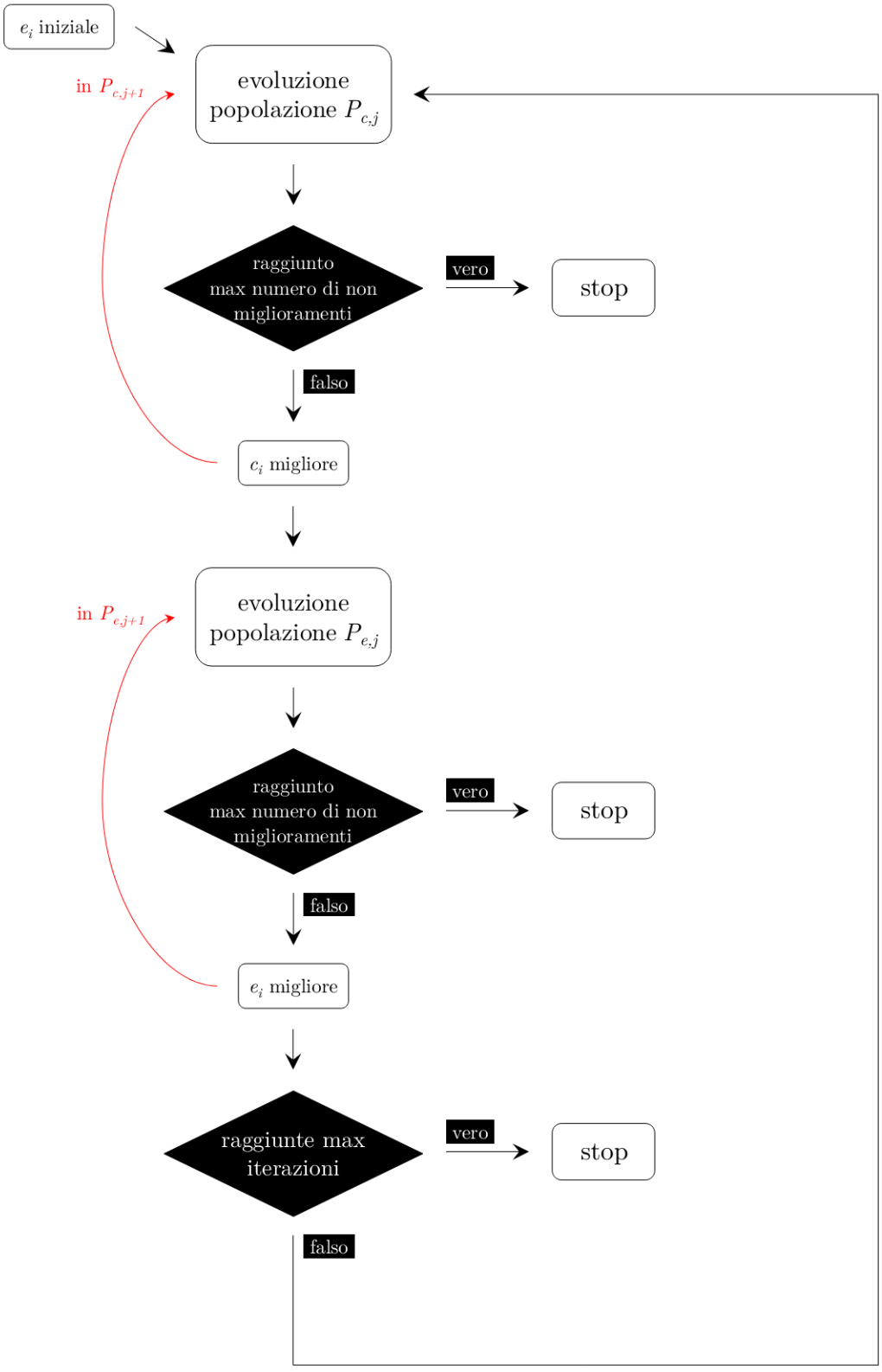


Figura 2.6: Coevoluzione elitaria con estensione delle condizioni di terminazione (l'elitarismo è in rosso e j indica il numero dell'iterazione)

Capitolo 3

Implementazione

In questo capitolo presentiamo gli stumenti impiegati nella realizzazione del sistema, per poi addentrarci nel codice, cercando sempre di mantenere il parallelismo con la progettazione dell'algoritmo.

3.1 Strumenti impiegati

Per la stesura del codice è stato impiegato Python (v3.11), un linguaggio di programmazione di alto livello ideato da Guido van Rossum all'inizio degli anni '90. In particolare, supporta i paradigmi di programmazione procedurale, orientata agli oggetti e funzionale. È un linguaggio interpretato, ossia non è prevista una fase di compilazione del codice volta alla generazione di un file target eseguibile; tuttavia, in Python, è presente una fase intermedia che disaccoppia il codice dall'esecuzione effettiva: la creazione di un file bytecode, che permette di evitare una continua interpretazione del codice effettivo, aumentando le prestazioni. Se paragonato ai linguaggi compilati (*e.g.* C, C++), queste ultime non sono sicuramente un punto di forza di Python, soprattutto nelle operazioni di calcolo. In ogni caso è possibile riscrivere le parti di programma più costose in termini computazionali in un linguaggio compilato di prestazioni più elevate.

Figura 3.1: Logo di Python

Per i nostri scopi, decidiamo di utilizzare la libreria DEAP (*Distributed Evolutionary Algorithms in Python*), che offre un framework di sviluppo per algoritmi evolutivi e, in particolare, di Programmazione Genetica. Anche se tale libreria fornisce algoritmi evolutivi già preimpostati, purtroppo non implementa l'elitarismo (essenziale per il nostro progetto); di conseguenza, sarà necessaria una modifica interna all'algoritmo di evoluzione che meglio rappresenta quello visto in fase di progettazione (Figura 2.2), con opportune modifiche alla condizione di terminazione: *eaSimple*.

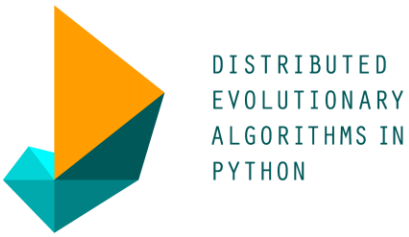


Figura 3.2: Logo di DEAP

3.2 Realizzazione

Partiamo delineando gli scopi dei diversi moduli del progetto. Troviamo innanzitutto i moduli delegati alla strutturazione dei dati, quali

- `dataset_structuring.py`, per dare una struttura uniformata ai pattern dotati di etichetta (*label*);

- `dataset_sectioning.py`, per la divisione del dataset in training set, validation set e test set.

