TalkAID

Anna Benedetta Salerno Michele D'Arienzo Raffaele Monti Luigi Petrillo

Samuele Sparno

23 Gennaio 2024

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.

I metodi tradizionali potrebbero non essere altamente personalizzati alle esigenze specifiche dei pazienti, poiché potrebbe essere difficile adattarli in modo rapido ed efficiente.

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

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 grado di severità della patologia. 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 sull'esperienza del paziente o sul lasso di tempo trascorso dall'ultima interazione con quella tipologia di esercizio;
- Migliorare il sistema di consigli nel tempo, facendolo evolvere sulla base dei feedback del logopedista, il quale deciderà se un esercizio è appropriato o meno.

3 Specifica PEAS

Di seguito abbiamo elencato la specifca PEAS dell'agente intelligente.

3.1 Performance:

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

- La sua capacità di consigliare esercizi più mirati possibile per il paziente in base alla patologia.
- Il punteggio dell'esercizio che il paziente ha effettuato su quell'esercizio.
- Quanto tempo è passato dall'ultima volta che il paziente ha svolto l'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 e gli esercizi svolti, e alla lista degli esercizi svolti per ogni paziente.
- Non deterministico, in quanto lo stato dell'ambiente cambia indipendentemente dalle azioni dell'agente.
- 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.
- A singolo agente, in quanto l'unico agente che opera in questo ambiente è quello in oggetto.

3.3 Actuators:

- Pagina web dove viene creata una lista di esercizi in maniera tabellare consigliata per ogni paziente.

3.4 Sensors:

- Gli esercizi svolti dal paziente, gli esercizi assegnati al paziente da parte del logopedista, il punteggio per esercizio del paziente, e il tempo passato dall'ultima volta che il paziente ha effettuato l'esercizio.
- Il dataset è un riadattamento del database utilizzato dal sito web principale, 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.

4 Soluzione proposta

Date le nostre necessità, abbiamo potuto constatare che ciò di cui abbiamo bisogno è un algoritmo 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 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 individuo sarà associato a un insieme di esercizi raccomandati specifici per quel determinato paziente.

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 cercare 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 logopedici.

Di conseguenza, abbiamo deciso di creare un dataset nostro, utilizzando un database in MySQL e generando le varie tipologie di esercizi di nostra iniziativa, ottenendo 84 esercizi. La generazione dei pazienti e la generazione delle valutazioni degli esercizi sono state prodotte sinteticamente mediante l'utilizzo di valori randomici.

- 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 within the past year
        DATE_SUB(NOW(), INTERVAL FLOOR(RAND() * 365) DAY), -- Random completion date within the past year
        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
            SET j = 1;
            WHILE j <= 10 DO
                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,
        FLOOR(1 + RAND() * 10) as ReadingSeverity
    FROM
        user u
    WHERE
        u.ID_Therapist != 0;

```

Listing 3: Generazione delle severity (tutti gli user)

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

```

Listing 4: Generazione delle severity (tutti gli user) pt.2

```
    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,
        FLOOR(1 + RAND() * 10) as ReadingSeverity
    FROM
        user u
    WHERE
        u.ID_Therapist != 0
    ORDER BY
        RAND() -- Randomly order the users and pick the first one
    LIMIT 30;

```

Listing 5: Generazione delle severity (solo x users)

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 degli utenti, 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 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 riguardanti gli esercizi, interessante poiché ci permette di poter trovare il target specifico di un determinato esercizio.

