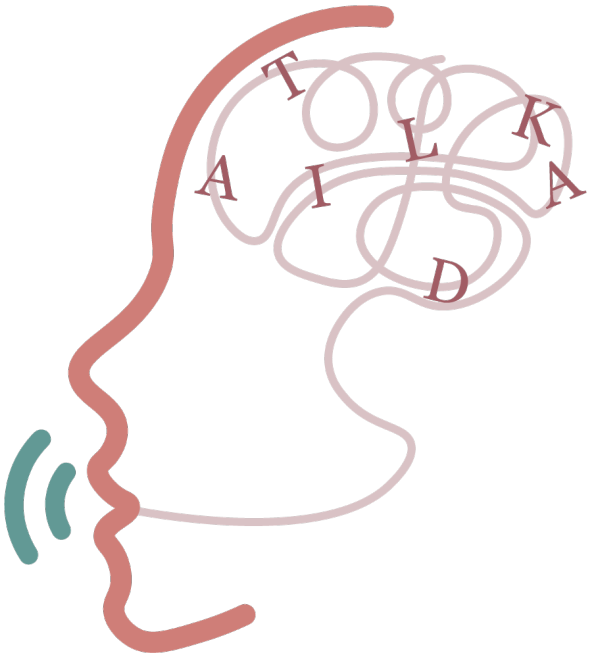

TalkAID - FIA

Algoritmo Genetico per la raccomandazione di esercizi di Logopedia, basati sulle necessità ed esperienza dei pazienti.

Raffaele Monti
Luigi Petrillo
Michele D'Arienzo
Samuele Sparno
Anna Benedetta Salerno

GitHub

26 Gennaio 2024

Indice

1 Sistema Attuale

Il progetto TALKAID rappresenta davvero un passo avanti significativo nel campo della riabilitazione e del supporto alle persone con disabilità del linguaggio. La possibilità di offrire trattamenti completamente *a distanza* e in maniera *asincrona* è un'innovazione che potrebbe aprire nuove opportunità per un numero ancora maggiore di individui affetti da queste patologie.

La registrazione e il monitoraggio dei progressi dei pazienti potrebbe essere complessa e limitata a causa della mancanza di strumenti tecnologici dedicati.

Inoltre i metodi tradizionali potrebbero non essere altamente personalizzati alle esigenze specifiche dei pazienti, poiché potrebbe essere difficile adattarli in modo rapido ed efficiente. Inoltre la necessità delle visite dal vivo può essere un ostacolo per coloro che abitano a grandi distanze dai centri di logopedia oppure per i lavoratori.

La componente di Intelligenza Artificiale è ritenuta fondamentale ai nostri obiettivi siccome aggiunge un livello di personalizzazione e adattabilità, permettendo ai pazienti di ricevere esercizi mirati in base al loro necessità ed esperienza. Questo non solo rende il trattamento più efficace, ma anche più agevole per i logopedisti, in quanto saranno aiutati dall'IA nella scelta degli esercizi migliori. I pazienti verranno incoraggiati a perseguire con impegno il percorso di miglioramento attraverso le loro statistiche.

2 Sistema Proposto

Lo scopo del nostro progetto è quello di realizzare un agente intelligente che possa:

- Consigliare un insieme di esercizi mirato per il paziente, basandosi sulle sue reali necessità;
- Migliorare il sistema di consigli nel tempo, facendolo evolvere sulla base delle esecuzioni degli esercizi fatti dal paziente, considerando fattori come valutazione, feedback etc.

Sarà quindi in grado di generare insiemi di esercizi anche per pazienti privi di esperienza, e di andare pari passo con l'andamento dei loro miglioramenti.

3 Specifica PEAS

Di seguito abbiamo elencato la specifica P.E.A.S. dell'agente intelligente.

3.1 Performance:

Le prestazioni dell'agente sono valutate attraverso le seguenti misure:

- La sua capacità di consigliare esercizi il più mirati possibile per il paziente in base alla patologia;
- Il punteggio dell'esercizio che il paziente ha effettuato su quell'esercizio.

3.2 Environment :

L'ambiente è:

- *Completamente osservabile*, in quanto si ha accesso a tutte le informazioni relative ai pazienti, in particolare le patologie, gli esercizi svolti e le informazioni sull'esecuzione degli esercizi come valutazione, feedback etc;
- *Non deterministico*, in quanto lo stato dell'ambiente cambia indipendentemente dalle azioni dell'agente, come ad esempio cambiamento delle gravità delle patologie o esecuzioni di esercizi raccomandati dal logopedista curante;
- *Sequenziale*, in quanto le esercitazioni effettuate dai pazienti e le scelte del logopedista influenzano le decisioni future dell'agente;
- *Dinamico*, in quanto nel corso delle elaborazioni dell'agente, un paziente potrebbe svolgere un esercizio, cambiando in tal modo le sue esigenze;
- *Discreto*, il numero di percezioni dell'agente è limitato in quanto ha un numero discreto di patologie, esercizi, pazienti, azioni e percezioni possibili;
- *Singolo agente*, in quanto per il singolo paziente sarà l'unico agente che opera in questo ambiente è quello in oggetto.

3.3 Actuators:

- Salvataggio della lista di esercizi in un database, che sarà poi esposto in forma tabellare al logopedista curante nella sua homepage, affinché quest'ultimo possa approvare o meno, per ogni paziente, i risultati dell'agente.

3.4 Sensors:

- L'agente accede ad un dataset, il quale è un riadattamento del database attualmente utilizzato dal sistema principale di TalkAID, eliminando gli attributi non necessari e aggiungendo gli attributi gravità di lettura e scrittura, necessari al fine di poter avere una più completa visualizzazione delle necessità reali del paziente. (Nella versione definitiva del sistema principale, gli attributi aggiunti per gravità di lettura e scrittura esisterebbero e verrebbero gestiti dal logopedista curante);
- Il dataset comprende informazioni e dati su:
  - Le condizioni trattate dal sistema principale;
  - Le esecuzioni di esercizi, che comprendono valutazione, feedback e data di completamento da parte dell'utente;
  - Il glossario degli esercizi, con informazioni su difficoltà, target e tipologia;
  - La gravità delle condizioni degli utenti.

4 Soluzione proposta

Date le nostre necessità, abbiamo potuto constatare che ciò di cui abbiamo bisogno è un algoritmo *Genetico* di ottimizzazione.