Proseguiamo elencando i moduli più importanti, ossia

- `ea_simple.py`, che contiene l'algoritmo omonimo di DEAP, con l'aggiunta dell'elitarismo della singola evoluzione, e la funzione per la sopravvivenza di una porzione di popolazione fra due iterazioni successive;
- `embedding.py`, per l'evoluzione della popolazione di embedding;
- `classifier.py`, per l'evoluzione della popolazione di classificatori;
- `_cfg_params.py`, che contiene tutti i parametri di configurazione per le due evoluzioni precedenti;
- `coevolution.py`, dove è presente l'algoritmo di coevoluzione.

Sono presenti infine anche gli script

- `testing.py`, per la fase di testing del sistema finale complessivo (*i.e.* embedding e classificatore);
- `multiple_coevolutions.py`, utilizzato semplicemente per svolgere più coevoluzioni contemporaneamente (con cifre e/o seed di generazione di numeri casuali differenti).

3.2.1 Modulo `_cfg_params.py`

Presentiamo interamente il modulo che contiene la parametrizzazione dell'intera coevoluzione e, quindi, anche delle singole evoluzioni.

```
# main
MAX_ITERATIONS = 25 # number of max coevolution iterations
DEFAULT_SEED = 3612 # default seed for random function
    initializing (if not specified)
```

```
4 SAVE_THE_FIRST = False # the first random embedding tree will
      be saved with the invalid validation score -1 (True ->
     classifier before embedding, False -> viceversa)
5 P = 0.6 # portion (from 0 to 1) of population to maintain
     between one evolution and the next (both classifier and
     embedding), always including the best-so-far individual to
      guarantee elitism
6 MINIMIZE = True # the fitness function must be minimized (
     both classifier and embedding ones)
7 NUM_WORSE_VALID_TERMINATION = 6 # number of successive
     validations without improvement to terminate the
     coevolution process
8 RESTART_NON_IMPROVEMENT_CNT = False # restart to 0 the non-
     improvement-count between two phases
9 START_EMBEDDING_RANDOM = True # True if start embedding must
     be randomly choosen from trees which contain all input
     words 'wi', False if it must be 'w0'
10
11 # dataset sectioning
12 DATASET_PATH = '../data/dataset32.dat'
13 NUM_CLASS = 10 # number of class (enumerated from 0 to
     NUM_CLASS-1)
14 TRAINING_DATA_PERCENTAGE = 0.8 # dataset section (from 0 to
     1) for training data (training set + validation set)
15 VALIDATION_SET_PERCENTAGE = 0.2 # training data section for
     validation (from 0 to 1)
16
17 # inputs
18 WORD_BIT = 32 # feature word bit number
19 DIM_PATTERN = 4 # number of components for each pattern in
     dataset (excluding label)
20
21 # classifier
22 CL_FG_MIN_TREE_HEIGHT = 1 # only for the first generation
23 CL_FG_MAX_TREE_HEIGHT = 7 # only for the first generation
24 CL_TOURN_SIZE = 7 # multisubset population size in tournament
      selection
```

```
25 CL_MUT_APP_MIN_TREE_HEIGHT = 1 # max height of appendable
       tree in mutation
26 CL_MUT_APP_MAX_TREE_HEIGHT = 6 # min height of appendable
       tree in mutation
27 CL_CX_MAX_TREE_HEIGHT = 12 # max reachable height during
       crossover
28 CL_MUT_MAX_TREE_HEIGHT = 12 # max reachable height during
       mutation
29 CL_CX_MAX_POP_SIZE = 4096 # max reachable population size
       during crossover
30 CL_MUT_MAX_POP_SIZE = 4096 # max reachable population size
       during mutation
31 CL_WEIGHT_IND_LENGTH_PENALTY = 0.000001 # individual length
       penalty weight (constrained problem)
32 CL_INI_POP_SIZE = 1000 # initial population size
33 CL_CX_PB = 0.8 # crossover probability
34 CL_MUT_PB = 0.15 # mutation probability
35 CL_GEN = 20 # max number of generations to evolve a label
       classifier
36 CL_GEN_PER_VALIDATION = 3 # number of generations after which
        a validation takes place
37 CL_HOF_MAX_SIZE = 1 # hall-of-fame max size (however, also
       for different value, we'll evaluate only the first best
       individual)
38
39 # embedding
40 EMB_FG_MIN_TREE_HEIGHT = 1 # only for the first generation
41 EMB_FG_MAX_TREE_HEIGHT = 7 # only for the first generation
42 EMB_TOURN_SIZE = 7 # multisubset population size in
       tournament selection
43 EMB_MUT_APP_MIN_TREE_HEIGHT = 1 # max height of appendable
       tree in mutation
44 EMB_MUT_APP_MAX_TREE_HEIGHT = 6 # min height of appendable
       tree in mutation
45 EMB_CX_MAX_TREE_HEIGHT = 12 # max reachable height during
       crossover
46 EMB_MUT_MAX_TREE_HEIGHT = 12 # max reachable height during
       mutation
```

```
47 EMB_CX_MAX_POP_SIZE = 4096 # max reachable population size
   during crossover
48 EMB_MUT_MAX_POP_SIZE = 4096 # max reachable population size
   during mutation
49 EMB_WEIGHT_IND_LENGTH_PENALTY = 0.000001 # individual length
   penalty weight (constrained problem)
50 EMB_INI_POP_SIZE = 1000 # initial population size
51 EMB_CX_PB = 0.8 # crossover probability
52 EMB_MUT_PB = 0.15 # mutation probability
53 EMB_GEN = 20 # max number of generations to evolve a label
   embedding
54 EMB_GEN_PER_VALIDATION = 3 # number of generations after
   which a validation takes place
55 EMB_HOF_MAX_SIZE = 1 # hall-of-fame max size (however, also
   for different value, we'll evaluate only the first best
   individual)
```

Listing 3.1: `_cfg_params.py`

3.2.2 Singole evoluzioni

Innanzitutto, importiamo i moduli di DEAP necessari.

```
1 from deap import base # which contains base classes to
   inherit
2 from deap import creator # which allows us to create new
   types starting from a base type
3 from deap import tools # which contains the operators such as
    selection, crossover and mutation
4 from deap import gp # which contains factory method to
   instantiate the primitive set
```