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, elencando le funzioni create sono:

Per quanto riguarda le estrazioni degli esercizi:

- select_exercises_not_done
- select_done_exercises
- select_random_exercise

Per quanto riguarda le estrazioni dei pazienti:

- informationUser

5.3.1 Estrazione degli esercizi

Successivamente vi è un estratto del codice dove viene mostrata ogni funzione inerente alle interazioni con il database.

select_exercises_not_done: Questa funzione permette di selezionare gli esercizi non ancora svolti dall'paziente.

```
        def select_exercises_not_done(ID: int) -> list:
    connessione = Connector()
    lista = []
    cursor = None
    try:
        if connessione.get_connection() is not None:
            cursor = connessione.get_connection().cursor(dictionary=True)
            query = """
                        SELECT *
                        FROM exercise_glossary eg
                        WHERE NOT EXISTS (
                            SELECT 1
                            FROM exercise e
                            WHERE e.ID_exercise = eg.ID_exercise AND e.ID_user = %s);
                    """

            parametro = (ID,)
            cursor.execute(query, parametro)

            records = cursor.fetchall()

            for record in records:
                esercizio = Exercise(record["ID_Exercise"],
                                     record["Difficulty"],
                                     record["Target"],
                                     record["Type"],
                                     None,
                                     None,
                                     None)
                lista.append(esercizio)
            return lista

    except mysql.connector.Error as e:
        print("Error while connecting to MySQL ", e)
        return list()
    finally:
        if connessione.get_connection() is not None:
            if cursor is not None:
                cursor.close()
            connessione.get_connection().close()
```

select_done_exercises: Questa funzione permette il recupero egli esercizi svolti dal paziente.

```
def select_done_exercises(ID: int) -> dict:
    connessione = Connector()
    cursor = None
    esercizi = {}
    try:
        if connessione.get_connection() is not None:
            cursor = connessione.get_connection().cursor(dictionary=True)
            query = """
                        SELECT
                            e.ID_exercise AS ExerciseID,
                            eg.Difficulty AS ExerciseDifficulty,
                            eg.Type AS ExerciseType,
                            eg.Target AS ExerciseTarget,
                            e.Evaluation AS ExerciseEvaluation,
                            e.CompletionDate AS ExerciseCompletionDate,
                            e.Feedback AS ExerciseFeedback,
                            DATE_FORMAT(e.CompletionDate, '%Y-%m-%d') AS Exercis
    eCompletionDate,
                            e.Evaluation AS ExerciseEvaluation,
                            e.Feedback AS ExerciseFeedback
                        FROM
                            exercise e
                        JOIN
                            exercise_glossary eg ON e.ID_exercise = eg.ID_exercise
                        WHERE
                            e.ID_user = %s
                        ORDER BY
                            e.CompletionDate DESC
                        LIMIT 50;
                    """

            parametro = (ID,)
            cursor.execute(query, parametro)

            records = cursor.fetchall()

            for record in records:
                esercizio = Exercise(record["ExerciseID"],
                                     record["ExerciseDifficulty"],
                                     record["ExerciseTarget"],
                                     record["ExerciseType"],
                                     record["ExerciseEvaluation"],
                                     record["ExerciseCompletionDate"],
                                     record["ExerciseFeedback"])
                esercizi[record["ExerciseID"]] = esercizio
            return esercizi

    except mysql.connector.Error as e:
        print("Error while connecting to MySQL", e)
        return dict()
    finally:
        if connessione.get_connection() is not None:
            if cursor is not None:
                cursor.close()
            connessione.get_connection().close()
```

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.

```
def select_random_exercise(n: int, ID: int) -> list[Exercise]:
    connessione = Connector()
    lst = list()
    cursor = None
    try:
        if connessione.get_connection() is not None:
            cursor = connessione.get_connection().cursor(dictionary=True)
            query = """
                        SELECT
                          eg_random.ID_exercise,
                          eg_random.Difficulty,
                          eg_random.Target,
                          eg_random.Type,
                          DATE_FORMAT(e.CompletionDate, '%Y-%m-%d') AS Exercis
    eCompletionDate,
                          e.Evaluation,
                          e.Feedback
                        FROM (
                          SELECT *
                          FROM exercise_glossary
                          ORDER BY RAND()
                          LIMIT %s
                        ) AS eg_random
                        LEFT JOIN exercise e ON eg_random.ID_exercise = e.ID_exercise
                          AND e.ID_user = %s
                          AND e.InsertionDate = (
                            SELECT MAX(InsertionDate)
                            FROM exercise
                            WHERE ID_exercise = eg_random.ID_exercise
                              AND ID_user = %s
                          )
                        ORDER BY e.InsertionDate DESC;
                    """
            parametro = (n, ID, ID,)
            cursor.execute(query, parametro)
            records = cursor.fetchall()

            if records is not None:
                for record in records:
                    esercizio = Exercise(record["ID_exercise"],
                                         record["Difficulty"],
                                         record["Target"],
                                         record["Type"],
                                         record["Evaluation"],
                                         record["ExerciseCompletionDate"],
                                         record["Feedback"])
                    lst.append(esercizio)

            return lst

    except mysql.connector.Error as e:
        print("Error while connecting to MySQL ", e)
        return list()
    finally:
        if connessione.get_connection() is not None:
            if cursor is not None:
                cursor.close()
            connessione.get_connection().close()
```