Nonostante un'attenta valutazione di vari algoritmi, tra i quali spicca l'utilizzo di tecniche come la *segmentazione degli utenti* o il *collaborative filtering*, in particolare il *clustering* potrebbe aiutare a limitare quali esercizi consigliare in base alle patologie del paziente. Purtroppo, al momento non esistono dataset pertinenti per la nostra valutazione, quindi disponiamo di un insieme di dati limitato e poco realistico, che non permette a un ipotetico algoritmo di apprendimento di effettuare un training ottimale.

Abbiamo quindi optato per un algoritmo di ricerca locale genetico, poiché è in grado di individuare un punto ottimo tra le diverse alternative, producendo soluzioni sempre migliori rispetto a una funzione obiettivo, anche in assenza di un dataset molto esteso. È importante notare che ciò non garantisce l'ottimalità, dato che solitamente produce soluzioni sub-ottimali. Proprio per questo motivo, affidiamo il lavoro di supervisione al logopedista.

Il nostro obiettivo è ottenere una lista di esercizi per ciascun paziente che possa soddisfare le sue specifiche esigenze. Questa lista sarà poi presa in considerazione dal logopedista. Di conseguenza, potremmo dire che ogni popolazione sarà associata ad un determinato paziente, i cui individui saranno un possibile insieme di esercizi raccomandati.

5 Raccolta e sviluppo del dataset

Avendo la necessità di utilizzare un dataset sul quale il nostro modello avrebbe dovuto estrapolare le informazioni riguardanti gli esercizi e i pazienti, la sfida consisteva in due opzioni:

- Cercare un dataset pre-esistente al fine di avere molte informazioni, e al massimo effettuare data cleaning e adeguarlo a quello che ci serve.
- Creare un dataset, formulando gli esercizi, aggiungendo i pazienti e creando le varie valutazioni per ogni esercizio.

Purtroppo la ricerca di un dataset pre-esistente non è stato proficuo data l'unicità delle nostre richieste; infatti, non abbiamo trovato dataset che riguardassero nel dettaglio la valutazione di esercizi di logopedia.

Di conseguenza, abbiamo deciso di creare un nostro dataset, utilizzando un database in MySQL e generando le varie tipologie di esercizi di nostra iniziativa, ottenendo *84 esercizi*, divisi per *7 diverse tipologie*, e una selezione di *12 patologie* tra le più comuni e con difficoltà il più possibile incentrate principalmente sulla Scrittura o Lettura. La generazione dei pazienti e la generazione delle valutazioni degli esercizi sono state prodotte sinteticamente mediante l'utilizzo di valori casuali.

- Di seguito vengono mostrati i vari codici in SQL realizzati per poter popolare il nostro database.

5.1 Popolazione del DataBase

```
    INSERT INTO exercise (ID_user, ID_exercise, InsertionDate, CompletionDate,
    Evaluation, Feedback)
        VALUES
        (
        FLOOR(904 + RAND() * (1003 - 904 + 1)), -- Random user ID
        FLOOR(1 + RAND() * (84 - 1 + 1)), -- Random exercise ID
        DATE_SUB(NOW(), INTERVAL FLOOR(RAND() * 365) DAY), -- Random date
        DATE_SUB(NOW(), INTERVAL FLOOR(RAND() * 365) DAY), -- Random completion date
        FLOOR(RAND() * (100 - 0 + 1)), -- Random evaluation between 0 and 100
        CASE
            WHEN RAND() <= 0.33 THEN -1
            WHEN RAND() <= 0.66 THEN 0
            ELSE 1
        END -- Random feedback (-1, 0, or 1)
        );

```

Listing 1: Generazione Casuale esecuzione esercizio

```
        DELIMITER //

        CREATE PROCEDURE GenerateUsers()
        BEGIN
            DECLARE i INT DEFAULT 1;
            DECLARE j INT;

            WHILE i <= 10 DO -- For each therapist user
                SET j = 1;
                WHILE j <= 10 DO -- Create 10 patient users
                    INSERT INTO user (ID_Therapist) VALUES (i);
                    SET j = j + 1;
                END WHILE;
                SET i = i + 1;
            END WHILE;
        END //

        DELIMITER ;

        CALL GenerateUsers();

```

Listing 2: Generazione degli utenti

```
INSERT INTO patientcondition (ID_condition, ID_patient, Severity, WritingSeverity,
 ReadingSeverity)
SELECT
    FLOOR(1 + RAND() * 12) as ID_condition,  -- Random condition ID (1 to 12)
    u.ID as ID_patient,
    1 as Severity,
    FLOOR(1 + RAND() * 10) as WritingSeverity, -- Random Severity
    FLOOR(1 + RAND() * 10) as ReadingSeverity -- Random Severity
FROM
    user u
WHERE
    u.ID_Therapist != 0;
```

Listing 3: Generazione delle Writing e Reading severity (tutti i pazienti)

```
INSERT INTO patientcondition (ID_condition, ID_patient, Severity, WritingSeverity,
 ReadingSeverity)
SELECT
    FLOOR(1 + RAND() * 12) as ID_condition,  -- Random condition ID (1 to 12)
    u.ID as ID_patient,
    1 as Severity,
    FLOOR(1 + RAND() * 10) as WritingSeverity, -- Random Severity
    FLOOR(1 + RAND() * 10) as ReadingSeverity -- Random Severity
FROM
    user u
WHERE
    u.ID_Therapist != 0
ORDER BY
    RAND()  -- Randomly order the users and pick the first one
LIMIT 30;

```

Listing 4: Generazione delle Writing e Reading severity (solo per 30 pazienti)

```
UPDATE patientcondition pc
SET
    pc.Severity =
        LEAST(GREATEST(pc.WritingSeverity, pc.ReadingSeverity) +
        IF(RAND() < 0.5, -1, 1), 10) -- Random Severity
WHERE
    pc.Severity = 1;

```

Listing 5: Generazione della General severity (tutti i pazienti)

La necessità di generare in due modi diversi le patologie degli utenti (una volta per tutti i pazienti, una volta solo per 30 pazienti) , è dovuta alla concreta possibilità di pazienti con più di una patologia, come ad esempio *Dislessia* e *Disgrafia*.