Listing 3.2: Import necessari per DEAP

Mostriamo le parti più importanti del codice relativo ai moduli delle singole evoluzioni. Partiamo dal classificatore, con il modulo `classifier.py`. La seguente funzione mostra la definizione del primitive set attraverso le primitive di DEAP.

```
def define_primitive_set() -> gp.PrimitiveSet:
    # creator settings
    creator.create("FitnessMin", base.Fitness, weights=(-1.0 if MINIMIZE else +1.0,)) # there is only a fitness to be minimized
    creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin) # an individual is a tree, but contains its fitness

    pset = gp.PrimitiveSet('pset', 1)

    # terminal set (T)
    pset.renameArguments(ARG0='we') # embedding argument (we = emb(w0, w1, ...))
    pset.addEphemeralConstant(f'ERC%d_cl' % define_primitive_set.count, lambda : random.randint(0, 2**WORD_BIT - 1)) # ephemeral random constant

    # function set (F)
    pset.addPrimitive(op.__and__, 2)
    pset.addPrimitive(op.__or__, 2)
    pset.addPrimitive(op.__invert__, 1)
    pset.addPrimitive(op.__xor__, 2)
    pset.addPrimitive(nand, 2)
    pset.addPrimitive(nor, 2)
    pset.addPrimitive(lcshf, 1)
    pset.addPrimitive(rcshf, 1)
    pset.addPrimitive(lambda a : circShift(a, 2, False), 1, 'lcshf2') # double left circular shift
    pset.addPrimitive(lambda a : circShift(a, 2, True), 1, 'rcshf2')
    pset.addPrimitive(lambda a : circShift(a, 4, False), 1, 'lcshf4')
    pset.addPrimitive(lambda a : circShift(a, 4, True), 1, 'rcshf4')

```

```
26      return pset
```

Listing 3.3: Definizione del primitive set (`classifier.py`)

Inizialmente (righe 3 e 4), abbiamo la definizione del tipo di problema di ottimizzazione (di minimo, nel nostro caso) e della funzione costruttrice di individui. In particolare alla prima definizione, DEAP consente di lavorare con problemi **multi-objective**; infatti, per ogni sotto-obiettivo è possibile difinire la funzione di fitness e il relativo peso. In generale, un peso negativo indica che la relativa fitness sarà da minimizzare, mentre un peso positivo indica che sarà da massimizzare; inoltre, più il modulo di un peso è maggiore, più l'obiettivo parziale sarà influente. Nel nostro caso, siccome la funzione di fitness è soltanto una ed è da minimizzare, l'ennupla di pesi è costituita da un unico peso negativo: scegliamo -1. I risultati non cambierebbero se al posto di -1 fosse presente un altro numero negativo; tuttavia, si avrebbero valori di fitness amplificati del modulo di tale numero. Alla riga 6 abbiamo l'istanziazione del primitive set tramite indicazione di `name` e `num_input` (numero di nodi di input nel terminal set). Per quanto riguarda il terminal set, con la primitiva `renameArguments(·)`, invocata sul primitive set, è possibile rinominare i nodi terminali di input o ridefinirne il prefisso (`ARG` di default). È inoltre possibile aggiungere una ERC (*Ephemeral Random Constant*, $\Re$) attraverso la primitiva `addEphemeralConstant(name,func)` che, per la generazione della costante effimera utilizza il valore di ritorno della funzione `func` (nel nostro caso `randint`, della libreria `random`, riga 10). Per quanto concerne il function set, l'aggiunta di un operatore/funzione è possibile tramite la primitiva `addPrimitive(operator,arity)`, indicandone anche l'arietà (da riga 13). Siccome si lavora su numeri naturali con l'applicazione di soli operatori bitwise, si è deciso di adottare una definizione del primitive set di tipo **loosely typed**, ossia senza specificare i tipi di dato accettati in input dagli operatori. DEAP mette a disposizione anche la variante **strongly typed**, che forza la creazione di alberi nei quali i singoli nodi possono essere subordinati ai soli altri nodi che accettino in input un dato dello stesso tipo di quello restituito dal nodo in subordine (in caso di nodi terminali, l'output

di questi è presto determinato).

Una volta definita la funzione di fitness, possiamo passare alla creazione del `toolbox`, che ingloba tutte le componenti fondamentali per l'algoritmo di GP.

```
1  def define_gp_params(label_classifier: int, pset: gp.
       PrimitiveSet, training_set: list) -> base.Toolbox: #
       toolbox creation
2      toolbox = base.Toolbox()
3
4      toolbox.register("expr", gp.genHalfAndHalf, pset=pset,
       min_=CL_FG_MIN_TREE_HEIGHT, max_=CL_FG_MAX_TREE_HEIGHT) #
       some kind of gene (allele)
5      toolbox.register("individual", tools.initIterate, creator
       .Individual, toolbox.expr) # individual (a syntax tree)
6      toolbox.register("population", tools.initRepeat, list,
       toolbox.individual) # population
7      toolbox.register("compile", gp.compile, pset=pset) # tree
        compilation -> a program (a function)
8
9      toolbox.register("evaluate", lambda individual : (
       fitness_for(label_classifier, individual, training_set,
       toolbox.compile),)) # fitness function
10     toolbox.register("select", tools.selTournament, tournsize
       =CL_TOURN_SIZE) # selection (tournament selection)
11
12     toolbox.register("mate", gp.cxOnePoint) # crossover (one-
       point cx)
13
14     toolbox.register("expr_mut", gp.genFull, min_=
       CL_MUT_APP_MIN_TREE_HEIGHT, max_=
       CL_MUT_APP_MAX_TREE_HEIGHT) # subtree to append in
       mutation
15     toolbox.register("mutate", gp.mutUniform, expr=toolbox.
       expr_mut, pset=pset) # mutation (subtree mutation)
16
17     toolbox.decorate("mate", gp.staticLimit(key=op.attrgetter
```

```
    ("height"), max_value=CL_CX_MAX_TREE_HEIGHT)) # tree
    height limits (bloat)
18   toolbox.decorate("mutate", gp.staticLimit(key=op.
    attrgetter("height"), max_value=CL_MUT_MAX_TREE_HEIGHT))
19
20   toolbox.decorate("mate", gp.staticLimit(key=len,
    max_value=CL_CX_MAX_POP_SIZE)) # pop size limits
21   toolbox.decorate("mutate", gp.staticLimit(key=len,
    max_value=CL_MUT_MAX_POP_SIZE))
22
23   return toolbox
```

Listing 3.4: Creazione del toolbox (`classifier.py`)

La registrazione di una funzione all'interno del toolbox avviene tramite il metodo `register(alias,func,args)`, specificandone alias e argomenti. Per esempio, a riga 6, la popolazione viene creata tramite la ripetizione della funzione per la creazione degli individui `toolbox.individual(·)`, precedentemente registrata. Inoltre, per evitare un possibile fenomeno di **bloat**, è stata imposta una limitazione all'altezza degli alberi durante operazioni di mutazione o di ricombinazione (righe 17 e 18). Alle righe 20 e 21 troviamo, infine, la limitazione alla size della popolazione di classificatori nelle fasi di ricombinazione e di mutazione (fissata a 4096).