5.3.2 Estrazione dei pazienti

informationUser: Questa funzione permette il recupero di informazioni inerenti alle varie patologie che il paziente soffre con il livello di gravità per ogniuna di essa.

```
def informationUser(ID: int) -> dict:
    connessione = Connector()
    cursor = None
    patologie = {}
    try:
        if connessione.get_connection() is not None:
            cursor = connessione.get_connection().cursor(dictionary=True)
            query = """
                    SELECT
                        c.Name,
                        pc.Severity,
                        pc.WritingSeverity,
                        pc.ReadingSeverity
                    FROM
                        patientcondition pc
                    JOIN
                        `condition` c ON pc.ID_condition = c.ID_condition
                    WHERE
                        pc.ID_patient = %s;
                            """
            parametro = (ID,)
            cursor.execute(query, parametro)

            records = cursor.fetchall()

            if len(records) == 0:
                return dict()
            else:
                for record in records:
                    tupla = (record["Severity"],
                             record["WritingSeverity"],
                             record["ReadingSeverity"])
                    patologie[record["Name"]] = tupla
                return patologie

    except mysql.connector.Error as e:
        print("Error while connecting to MySQL ", e)
        return dict()
    finally:
        if connessione.get_connection() is not None:
            if cursor is not None:
                cursor.close()
            connessione.get_connection().close()
```

6 Studio della funzione di fitness

Da qui cominciamo a descrivere lo studio che è stato effettuato per poter sviluppare la funzione di fitness.
La funzione di fintness è basata su varie informazioni inerenti agli esercizi e all'utente, più precisamente:

- **evaluateBasedFitness**: La funzione restituisce un valore che va da 0 a 100 ottenuto sottraendo alla votazione massima che si può avere e la votazione che l'utente ha avuto su quel esercizio.
- **dateBasedFitness**: Restituisce un valore sulla nase della quantita di tempo passata dalla data odierna alla data di completamento del esercizio.
- **difficultBasedFitness**: Restituisce un valore sulla base della difficolta del singolo esercizio.
- **severityBasedFitness**: Restituisce un valore da 0 a 100 sulla base dell'utilità dell'esercizio per il paziente controllando il tipo di esercizio e il tipo di probelmatiche del paziente, se l'esericizo concide con le problematiche del paziente.
- **targetBasedFitness**: Restituisce un valor ein base all'adeguatezza dell'esercizio alle tipologie di patologie del paziente(Es. Esercizio per chi soffre di dislessia: L'utente soffre di dislessia).
- **feedbackBasedFitness**: Restituisce un valore in base al feedback del paziente, quindi controlla se il paziente ha ricevuto un esercizio che gli aggrada e che gli è stato utile.
- **last50Fitness**: Valuta se l'esercizio offerto all'utente se è compreso negli ultimi 50 esercizi fatti se no allora verrà riproposto.

Tutti questi dati sono di vario tipo e di varia importanza infatti ogni dato nel calcolo del valore fitness è moltiplicato con una costante di importanza dedicata, tale variabile è modificabile in modo tale che il logopedista può modificare questa variabile in modo da considerare di più un dato rispetto ad un altro a sua descrizione.