5.2 Analisi del DataBase

In questa sezione, illustriamo il processo di ideazione delle tabelle, riportando il loro nome, descrivendone la funzione e indicando gli attributi contenuti.

- *condition* - Contiene tutte le informazioni sulle patologie trattate dal sistema, importante per le relazioni tra esercizi e patologie.

Tabella 1: condition Table

Attributo	**Descrizione**
ID_condition	id della patologia
Description	serve per capire cosa provoca la patologia, non è un dato rilevante per il nostro agente
Name	nome della patologia

- *exercise* - Contiene tutti gli esercizi assegnati e/o svolti, utile per poter tenere traccia dell'andamento dei vari pazienti.

Tabella 2: exercise Table

Attributo	**Descrizione**
ID_user	id del paziente associato all'esercizio
ID_exercise	id dell'esercizio che ha svolto il paziente
InsertionDate	data di assegnazione dell'esercizio da parte del logopedista
CompletionDate	data di completamento dell'esercizio
Evaluation	che punteggio il paziente ha raggiunto
Feedback	feedback per mostrare se l'esercizio è piaciuto o meno

- *exercise_glossary* - Contiene le varie informazioni specifiche degli esercizi, interessante poiché ci permette di poter trovare il target specifico di un determinato esercizio, o la sua difficoltà.

Tabella 3: execise_glossary Table

Attributo	**Descrizione**
ID_exercise	id dell'esercizio
ExerciseName	nome dell'esercizio
ExerciseDescription	breve descrizione dell'esercizio
Type	tipo dell'esercizio, può essere un esercizio di lettura testo, associare immagini, trovare la frase corretta etc...
Difficulty	grado di difficoltà dell'esercizio
Target	il target è la patologia alla quale questo esercizio è mirato

- *patient_condition* - Contiene le informazioni riguardo la patologia che affligge il paziente, in base a questa il nostro agente potrà selezionare esercizi specifici per il paziente.

Tabella 4: patient_condition Table

Attributo	**Descrizione**
ID_condition	id della patologia affetta dal paziente
ID_patient	id del paziente riferito
Severity	gravità della patologia del paziente
WritingSeverity	gravità della condizione del paziente nello scrivere
ReadingSeverity	gravità della condizione del paziente nel leggere

5.3 Interazioni con il DataBase

Le interazioni effettuate dal nostro agente sul database sono esclusivamente operazioni di estrapolazione dati. Le funzioni create per le estrazioni degli esercizi, e relative informazioni sulla loro esecuzione da parte del paziente, sono:

- **select_exercises_not_done**: Questa funzione permette di ottenere tutti gli esercizi non ancora svolti dal paziente.
- **select_done_exercises**: Questa funzione permette il recupero egli esercizi svolti dal paziente.
- **select_random_exercise**: Questa funzione recupera casualmente alcuni esercizi e, nel caso in cui vi siano esercizi precedentemente completati dal paziente, restituisce anche i relativi risultati.

Per quanto riguarda le estrazioni dei dati inerenti ai pazienti e alle loro patologie:

- **informationUser**: Questa funzione permette il recupero di informazioni inerenti alle varie patologie che il paziente è affetto con i livelli di gravità per ognuna di essa.

Data la lunghezza del codice, sono stati inseriti solo i nomi delle funzioni. Per la completa documentazione, fare riferimento alla pagina GitHub.

6 Studio della funzione di fitness

Da qui cominciamo a descrivere lo studio che è stato effettuato per poter sviluppare la funzione di fitness. Essa è basata su varie informazioni inerenti al paziente, agli esercizi e la loro ultima esecuzione, se esiste.

Più precisamente, le seguenti calcolano la fitness unicamente dall'esercizio e necessità del paziente:

- **difficultBasedFitness**: Restituisce un valore sulla base della difficoltà del singolo esercizio. Per prima cosa otteniamo l'esperienza del paziente andando a calcolare la *mediana* della difficoltà degli ultimi 50 esercizi svolti. Calcoliamo poi la media delle General Severity delle conditions. Quando più è alta questa media, più lenta deve essere la curva di crescita della difficoltà, rispetto all'esperienza. Se la difficoltà rientra nella curva di crescita, restituisce 100, altrimenti un valore sempre più basso quanto si discosta. (Nel caso di paziente con zero esperienza, il valore di experience è uguale a 0).

```
    def difficultBasedFitness(e: Exercise, u: User) -> float:
        difficulty = e.getExerciseDifficulty()
        severity = fu.getMeanSeverity(u)
        experience = fu.getExperience(u)
        maxD, minD = fu.getMaxMinExperience(u)

        r = maxD - minD
        if r == 0:
            r = 1

        distance = abs(difficulty - experience)
        slope = fu.getSlope(severity)

        if distance < difficulty <= distance + slope:
            value = 100
        else:
            value = 100 - ((distance / r) * 100)
        return value
```

- **severityBasedFitness**: Restituisce un valore da 0 a 100 sulla base della tipologia (Lettura o Scrittura) dell'esercizio e la gravità del paziente in tale ambito.

```
    def severityBasedFitness(e: Exercise, u: User) -> float:
    if e.getExerciseType() == "READTEXT" or e.getExerciseType() == "TEXTTOIMAGES"
    or e.getExerciseType() == "READIMAGES":
        value = fu.getMeanReadingSeverity(u)
    else:
        value = fu.getMeanWritingSeverity(u)

    return value * 10
```

- **targetBasedFitness**: Restituisce un valore che va da 0 a 100 in base all'adeguatezza dell'esercizio per le patologie affette del paziente, tramite le informazioni contenute nel Target dell'esercizio.

```
    def targetBasedFitness(e: Exercise, u: User) -> float:
    rightTarget = 0
    for key in u.getConditions().keys():
        if key in e.getExerciseTarger():
            rightTarget += 1

    return (rightTarget / len(e.getExerciseTarger().split(","))) * 100
```

Le seguenti invece, calcolano la fitness utilizzando le informazioni inerenti all'ultima esecuzione dell'esercizio da parte del paziente. Se non è mai stato eseguito, restituiscono il massimo della fitness:

- **evaluateBasedFitness**: La funzione restituisce un valore che va da 0 a 100 ottenuto sottraendo la votazione massima che si può avere da un'esecuzione di un esercizio, e la votazione che il paziente ha avuto su quell'esercizio.

```
    def evaluateBasedFitness(e: Exercise) -> float:
    if e.getLastEvaluation() is not None:
        value = 100 - e.getLastEvaluation()
    else:
        value = 100
    return value
```

- **dateBasedFitness**: Restituisce un valore che va da 0 a 100 sulla base della quantità di tempo passata dalla data odierna all'ultima data di completamento del esercizio.

```
    def dateBasedFitness(e: Exercise, maxDays: int) -> float:
    value = 100
    if e.getLastCompletionDate() is not None:
        completionDate = datetime.strptime(e.getLastCompletionDate(), "%Y-%m-%d")
        data_odierna = datetime.now()
        giorni_passati = (data_odierna - completionDate).days
        if giorni_passati < maxDays:
            value = fu.getDaysValue(giorni_passati, maxDays)
    return value
```

- **feedbackBasedFitness**: Restituisce un valore in base al feedback del paziente, quindi controlla se il paziente ha espresso un giudizio positivo (100) o uno negativo (0).

```
    def feedbackBasedFitness(e: Exercise) -> float:
    if e.getLastFeedback() is not None:
        if e.getLastFeedback() > 0:
            return 100
        else:
            return 0
    else:
        return 100
```

- **last50Fitness**: Restituisce un valore in base alla posizione negli ultimi 50 esercizi fatti dall'paziente. Nel caso in cui non è presente, restituisce 100, altrimenti un valore sempre più basso quanto più alta è la posizione.

```
    def last50Fitness(e: Exercise, u: User) -> float:
    if e.getExerciseID() in u.getExercises().keys():
        position = 0
        for index, ID in enumerate(u.getExercises().keys()):
            if ID == e.getExerciseID():
                position = index
                break
        return fu.getPositionValue(position)
    else:
        return 100
```

Tutti questi valori sono di varia importanza e origine, per questo ogni funzione nel calcolo del valore fitness è moltiplicato con una costante di importanza dedicata.

```
    def fitness(ex: Exercise, u: User) -> float:
    f1, f2, f3, f4, f5, f6, f7 = F1, F2, F3, F4, F5, F6, F7

    somma = (f1 * evaluateBasedFitness(ex)) + (f2 * dateBasedFitness(ex, MDAYS)) + (f3
     * difficultBasedFitness(ex, u))
    somma += (f4 * severityBasedFitness(ex, u)) + (f5 * targetBasedFitness(ex, u))
    somma += (f6 * feedbackBasedFitness(ex)) + (f7 * last50Fitness(ex, u))

    # somma = (f3 * difficultBasedFitness(ex, u))
    # somma += (f4 * severityBasedFitness(ex, u))
    # somma += (f5 * targetBasedFitness(ex, u)) <- Only exercise info, ignores
    executions from user

    return somma
```

Tale variabile è modificabile in modo tale che il logopedista può decidere se considerare più un determinante rispetto ad un'altra in base alle proprie esigenze ed esperienza. Inoltre ci sembrava inopportuno forzare l'utilizzo delle nostre personali costanti, data la nostra incompetenza nel settore della terapia logopedica rispetto all'effettivo terapista.

7 Sviluppo del nostro algoritmo genetico

Vista la dimensione del nostro gruppo abbiamo anche deciso di implementare e provare diversi algoritmi per le diverse fasi dell'algoritmo Genetico.

L'algoritmo genetico è suddiviso nelle fasi di Selezione, Crossover, Mutazione e valutazione.

7.1 Scelte progettuali

- La ***size della popolazione*** è fissa, ovviamente può essere modificata, ma una volta avviato l'algoritmo la dimensione rimane tale. Dato che abbiamo implementato sia crossover con 2 che con 3 genitori, l'unica accortezza è scegliere una dimensione tale che possa permettere il crossover scelto (esempio, se la popolazione è dispari e si prova a eseguire un crossover con 2 genitori, potrebbe portare l'agente ad un arresto o un comportamento imprevisto);
- La ***size mating pool*** è grande quanto il numero di individui. Il possibile rallentamento che comporta non è un fattore rilevante, infatti l'algoritmo potrebbe aggirare tale problema venendo eseguito sempre in orari in cui il tempo di esecuzione non è un problema (quindi quando il paziente ha già esercizi da fare / quando il logopedista ha già degli esercizi da valutare, oppure in orari notturni in cui sia il paziente che il logopedista non utilizzano il sistema principale);
  - Inoltre, è stato implementato con funzionalità di multiprocessing, quindi può eseguire in parallelo più utenti. Le performance temporali non sono dunque un problema rilevante per il reale contesto di utilizzo;
- La ***probabilità di crossover*** dipende dal tipo di crossover scelto. Infatti abbiamo implementato crossover dove è certo lo scambio dei geni, e crossover dove si può specificare la probabilità;
- La ***probabilità di mutazione*** è dichiarabile prima dell'avvio dell'algoritmo;
- L' ***inizializzazione*** avviene selezionando casualmente degli esercizi dal dataset e, nel caso fossero già stati eseguiti almeno una volta, conterrebbero le informazioni dell'ultima esecuzione ;
- Gli ***individui*** sono istanze di una classe, essa contiene il valore di fitness dell'individuo e una lista di istanze di esercizi (più informazioni alla sezione 7.2);
  - Ogni ***gene*** è anch'esso un'istanza di una classe, che al suo interno contiene tutte le informazioni dell'esercizio generale, sia le eventuali informazioni inerenti all'ultima esecuzione da parte del paziente (più informazioni alla sezione 7.2).
- Per la ***Selezione*** abbiamo sviluppato sia l'algoritmo *Roulette Wheel* (ma abbiamo notato una tendenza verso la convergenza prematura), che *Rank Selection*, il quale risolve il problema riscontrato dalla Roulette Wheel. Inoltre abbiamo anche implementato una *Random Selection*.
- Per il ***Crossover*** ne abbiamo sviluppati diversi, tutti funzionanti sia per 2 che per 3 genitori:
  - *nPoint* e *nPointReverse* dove è possibile specificare il numero di punti per la divisione degli individui. nPointReverse si differenzia da nPoint poiché inizia con uno scambio, in questo modo anche le posizioni iniziali vengono scambiate per aumentare la casualità della scelta;
    - Entrambe sono state implementate con e senza probabilità di crossover.
  - Inoltre è stato implementato anche *uniformCrossover*, in cui i singoli geni, tramite la % di crossover, hanno la probabilità di venire scelti casualmente tra geni dei genitori.