Di tutte le componenti registrate nel toolbox, ci concentriamo in particolare sulla funione di fitness.

```
1 def bit_fitness(ind_length: int, fp: int, fn: int, totp: int,
      totn: int) -> float: # return bit fitness value
2     return np.sqrt(50*(fp**2 + fn**2) / (totp + totn)**2) +
    ind_length*CL_WEIGHT_IND_LENGTH_PENALTY # the second
    adding is the length-proportional individual penalty
3
4 def fitness_bestbit_for(label_classifier: int, individual: gp
    .PrimitiveTree, evaluation_set: list, compile: callable)
    -> tuple[float, int]: # fitness function and the best bit
    for all classifier
```

```
5     confusion_matrix = {}
6     confusion_matrix['TP'] = [0] * WORD_BIT # it isn't a bit,
      but a count value (true positive)
7     confusion_matrix['TN'] = [0] * WORD_BIT # true negative
8     confusion_matrix['FP'] = [0] * WORD_BIT # false positive
9     confusion_matrix['FN'] = [0] * WORD_BIT # false negative
10    tot_P, tot_N = 0, 0 # total of positives and negatives
11
12    func_ind = compile(individual)
13    for features in evaluation_set:
14        result = func_ind(we=features[0])
15        positive = (features[1] == label_classifier) #
    features[1] is the label of the input features[0]
16        if positive:
17            tot_P += 1
18        else:
19            tot_N += 1
20        bitmask = 1 # 000...001b, WORD_BIT bits
21        for j in range(WORD_BIT):
22            bit_j = result & bitmask
23            str_case = ('TP' if bit_j else 'FN') if positive
    else ('FP' if bit_j else 'TN')
24            confusion_matrix[str_case][j] += 1
25            bitmask <<= 1
26    bit_fitness_lst = [bit_fitness(len(individual),
    confusion_matrix["FP"][j], confusion_matrix["FN"][j],
    tot_P, tot_N) for j in range(WORD_BIT)]
27    min_bit_fitness = min(bit_fitness_lst) # final fitness
    value is the minimum among all bit fitness values
28
29    return min_bit_fitness, bit_fitness_lst.index(
    min_bit_fitness)
```

Listing 3.5: Fitness function (`classifier.py`)

Inizialmente (riga 1) viene definita la funzione `bit_fitness(·)` per la valutazione del singolo bit in output al classificatore, ossia la funzione che incarna la corrispondente funzione matematica f_{sub}, definita a (2.3). Successivamen-

te (riga 4), si ha la scrittura della funzione per la valutazione dell'albero sintattico (di tipo `gp.PrimitiveTree`) corrispondente ad un classificatore (`fitness_bestbit_for(·)`). Per ogni bit, viene creata una *confusion matrix* per contenere le prestazioni ottenute su tale bit in termini di veri positivi, veri negativi, falsi positivi e falsi negativi (righe da 5 a 9). L'individuo viene inizialmemte trasformato in una funzione attraverso la primitiva `compile(·)` del modulo `gp` (riga 12); in seguito, per ogni pattern presente nell'evaluation set (training set o validation set, a seconda dei casi), già trasformato con l'embedding fissato e_i, si verifica, per ogni bit, in quale caso ricade la suddetta predizione (righe da 13 a 25). In particolare, siccome `bitmask` all'interazione j vale 2^j, la codifica binaria avrà solo un bit a 1 (quello in posizione j); di conseguenza, il valore di `bit_j = result & bitmask` sarà pari a 0 se il bit di posizione j di `result` è 0, mentre sarà pari a 2^j se lo stesso bit è a 1. Anche se la variabile `bit_j` non è booleana, di fatto viene considerata come tale, in quanto non interessa il suo reale valore, ma solo se è uguale o diversa da 0. Utilizzando la funzione `bit_fitness(·)`, vengono valutati tutti i bit attraverso i valori contenuti nelle matrici di confusione e al numero di nodi dell'albero sintattico dell'individuo (ovviamente la stessa per tutti i bit). La fitness dell'individuo corrisponde al valore minimo tra tutti i valori appena calcolati, come da definizione di f a (2.2). Più precisamente, la funzione restituisce una coppia di valori: il primo corrisponde al minimo valore di fitness ottenuto sui diversi bit del classificatore, mentre il secondo corrisponde al peso di uno dei bit migliori (riga 29).

Decidiamo di omettere l'algoritmo di evoluzione della popolazione di classificatori, perché strettamente fedele all'algoritmo trattato in fase di progettazione (Figura 2.3).

Per quanto riguarda l'evoluzione della popolazione di embedding, rispetto alla parte vista precedentemente per i classificatori, anche se con qualche modifica, è sostanzialmente analoga e, per questa ragione, ne omettiamo la

trattazione. Tuttavia, mostriamo comunque una componente differente: la fitness function f_e.

```
1  def fitness_for(individual: gp.PrimitiveTree, evaluation_set:
       list, bsf_bin_cl: tuple[int, callable], compile: callable
   ) -> tuple[float, int]: # fitness function and the best
   bit for all classifier
2      trans_eval_set = data_transformation(evaluation_set,
   compile(individual))
3      label, bin_cl_func = bsf_bin_cl
4      fp, fn, totp, totn = 0, 0, 0, 0
5
6      for features in trans_eval_set:
7          real = features[1] == label
8          prediction = bin_cl_func(features[0])
9          if real:
10             totp += 1
11             if not prediction:
12                 fn += 1
13         else:
14             totn += 1
15             if prediction:
16                 fp += 1
17     return bit_fitness(len(individual), fp, fn, totp, totn)
```

Listing 3.6: Fitness function (`embedding.py`)

Come da progetto, la funzione appena definita trasforma l'evaluation set (training set o validation set) con l'albero sintattico da valutare `individual` (riga 2) e restituisce le prestazioni ottenute sul classificatore `bsf_bin_cl`: quello migliore fino ad ora (righe da 6 fino alla fine).