- Come ***Mutazione*** abbiamo implementato, con probabilità di mutazione a nostra scelta:
  - *Mutazione del singolo gene*, in cui viene selezionato casualmente un gene in un unico individuo e viene rimpiazzato con un nuovo individuo prelevato dal dataset;
  - *Mutazione di un intero Individuo*, in cui viene selezionato:
    * Un *individuo a caso* e completamente scartato e ricreato;
    * Il *peggior individuo*, anch'esso completamente scartato e ricreato.
- La ***Stopping condition*** ha due implementazioni, che si possono usare singolarmente o in coppia:
  - *Stop dopo x generazioni*, in cui l'algoritmo si ferma dopo aver eseguito x generazioni;
  - *Stop se il valore fitness aumenta per **x** generazioni di **y%***, in cui l'algoritmo controlla ad ogni generazione, l'aumentare del valore di fitness, e nel caso in cui non aumenta di una percentuale di nostra decisione, si interrompe e restituisce l'ultima generazione.

7.2 Implementazione delle classi

L'algoritmo GA esegue le varie generazioni e valutazioni su istanze della classe *Population*.
Tale classe è strutturata dai seguenti attributi, ognuno dotato di funzioni di modifica e visualizzazione. Più precisamente sono:

- **user**: contiene l'istanza della classe User, dotata delle informazioni inerenti al paziente;
- **entireFitness**: valore fitness dell'intera Popolation;
- **currGen**: valore attuale della generazione;
- **individuals**: una lista di istanze della classe Individual, ognuna contenente una lista di esercizi.

```
1     class Population:
2     def __init__(self, u: User, *args: Individual):
3
4         self._user = u
5         self._entireFitness = 0
6         self._currGen = 0
7         self._individuals = list()
8         for i in args:
9             self._individuals.append(i)
```

La classe *User* è caratterizzato da:

- **ID**: indicante l'id identificativo del paziente;
- **conditions**: dizionario delle patologie del paziente, in cui la chiave è il nome della patologia e il valore è una tupla delle tre gravità (Generale, Scrittura, Lettura);
- **exercises**: dizionario degli ultimi 50 esercizi eseguiti dal paziente, in cui la chiave è l'ID dell'esercizio e il valore è l'istanza della classe Exercise inerente a quell'esercizio.

```
1     class User:
2     def __init__(self, ID: int, conditions: dict, exercises: dict):
3
4         self._ID = ID
5         self._conditions = conditions
6         self._exercises = exercises
```

Gli individui sono istanze della classe *Instance*, contenente un set di esercizi. Ogni esercizio è un'istanza della classe *Exercise*, ognuna rappresenta un gene.

La classe *Exercise* è caratterizzata da:

- **fitnessValue**: valore di fitness calcolato del singolo esercizio;
- **generation**: indica in quale generazione l'esercizio è stato prelevato dal dataset;
- **ID**: indica l'id identificativo dell'esercizio;
- **exerciseDifficulty**: la difficoltà dell'esercizio;
- **target**: indica a quali patologie è meglio dedicata l'esercizio;
- **extType**: indica il tipo di esercizio tra i 7 sviluppati;
- **lastEvaluation**: indicante l'ultima valutazione dell'esercizio se questo è stato già eseguito;
- **lasCompletionDate**: indicante l'ultima data in cui è stato completato l'esercizio;
- **lastFeedback**: indicante il feedback lasciato dal paziente all'esercizio.

```
    class Exercise:
    def __init__(self, ID: int, exerciseDifficulty: int, target: str, exType: str,
                 lastEvaluation: int | None, lastCompletionDate: str | None,
    lastFeedback: int | None):
        self._fitnessValue = 0
        self._generation = 0
        self._exerciseID = ID
        self._exerciseDifficulty = exerciseDifficulty
        self._target = target
        self._exType = exType
        self._lastEvaluation = lastEvaluation
        self._lastCompletionDate = lastCompletionDate
        self._lastFeedback = lastFeedback
```

Infine, più precisamente la classe *Individual* è caratterizzato da:

- **individualFitness**: valore fitness del singolo set di esercizi;
- **exercises**: la lista delle istanze di Exercise che compongono il set di esercizi.

```
    class Individual:
    def __init__(self, *args: Exercise):
        self._individualFitness = 0
        self._exercises = list()
        for ex in args:
            self._exercises.append(ex)
```

7.3 Selezione

Nella fase di Selezione, viene anche attuata la fase di valutazione, poiché abbiamo implementato metodi che si basano sui valori di fitness. Nello specifico, abbiamo sviluppato 3 tipi diversi di algoritmi per la selezione degli individui da inserire nella mating pool:

- **Roulette Wheel**: Come nei Casinò, abbiamo una "ruota" in cui ogni porzione è proporzionale al valore di fitness dell'individuo. Facendo girare la ruota, si selezionano gli individui vincenti.

```
def rouletteWheel(population: Population):
    user = population.getUser()
    totalFitness = 0
    for i in population.getIndividuals():
        evaluate(i, user)
        totalFitness += i.fitness()

    newIndividuals = list()
    while len(newIndividuals) < len(population):
        value = random.random() * totalFitness
        comulativeFitness = 0
        for i in population.getIndividuals():
            comulativeFitness += i.fitness()
            if comulativeFitness >= value:
                newIndividuals.append(Individual(*i.getList()))
                break
    return newIndividuals
```

- **Rank Selection**: Questo metodo sfrutta il rango di ciascun individuo per suddividere una "torta" di selezione in partizioni proporzionali al rango dell'elemento. Il rango varia dal più basso al più alto, con la percentuale di essere selezionato proporzionata al rango. Riesce a mitigare i difetti della Roulette Wheel, poiché la percentuale è legata agli elementi e non unicamente al valore di fitness.
  - Per individuare i ranghi degli elementi, viene utilizzata la funzione *findRank*.

```
def rankSelection(population: Population):
    user = population.getUser()
    for i in range(len(population)):
        evaluate(population[i], user)

    sortedIndividuals = sorted(population.getIndividuals(), key=lambda ind: ind.
    fitness())

    newPopolation = list()
    size = len(sortedIndividuals)
    subdivision = (size * (size+1))/2

    while len(newPopolation) < len(population):
        newPopolation.append(su.find_rank(sortedIndividuals, subdivision))
    return sortedIndividuals
```

```
def find_rank(individuals: list[Individual], subdivision: float) -> Individual:
    num = random.randint(1, 100)
    prec = 0
    for i in range(1, len(individuals)):
        if prec < num <= (100/subdivision)*i:
            return Individual(*individuals[i].getList())
        else:
            prec = (100/subdivision)*i
```

- **Random Selection**: La selezione degli individui attuata è completamente casuale; in effetti, l'assenza di una logica sottostante può generare dati del tutto inefficienti o dati straordinariamente eccezionali. Tutto è lasciato al caso.

```
def randomSelection(population: Population):
    newP = []
    while len(newP) < len(population):

        newP.append(Individual(*random.choice(population)).getList())
    return newP

```

7.4 Crossover

Questa fase si occupa di generare nuovi elementi a partire da quelli selezionati. I metodi sono categorizzati in base al tipo, poiché concettualmente eseguono lo stesso lavoro, ma con un numero di individui diversi. Per questo motivo, vengono mostrati solo i metodi con 3 genitori, poiché possono risultare più interessanti.

7.4.1 Metodologie di Crossover applicate a 3 elementi

- **nPoint**: Spezza le liste degli individui selezionati in n punti. Ciascuna lista viene quindi suddivisa in sottoliste più piccole, le quali saranno scambiate con le altre analoghe sottoliste nei punti di divisione.

```
def nPoint(i1: Individual, i2: Individual, i3: Individual, n: int) -> tuple[Individual
    , Individual, Individual]:
    if len(i1) != len(i2) != len(i3):
        raise ValueError("Invalid individual length! They have to be the same.")
    elif n > len(i1):
        raise ValueError("Invalid n value, must be less than the lenght of the
    individual!")

    dividedl1, dividedl2, dividedl3 = divide(i1.getList(), n), divide(i2.getList(), n)
    , divide(i3.getList(), n)

    newl1 = list()
    newl2 = list()
    newl3 = list()

    for i, triplet in enumerate(zip(dividedl1, dividedl2, dividedl3)):
        if i % 3 == 0:
            newl1.extend(triplet[0])
            newl2.extend(triplet[1])
            newl3.extend(triplet[2])
        elif i % 3 == 1:
            newl1.extend(triplet[2])
            newl2.extend(triplet[0])
            newl3.extend(triplet[1])
        elif i % 3 == 2:
            newl1.extend(triplet[1])
            newl2.extend(triplet[2])
            newl3.extend(triplet[0])

    i1.setList(newl1)
    i2.setList(newl2)
    i3.setList(newl3)

    return i1, i2, i3
```

- **nPointReverse**: A differenza della precedente implementazione, questa versione esegue lo scambio già dal primo elemento, affinché possa risultare più casuale un possibile ripescaggio dello stesso individuo.

```
def nPointReverse(i1: Individual, i2: Individual, i3: Individual, n: int) -> tuple[Individual, Individual, Individual]:
    if len(i1) != len(i2) != len(i3):
        raise ValueError("Invalid individual length! They have to be the same.")
    elif n > len(i1):
        raise ValueError("Invalid n value, must be less than the lenght of the individual!")

    dividedl1, dividedl2, dividedl3 = divide(i1.getList(), n), divide(i2.getList(), n), divide(i3.getList(), n)

    newl1 = []
    newl2 = []
    newl3 = []

    for i, triplet in enumerate(zip(dividedl1, dividedl2, dividedl3)):
        if i % 3 == 0:
            newl1.extend(triplet[1])
            newl2.extend(triplet[2])
            newl3.extend(triplet[0])
        elif i % 3 == 1:
            newl1.extend(triplet[0])
            newl2.extend(triplet[1])
            newl3.extend(triplet[2])
        elif i % 3 == 2:
            newl1.extend(triplet[2])
            newl2.extend(triplet[0])
            newl3.extend(triplet[1])

    i1.setList(newl1)
    i2.setList(newl2)
    i3.setList(newl3)

    return i1, i2, i3

```

- **nPointRandom**: Questa versione di nPoint aggiunge la possibilità di dichiarare la *probabilità di crossover*(precedentemente avveniva sempre).

```
def nPointRandom(i1: Individual, i2: Individual, i3: Individual, n: int,
    crossoverProbability: float) -> tuple[Individual, Individual, Individual]:
    if len(i1) != len(i2) != len(i3):
        raise ValueError("Invalid individual length! They have to be the same.")
    elif n > len(i1):
        raise ValueError("Invalid n value, must be less than the lenght of the
    individual!")
    elif crossoverProbability < 0 or crossoverProbability > 1:
        raise ValueError("Invalid crossoverProbability! Must be between 0 and 1.")

    dividedI1, dividedI2, dividedI3 = divide(i1.getList(), n), divide(i2.getList(), n)
    , divide(i3.getList(), n)

    newI1 = []
    newI2 = []
    newI3 = []

    for triplet in zip(dividedI1, dividedI2, dividedI3):
        if random() < crossoverProbability:
            i = randint(0, 2)
            if i % 3 == 0:
                newI1.extend(triplet[0])
                newI2.extend(triplet[1])
                newI3.extend(triplet[2])
            elif i % 3 == 1:
                newI1.extend(triplet[2])
                newI2.extend(triplet[0])
                newI3.extend(triplet[1])
            elif i % 3 == 2:
                newI1.extend(triplet[1])
                newI2.extend(triplet[2])
                newI3.extend(triplet[0])
        else:
            newI1.extend(triplet[0])
            newI2.extend(triplet[1])
            newI3.extend(triplet[2])

    i1.setList(newI1)
    i2.setList(newI2)
    i3.setList(newI3)

    return i1, i2, i3