3.2.3 Coevoluzione

Giunti all'apice del livello di astrazione, mettiamo insieme i diversi moduli per formare l'algoritmo di coevoluzione.

```
1  label = int(argv[1])
```

```
random.seed(int(argv[2] if len(argv) == 2+1 else DEFAULT_SEED
    )) # random seed initializing
training_set = read_data_from('../data/training_set.dat')
validation_set = read_data_from('../data/validation_set.dat')
bsf_bin_classifier = (label, None)
emb_survival, cl_survival = [], []
nic = 0 # non-improvement-count
emb_valid_score, cl_valid_score = None, None
for iter in range(MAX_ITERATIONS):
    print(f'--- Iter %d ---' % iter) # start iter log

    # evolving embedding (with fixed classifier, after first
    embedding) - phase E
    embedding, emb_valid_score, emb_survival, nic = embmain(
    training_set, validation_set, bsf_bin_classifier,
    emb_survival, nic, emb_valid_score)
    if nic == NUM_WORSE_VALID_TERMINATION:
        early_exit_log()
        break

    # data transformation
    trans_training = data_transformation(training_set,
    embedding)
    trans_validation = data_transformation(validation_set,
    embedding)

    # evolving binary classifier (with fixed embedding) -
    phase C
    bsf_func, best_bit, cl_valid_score, cl_survival, nic =
    clmain(label, trans_training, trans_validation,
    cl_survival, nic, cl_valid_score)
    if nic == NUM_WORSE_VALID_TERMINATION:
        early_exit_log()
        break
    bsf_bin_classifier = label, partial((lambda we, bsf_func,
    best_bit : (bsf_func(we) & 2**best_bit) != 0), bsf_func=
```

```
    bsf_func, best_bit=best_bit)
```

Listing 3.7: Coevoluzione (`coevolution.py`)

Inizialmente, si procede con l'inizializzazione delle variabili utili al processo di coevoluzione (righe da 1 a 8). Quest'ultima si articola in tre fasi:

1. **fase E** (righe 12 e 13), dove viene richiamato il punto di ingresso (*i.e.* la funzione principale) del modulo `embedding.py`, per la determinazione di un buon `embedding` in ingresso al classificatore specificato `bsf_bin_classifier` (alla prima iterazione, quando `iter` vale 0, siccome la seconda componente di `bsf_bin_classifier` vale `None`, la funzione `embmain(·)` sceglie l'embedding iniziale fra un insieme di alberi aventi, fra i nodi terminali, tutte le componenti w_i);

2. **trasformazione dei dati** (righe 19 e 20), dove il training set e il validation set subiscono la trasformazione indotta dalla migliore rappresentazione latente appena determinata;

3. **fase C** (riga 23), nella quale viene richiamato il punto di ingresso del modulo `classifier.py` per la determinazione del miglior classificatore (`bsf_func`, già ridotto a funzione) in cascata all'embedding predeterminato.

Inoltre, alle righe 14 e 24 troviamo le due condizioni di terminazione estese dalle singole evoluzioni; per questa ragione, oltre ai parametri già scambiati, troviamo anche il numero di non miglioramenti consecutivi `nic`, continuamente scambiato fra le due evoluzioni, e le valutazioni migliori sul validation set `emb_valid_score` (per la sola fase E) e `cl_valid_score` (per la sola fase C). La struttura del codice è dunque la stessa vista in fase di progettazione (Figura 2.6), con l'aggiunta della sopravvivenza di parte delle due popolazioni (`emb_survival` e `cl_survival`). In aggiunta, è comunque possibile cambiare la politica di scelta del primo embedding con quella $e_i(w) = w_0$: basta impostare su `False` il parametro `START_EMBEDDING_RANDOM` nel modulo `_cfg_params.py`.

3.3 Generalizzazione

Fino ad ora abbiamo trattato la risoluzione di un problema di ottimizzazione avente come scopo la determinazione della migliore coppia $C_i = (e_i, c_i)$ embedding-classificatore per ogni cifra i. In realtà, grazie all'alto grado di parametrizzazione raggiunto, il problema attuale può essere adattato a diversi ambiti (non soltanto alle cifre), tutti caratterizzati dalle seguenti due proprietà generali:

- l'obiettivo si identifica nella ricerca di `NUM_CLASS` coppie classificatrici $C_i = (e_i, c_i)$ o classificatori *one-vs-all* (uno per ogni classe, $i = 0, 1, ..., \texttt{NUM\_CLASS} - 1$);
- i pattern da classificare sono patterns composti da `DIM_PATTERN` componenti naturali con valori da 0 a $2^{\texttt{WORD\_BIT}-1}$.

Il sistema realizzato è quindi indipendente dal tipo di pattern da classificare: questo significa che è **generale**.

Capitolo 4

Risultati

In quest'ultimo capitolo presentiamo i risultati raggiunti con il sistema realizzato, effettuando un paragone con il classificatore stand-alone. Prima di cominciare, introduciamo alcuni concetti statistici di misurazione delle prestazioni.

4.1 Statistiche di Testing

Siano C_i un classificatore binario per la classe i, w un pattern di input, $r_{i,w}$ il valore di verità dell'enunciato "w appartiene alla classe i" e $C_i(w)$ la predizione del classificatore. Chiamiamo:

- **vero positivo** (*true positive*) una situazione per la quale $r_{i,w} = \mathrm{V}$ e $C_i(w) = \mathrm{V}$;
- **falso positivo** (*false positive*) una situazione per la quale $r_{i,w} = \mathrm{F}$ e $C_i(w) = \mathrm{V}$;
- **vero negativo** (*true negative*) una situazione per la quale $r_{i,w} = \mathrm{F}$ e $C_i(w) = \mathrm{F}$;
- **falso negativo** (*false negative*) una situazione per la quale $r_{i,w} = \mathrm{V}$ e $C_i(w) = \mathrm{F}$.

Indichiamo il totale di veri positivi con TP, quello di falsi positivi con FP, quello di veri negativi con TN e quello di falsi negativi con FN. La matrice quadrata

$$\begin{pmatrix} TP & FP \\ FN & TN \end{pmatrix} \tag{4.1}$$

è detta **confusion matrix**.