```

- **uniformCrossover**: In questo caso, ogni gene ha la possibilità di venir casualmente rimpiazzato con un gene degli altri genitori.

```
def uniformCrossover(i1: Individual, i2: Individual, i3: Individual,
    crossoverProbability: float) -> tuple[Individual, Individual, Individual]:
    if len(i1) != len(i2) != len(i3):
        raise ValueError("Invalid individual length! They have to be the same.")
    elif crossoverProbability < 0 or crossoverProbability > 1:
        raise ValueError("Invalid crossoverProbability! Must be between 0 and 1.")

    dividedI1 = divide(i1.getList(), len(i1))
    dividedI2 = divide(i2.getList(), len(i2))
    dividedI3 = divide(i3.getList(), len(i3))

    newI1 = []
    newI2 = []
    newI3 = []

    for triplet in zip(dividedI1, dividedI2, dividedI3):
        if random() < crossoverProbability:
            if random() < 0.5:
                newI1.extend(triplet[1])
            else:
                newI1.extend(triplet[2])
        else:
            newI1.extend(triplet[0])

        if random() < crossoverProbability:
            if random() < 0.5:
                newI2.extend(triplet[2])
            else:
                newI2.extend(triplet[0])
        else:
            newI2.extend(triplet[1])

        if random() < crossoverProbability:
            if random() < 0.5:
                newI3.extend(triplet[0])
            else:
                newI1.extend(triplet[1])
        else:
            newI1.extend(triplet[2])

    i1.setList(newI1)
    i2.setList(newI2)
    i3.setList(newI3)

    return i1, i2, i3

```

La funzione *divide* si occupa, in base a n, della divisione delle liste in input in sottoliste usando la seguente *comprehension*.

```
def divide(lst: list, n: int) -> list:

    return [lst[i * len(lst) // n: (i + 1) * len(lst) // n] for i in range(n)]

```

Questi metodi sono eseguiti sugli individui scelti, ma la scelta è eseguita dalla seguente funzione:

- **execute3Crossover**: Questa funzione infatti, seleziona casualmente su una popolazione della mating pool, gli individui che procederanno ad accoppiarsi, per formare poi una nuova popolazione una volta che tutti gli individui si siano accoppiati.

```
def execute3Crossover(p: Population, crossoverType: crossover, *args) -> list[
    Individual]:
    """
    Executes the crossover on a population taking 3 individuals instead of 2.
    :param p: The starting population.
    :param crossoverType: The crossover algorithm we want to perform.
    :param args: The arguments to pass to the crossover method as needed.
    :return: The new population.
    """
    newIndividuals = list()
    individuals = p.getIndividuals()
    size = len(p)

    if size % 3 != 0:
        raise ValueError("The size of the population must be divisible by 3!")
    else:
        for _ in range(round(len(p) / 3)):
            i1 = random.choice(individuals)
            p.removeIndividual(i1)
            i2 = random.choice(individuals)
            p.removeIndividual(i2)
            i3 = random.choice(individuals)
            p.removeIndividual(i3)
            i1, i2, i3 = crossoverType(i1, i2, i3, *args)
            newIndividuals.append(i1)
            newIndividuals.append(i2)
            newIndividuals.append(i3)

        return newIndividuals
```

7.5 Mutazione

La mutazione è quella fase degli algoritmi GA che esegue una modifica al corredo genetico, tale modifica può portare sia ad un vantaggio che ad un svantaggio, dipende esclusivamente dal caso, quelli da noi sviluppati sono:

- **randomSingleMutation**: Questa funzione può eseguire una mutazione su un singolo gene all'interno di un individuo scelto casualmente. Per popolazioni molto grandi, comporta troppa poca casualità. È quindi sconsigliato l'uso di della condizione di stop sulla base dell'aumento del valore di fitness costante, e di invece usare esclusivamente il numero di generazioni per evitare interruzioni precoci.

```
def randomSingleMutation(p: Population, mutationRate: float) -> Population:
    if random.random() < mutationRate:
        i = random.choice(p.getIndividuals())
        p.replaceIndividual(i, mu.mutateEx(i, p))

    return p
```

- **randomIndividualMutation**: Questa funzione invece, può eseguire una mutazione sull'intero individuo di una popolazione scelto casualmente, risolvendo il problema delle grandi popolazioni incontrata nella precedente versione.

```
def randomIndividualMutation(p: Population, mutationRate: float) -> Population:
    individuals = p.getIndividuals()

    if random.random() < mutationRate:
        i = random.choice(individuals)
        p.replaceIndividual(i, mu.mutateIndividual(i, p))

    return p
```

- **worstIndividualMutation**: Quest'ultima infine, può eseguire una mutazione sul peggior individuo di una popolazione controllando i singoli valori di fitness degli individui.

```
def worstIndividualMutation(p: Population, mutationRate: float) -> Population:
    individuals = p.getIndividuals()
    minF = individuals[0].fitness()
    indice = 0
    for index, individual in enumerate(individuals):
        if minF > individual.fitness():
            minF = individual.fitness()
            indice = index

    if random.random() < mutationRate:
        i = p.getIndividuals()[indice]
        p.replaceIndividual(i, mu.mutateIndividual(i, p))

    return p

```

Per effettuare le 3 tipologie di mutazioni, vengono utilizzati i seguenti metodi di supporto:

- **mutateEX**: Questa funzione seleziona un singolo gene casualmente dal dataset.

```
def mutateEx(i: Individual, p: Population) -> Individual:
    userId = p.getUser().getID()
    gen = p.getGeneration()
    oldEx = random.choice(i.getList())
    newEx = db.select_random_exercise(1, userId)[0]
    i.replaceExercise(oldEx, newEx, gen)
    evaluate(i, p.getUser())

    return i
```

- **mutateIndividual**: Questa funzione invece si occupa della generazione di un nuovo individuo andando sempre alla ricerca dei geni nel dataset.

```
def mutateIndividual(i: Individual, p: Population):
    userId = p.getUser().getID()
    gen = p.getGeneration()
    newIndividual = Individual(*db.select_random_exercise(len(i.getList()), userId))
    evaluate(newIndividual, p.getUser())
    for ex in newIndividual.getList():
        ex.setGeneration(gen)

    return newIndividual
```

È da precisare, che unicamente in queste due funzioni vengono prelevati nuovi geni dal dataset, e proprio per questo, viene inserita la corretta Generazione all'esercizio affinché, nel caso ci fosse necessità o curiosità, si potrebbe risalire sia a quante generazioni è sopravvissuto, sia in quale generazione è stato pescato.

7.6 Stopping Condition

Infine, l'intera generazione può terminare unicamente in due modi:

- **stoppingCondition**: esegue un controllo ad ogni generazione, in modo tale che nel caso non ci siano miglioramenti al valore Fitness entro un numero prestabilito di generazioni, l'algoritmo si ferma.

```
def stoppingCondition(p: Population, lastFitness, unchangedCount: int,
    increaseRate: float) -> tuple[int, int]:
    currentFitness = round(p.totalFitness())
    if lastFitness < currentFitness <= lastFitness + (lastFitness * increaseRate):
        unchangedCount += 1
    else:
        unchangedCount = 0
    lastFitness = p.totalFitness()
    return unchangedCount, lastFitness
```

- **condizione While**: ad ogni ciclo, viene incrementato il valore di una variabile *gen*. Quando raggiunge il valore di *maxGen*, l'algoritmo di ferma.

```
    [...]
    gen = 0
    maxGen = 400

    while gen < maxGen:
        gen += 1
        [...]
```

8 Esempio di esecuzione completa

Un esempio di esecuzione del nostro algoritmo Genetico. Viene utilizzato *Rank Selection*, *nPoint* e *random Individual Mutation*. I risultati vengono stampati in un file chiamato "*result.txt*" grazie alla funzione *printIntoResults*.

```
from GA.Initialization.populationInitializer import initialize
from GA.StoppingCondition.stopConditionMethods import stoppingCondition
from GA.Crossover.crossover3Parents import execute3Crossover as crossover3P
import GA.Crossover.crossover3Methods as c3Type
from Printer.printerMethods import printIntoResults
import GA.Selection.selectionMethods as sType
import GA.Mutation.mutationMethods as mType
from multiprocessing import Pool, cpu_count


# Responsabile della parallelizzazione dell'algoritmo su multipli processi
def startGA():
    p1 = initialize(6, 5, 904)
    p2 = initialize(6, 5, 914)
    p = [p1, p2]

    pool = Pool(processes=(cpu_count() - 1))

    for index, population in enumerate(p):
        pool.apply_async(GATasks, args=(index, population))

    pool.close()
    pool.join()

# Selezione -> Crossover -> Mutazione
def GATasks(processNumber, population):
    gen = 0
    maxGen = 400

    unchangedCount = 0
    lastFitness = 0
    maxEqualsGen = 5
    increaseRate = 1

    print(f"Started process {processNumber}.")
    while gen < maxGen and unchangedCount < maxEqualsGen:
        gen += 1

        print(f"Current generation for P:{processNumber}: {gen}")
        population.setIndividuals(sType.rankSelection(population))
        population.setIndividuals(crossover3P(population, c3Type.nPoint, 2))
        mType.randomIndividualMutation(population, 0.5)
        population.incrementGeneration()
        print("————————————————————————————————————————————————————————————————————————————————")
        unchangedCount, lastFitness = stoppingCondition(population, lastFitness, unchangedCount, increaseRate)

    print(f"\n\nProcess {processNumber} has ended.\n\n")
    printIntoResults(processNumber, population)


if __name__ == '__main__':
    startGA()
```

9 Considerazioni finali

Siamo giunti alla conclusione di un lungo viaggio, alla fine del quale ci riteniamo soddisfatti di quanto ottenuto, in quanto tutti i membri non avevano mai avuto esperienze pregresse con la programmazione di agenti intelligenti. L'emozione ottenuta al primo corretto funzionamento sarà un ricordo indelebile nella memoria di noi tutti. Ma non solo le gioie rimarranno, rimarranno anche gli sforzi e l'esperienza ottenuti sbattendo la testa innumerevoli volte, ma senza mai perderci d'animo. Riteniamo che sono stati raggiunti gli obiettivi che ci eravamo prefissati, riuscire a raccomandare esercizi. Sicuramente la fitness andrebbe calibrata con l'aiuto di una figura professionale nell'ambito della logopedia. Un aspetto di cui andiamo particolarmente fieri è la gestione in multiprocessing dell'algoritmo che rende possibile, se necessario, l'esecuzione sulla stessa popolazione, ma variando i fatti di selezione, crossover e mutazione. Un ringraziamento speciale a tutti coloro che hanno contribuito alla realizzazione di questo progetto, dai Project Manager, ai colleghi, che ci hanno fornito l'idea di base, dai professori ai tutor per averci insegnato le competenze necessarie alla reale realizzazione dell'algoritmo. Infine un ringraziamento a chiunque abbia raggiunto queste parole.

9.1 Cosa abbiamo Imparato

Grazie a questo progetto abbiamo potuto provare e applicare le conoscenze che abbiamo acquisito durante il corso di *Fondamenti di Intelligenza Artificiale*, e provare sulla pratica il nostro primo algoritmo genetico. Grazie a questa prova abbiamo imparato l'importanza dell'organizzazione e della stesura dei requisiti del progetto, prendendoci più tempo possibile per definire al meglio gli obiettivi da raggiungere e da realizzare. Sono state tante le ore passate solo all'idealizzazione dell'algoritmo generico, ancor più le ore dedicate allo studio della funzione di Fitness, per la scelta e lo scarto dei fattori da valutare per dare una raccomandazione mirata ed efficace. Ma ne è valsa la pena.

> *"Any sufficiently advanced technology is indistinguishable from magic"*.

10 Glossario

Termini speciali utilizzati:

- **Dataset**: Insieme di dati su cui un algoritmo opera durante l'analisi o l'addestramento.
- **Database**: Archivio centralizzato di dati, accessibile per il recupero e la modifica delle informazioni salvate.
- **Algoritmo Genetico**: Algoritmo di Intelligenza Artificiale che simula il processo evolutivo per tentare di risolvere problemi di ottimizzazione.
- **SQL**: Linguaggio standardizzato per la gestione di Database, utilizzato per eseguire operazioni sui dati.
- **Fitness**: Valore utilizzato negli Algoritmi Genetici per valutare la qualità di una soluzione proposta.
- **Dizionario**: Struttura dati in programmazione composta da associazioni chiave-valore.
- **Classe**: Costrutto in un linguaggio di programmazione utilizzato come modello per creare oggetti.
- **Algoritmo**: Soluzione strutturata a un problema, composta da operazioni e condizioni ben definite.
- **Comprehension**: È una tecnica sintattica di Python che consente di creare liste, dizionari o insiemi in modo conciso ed efficiente in una singola riga di codice.