Per misurare le prestazioni del classificatore binario, conviene innanzitutto calcolare questi quattro valori sul test set; successivamente, solo in caso di test set bilanciato (con pari numero di positivi e numero di negativi), è possibile calcolare l'accuratezza del classificatore:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}. \tag{4.2}$$

In caso di test set sbilanciato (come nel nostro caso, dove il numero di positivi è pari a $\frac{1}{10}$ dei patterns totali) l'accuracy può risultare una misura ingannevole. Per questa ragione, esistono altre metriche da utilizzare in queste situazioni; in questo progetto utilizzeremo le due seguenti:

- **sensitivity** (o **True Positive Rate**, **TPR**), ossia la probabilità che il risultato del test sia positivo (appartenenza alla classe) quando il campione di test è veramente positivo:

$$\text{TPR} = \mathbb{P}(C_i(w) = \text{V} | r_{i,w} = \text{V}) = \frac{TP}{TP + FN}; \tag{4.3}$$

- **specificity** (o **True Negative Rate**, **TNR**), ossia la probabilità che il risultato del test sia negativo (non appartenenza alla classe) quando il campione di test è veramente negativo:

$$\text{TNR} = \mathbb{P}(C_i(w) = \text{F} | r_{i,w} = \text{F}) = \frac{TN}{TN + FP}. \tag{4.4}$$

4.2 Risultati raggiunti

Per scelta, decidiamo di mostrare i risultati relativi ai classificatori binari per le cifre considerate più "ostiche", ossia quelle che solitamente vengono

riconosciute con maggiore difficoltà. Per avere un confronto rispetto al classificatore stand-alone, recuperiamo le prestazioni di quest'ultimo dalla tesi di laurea di Fabrizio De Santis (a.a. 2019-2020). Tali prestazioni sono riportante nella Tabella 4.1.

		TPR				**TNR**			
label	#runs	max	min	avg	d.s.	max	min	avg	d.s.
0	5	97.41	88.02	88.70	8.98	99.71	98.78	99.25	0.45
5	5	90.02	86.03	87.90	1.32	99.27	99.07	99.29	0.21
6	5	92.22	79.04	83.91	4.48	98.71	97.98	98.64	0.38
7	5	94.41	93.21	93.90	0.43	99.31	99.67	99.60	0.14
8	5	88.02	85.63	86.35	0.96	99.60	99.33	99.41	0.14

Tabella 4.1: Risultati per il **classificatore stand-alone** con controllo sull'overfitting [Fabrizio De Santis]

Presentiamo ora i risultati ottenuti sul classificatore binario con embedding dei dati in ingresso (Tabella 4.2). Per avere un buon confronto con i dati precedenti, sono state eseguite cinque runs per ogni cifra "ostica" con diversi seed: 318, 532, 708, 1375 e 3612. Effettuando un confronto con i risultati

		TPR				**TNR**			
label	#runs	max	min	avg	d.s.	max	min	avg	d.s.
0	5	99.51	93.17	97.95	0.21	100.0	99.62	99.87	0.22
5	5	95.12	86.34	91.51	2.92	99.67	99.13	99.49	0.20
6	5	93.66	72.20	84.39	7.57	99.84	99.30	99.55	0.24
7	5	96.10	92.68	93.95	1.18	99.67	99.51	99.60	0.05
8	5	91.71	83.41	88.19	2.70	99.67	96.86	98.98	1.06

Tabella 4.2: Risultati per il **classificatore con embedding**

precedenti, tutti i valori massimi, sia per TPR che per TNR, sono stati su-

perati; addirittura, solo in un caso il TNR più basso è inferiore al 99%.

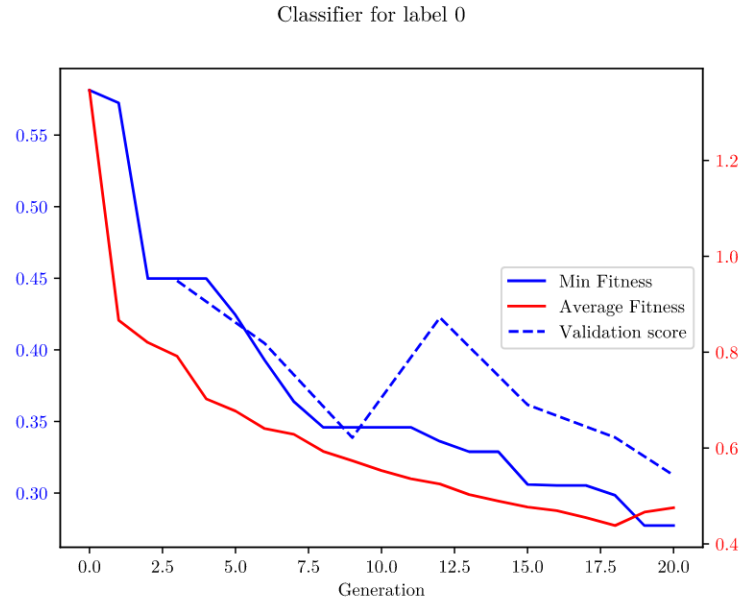
Anche se il training set risulta sbilanciato, è comunque possibile ottenere una stima del valore di accuracy attraverso la **balanced accuracy**: nient'altro che la media aritmetica fra TPR e TNR. In effetti, questo valore può essere considerato come una stima della probabilità che un pattern venga classificato correttamente. Dunque, presentiamo anche i valori di balanced accuracy per i classificatori binari con embedding (Tabella 4.3).

		Balanced Accuracy			
label	#runs	max	min	avg	d.s.
0	5	99.65	96.40	98.91	0.22
5	5	97.29	92.74	95.50	1.53
6	5	96.75	85.75	91.97	3.89
7	5	97.86	96.13	96.78	0.59
8	5	95.61	91.54	93.59	1.41

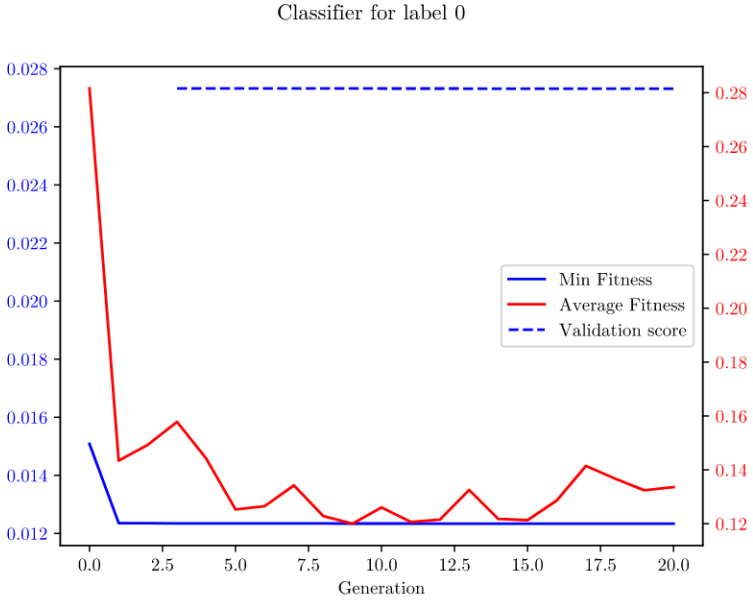
Tabella 4.3: Balanced accuracy per il **classificatore con embedding**

4.3 Grafici di Learning

Per tenere traccia dei dati di apprendimento, oltre a essere visualizzati con dei log, i valori minimi e medi di fitness cacolati sul training set e i valori di fitness sul validation set (validation scores) vengono memorizzati in ordine temporale all'interno di un `logbook`, messo a disposizione da DEAP. Al termine della singola evoluzione, i valori salvati vengono rappresentati su un piano cartesiano. Vediamo un esempio per la cifra 0 (Figura 4.1 e Figura 4.2). Riferita alla scala numerica destra del grafico, in rosso, troviamo la curva della fitness media della popolazione; con linea continua blu è rappresentata la funzione monotona debolmente decrescente bestfit(t), riferita alla scala numerica sinistra, anch'essa in blu; infine, con linea tratteggiata blu e

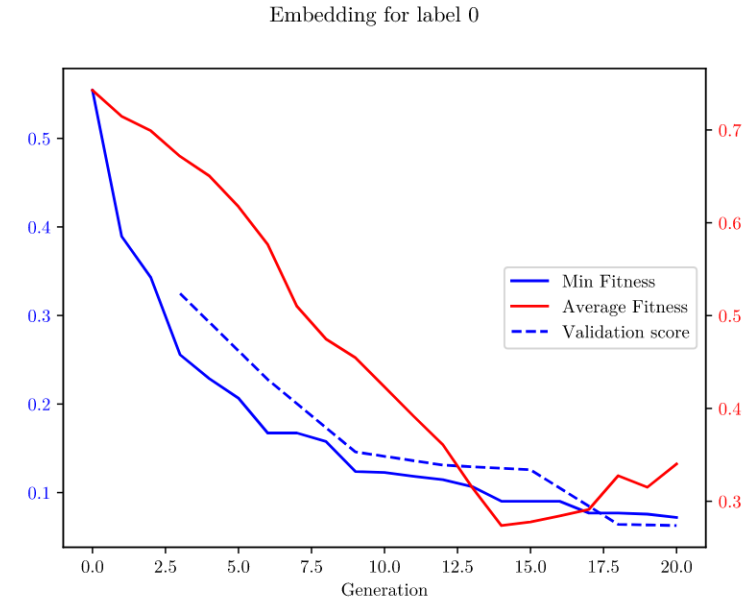


(a) Prima evoluzione

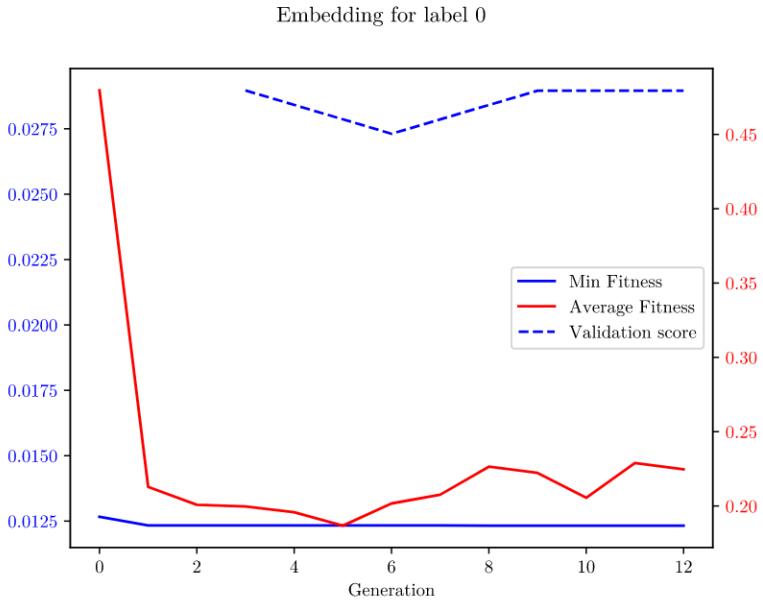


(b) Ultima evoluzione

Figura 4.1: Grafici di apprendimento per il classificatore c_0 (seed 318)



(a) Prima evoluzione



(b) Ultima evoluzione

Figura 4.2: Grafici di apprendimento per l'embedding e_0 (seed 318)

riferita alla stessa scala precedente, troviamo la curva del validation score, sulla quale avviene il conteggio inter-evoluzionale di non miglioramenti per un'eventuale terminazione del processo di coevoluzione. Come si può notare, inizialmente, l'apprendimento si può scorgere facilmente, mentre a mano a mano che ci si sposta verso la fine dell'evoluzione, anche gli stessi grafici preannunciano il termine della stessa, con andamenti del validation score sempre più stazionari; questa proprietà è valida per tutte le evoluzioni (e per ogni label), sia di classificatori che di embedding.

4.4 Classificatore completo

In quest'ultimo paragrafo stimiamo quella che potrebbe essere l'accuratezza di un classificatore completo di cinque classi, corrispondenti alle cinque cifre analizzate, basandoci sui risultati presenti in Tabella 4.2 (i calcoli sono comunque estendibili a qualsiasi classificatore completo). Il classificatore di riferimento è mostrato in Figura 4.3 ed è costituito semplicemente dai cinque classificatori con embedding C_i, con $i \in \{0, 5, 6, 7, 8\}$. Identificando con $r_w \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ l'etichetta del pattern w e

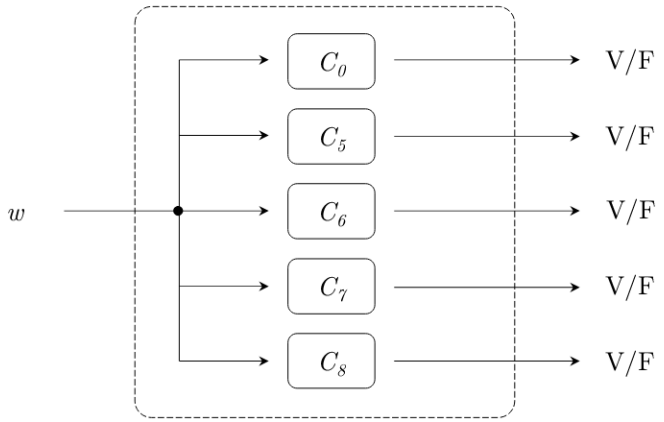


Figura 4.3: Un esempio di classificatore completo (caso senza decisore)

con $C_i(w) \in \{\mathrm{V}, \mathrm{F}\}$ l'uscita del classificatore completo corrispondente alla predizione del classificatore binario C_i, chiamiamo:

- **classificazione corretta** (evento $\mathcal{C}$), una situazione per cui tutte le predizioni in uscita sono corrette, ossia

$$C_i(w) = \begin{cases} \text{V} & \text{se} \quad i = r_w \\ \text{F} & \text{se} \quad i \neq r_w \end{cases} ; \tag{4.5}$$

- **classificazione errata** (evento $\mathcal{E}$), una situazione per cui $r_w = i \in \{0, 5, 6, 7, 8\}$ e $C_i(w) = \text{F}$, oppure $r_w \in \{1, 2, 3, 4, 9\}$ e $\exists j \in \{0, 5, 6, 7, 8\}$ tale che $C_j(w) = \text{V}$;

- **classificazione ambigua** (evento $\mathcal{A}$), una situazione per cui $r_w = i \in \{0, 5, 6, 7, 8\}$ e $C_i(w) = \text{V}$, ma $\exists j \in \{0, 5, 6, 7, 8\} \setminus \{i\}$ tale che $C_j(w) = \text{V}$ (ovviamente, le classi sono **disgiunte**).

Calcoliamo la probabilità di questi tre eventi considerando indipendenti le singole predizioni e impiegando i migliori classificatori C_i (con i valori massimi di balanced accuracy BA_i). Per questi ultimi, riportiamo anche gli effettivi valori di TPR e di TNR in Tabella 4.4.

	Prestazioni		
classificatore C_i	BA_i	TPR_i	TNR_i
C_0	99.65	99.51	99.78
C_5	97.29	95.12	99.46
C_6	96.75	93.66	99.84
C_7	97.86	96.10	99.62
C_8	95.61	91.71	99.51

Tabella 4.4: Classificatori scelti per la composizione di quello completo

- $\mathbb{P}(\mathcal{C}) \equiv \text{Accuracy} = \sum_{j=0}^{9} \mathbb{P}(\mathcal{C}|r_w = j)\mathbb{P}(r_w = j)$, per il Teorema della Probabilità Totale; inoltre, sappiamo che $\mathbb{P}(r_w = j) = 0.1$ (dalla composizione del test set) e che

$$\mathbb{P}(\mathcal{C}|r_w = j) = \begin{cases} TPR_j \cdot \prod_{i \in \{0,5,6,7,8\} \setminus \{j\}} TNR_i & \text{se} \quad j \in \{0, 5, 6, 7, 8\} \\ \prod_{i \in \{0,5,6,7,8\}} TNR_i & \text{se} \quad j \in \{1, 2, 3, 4, 9\} \end{cases} . \tag{4.6}$$

Con qualche conto si giunge al valore di probabilità $\mathbb{P}(\mathcal{C}) \simeq 96.04\%$.

- $\mathbb{P}(\mathcal{E}) = \sum_{j=0}^{9} \mathbb{P}(\mathcal{E}|r_w = j)\mathbb{P}(r_w = j)$, per il Teorema della Probabilità Totale; inoltre, come prima $\mathbb{P}(r_w = j) = 0.1$ (dalla composizione del test set) e

$$\mathbb{P}(\mathcal{E}|r_w = j) = \begin{cases} 1 - TPR_j & \text{se} \quad j \in \{0,5,6,7,8\} \\ 1 - \prod_{i \in \{0,5,6,7,8\}} TNR_i & \text{se} \quad j \in \{1,2,3,4,9\} \end{cases}. \tag{4.7}$$

  Di conseguenza, con qualche conto si arriva al valore di probabilità $\mathbb{P}(\mathcal{E}) \simeq 3.28\%$.

- Siccome i tre eventi sono complementari (quindi anche incompatibili), conoscendo la probabilità dei primi due, quella del terzo è presto determinata: $\mathbb{P}(\mathcal{A}) = 1 - \mathbb{P}(\mathcal{C}) - \mathbb{P}(\mathcal{E}) \simeq 0.68\%$.

Per raggiungere prestazioni più elevate sul classificatore completo, si potrebbe procedere con tecniche di *Ensemble Learning*, aggregando un numero dispari di classificatori binari (o di classificatori completi) indipendenti, per prendere una decisione finale più affidabile attraverso un blocco decisore; tutti aspetti che, comunque, esulano dall'obiettivo di questa tesi.

Conclusioni

Giunti al termine del progetto di tesi, possiamo concludere che gli obiettivi posti in principio sono stati raggiunti: un embedding in ingresso al classificatore consente di aumentare le prestazioni di quest'ultimo. Anche se il miglioramento può sembrare esiguo, bisogna comunque tenere conto che un incremento di prestazioni, partendo da una base già elevata, risulta più complesso da ottenere rispetto ad un miglioramento da una situazione di base più minimale. Nella pratica, ad esempio, un miglioramento di prestazioni da 95% a 99% richiede un maggiore sforzo rispetto a un incremento da 70% a 90%. In generale, il passo con cui si migliora decresce più si è vicini al 100%; da un punto di vista matematico, questa crescita, via via sempre più lenta, può essere rappresentata da una successione di Cauchy.

Se l'obiettivo è migliorare ancora le prestazioni, si potrebbe estendere il progetto con una coevoluzione **bialbero**, dove gli individui sono costituiti non più da uno, ma da ben due alberi. Se si utilizza DEAP, gli individui dovranno esssere comunque costituiti da un unico albero fisico, che, però, sarà formato da due sottoalberi logici distinti. Questi ultimi verranno poi connessi insieme per formare l'albero fisico attraverso il nodo radice `list(·)`, per restituire i risultati dei due alberi logici sottoforma di lista (un solo oggetto). Una volta giunti a questo punto, il passaggio da bialbero a **multialbero** consisterà in una semplice estensione.

Bibliografia

[1] Andries P. Engelbrecht. *Computational Intelligence - An Introduction*, pages 127–185. 2007.

[2] Stefano Cagnoni, Federico Bergenti, Monica Mordonini, and Giovanni Adorni. Evolving binary classifiers through parallel computation of multiple fitness cases. 2005.

[3] Multiinsieme. https://it.wikipedia.org/wiki/Multiinsieme.

[4] Fabrizio De Santis. *Un approccio coevolutivo alla classificazione di pattern basata su Programmazione Genetica*. 2020.

[5] Deap documentation (v1.3.3). https://deap.readthedocs.io/en/master/api/tools.html.