Più precisamente i dati su cui noi eseguiamo l'algoritmo GA è Popolation tale dato è strutturato da vari campi, come:

- **user**: informazioni inerenti al paziente come le sue patologie
- **individuals**: una lista di set di esercizi da elaborare
- **entirefitness**: valore fitness dell'intero Popolation

```python
from GA.Individual.user import User
from GA.Individual.exerciseIndividual import Individual


class Population:
    def __init__(self, u: User, *args: Individual):
        self._user = u
        self._entireFitness = 0
        self._currGen = 0
        self._individuals = list()
        for i in args:
            self._individuals.append(i)

    def __len__(self):
        return len(self._individuals)

    def __str__(self):
        return f"Population for user {self._user}: {self._individuals}"

    def __getitem__(self, item: int):
        if self._individuals[item] is not None:
            return self._individuals[item]
        else:
            return None

    def totalFitness(self) -> int:
        self._entireFitness = 0
        for individual in self._individuals:
            self._entireFitness += individual.fitness()
        return self._entireFitness

    def replaceIndividual(self, oldIndividual: Individual, newIndividual: Individual):
        if oldIndividual in self._individuals:
            index = self._individuals.index(oldIndividual)
            self._individuals[index] = newIndividual
        else:
            raise ValueError(f"Invalid Individual_ {Individual}")

    def removeIndividual(self, i: Individual):
        if i in self._individuals:
            self._individuals.remove(i)
        else:
            raise ValueError("Individual not found in the population")

    def getIndividuals(self) -> list[Individual]:
        return self._individuals

    def setIndividuals(self, newIndividuals: list[Individual]):
        self._individuals = newIndividuals

    def getUser(self) -> User:
        return self._user

    def incrementGeneration(self):
        self._currGen += 1

    def getGeneration(self) -> int:
        return self._currGen
```

User è caratterizzato da:

- **ID**: indicante l'id identificativo del paziente
- **conditions**: lista di patologie del paziente
- **exercises**: ultimi 50 esercizi fatti dal paziente

```
class User:
def __init__(self, ID: int, conditions: dict, exercises: dict):

    self._ID = ID
    self._conditions = conditions
    self._exercises = exercises

def __str__(self):
    return f"User ID: {self._ID}"

def __repr__(self):
    return f"User ID: {self._ID}"

def getID(self) -> int:
    return self._ID

def getConditions(self) -> dict:
    return self._conditions.copy()

def getExercises(self) -> dict:
    return self._exercises.copy()
```

Conditions è un dizionario caratterizzato da:

- **Key**: nome della patologia
- **value**: uan tupla che indica la gravità della patologia nei tre aspetti Severity(gravità generale), SeverytyWrite(gravità nella scrittura) e SeverityRead(gravità nella lettura)

Più precisamente Individual è caratterizzato:

- **individualFitness**: valore fitness del singolo set di esercizi
- **exercises**: l'insieme di esercizi che compongono il set di esercizi

```python
from GA.Individual.exercise import Exercise


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

    def __len__(self):
        return len(self._exercises)

    def __iter__(self):
        return iter(self._exercises)

    def __str__(self):
        return f"Individual {id(self)}: {self._exercises}"

    def __repr__(self):
        return f"\nI—{id(self)}: {self._exercises}"

    def fitness(self) -> float:
        self._individualFitness = 0
        for ex in self._exercises:
            self._individualFitness += ex.getFitnessValue()
        return self._individualFitness

    def getExerciseByIndex(self, i: int) -> Exercise | None:
        if self._exercises[i] is not None:
            return self._exercises[i]
        else:
            return None

    def replaceExercise(self, oldEx: Exercise, newEx: Exercise, gen: int):
        newEx.setGeneration(gen)
        index = self._exercises.index(oldEx)
        self._exercises[index] = newEx

    def getList(self) -> list[Exercise]:
        return self._exercises

    def setList(self, lst: list):
        self._exercises = lst
```

I singoli esercizi sono caratterizzati da:

- **fitness**: valore fitness calcolato del singolo esercizio
- **generation**: indicante a quando è stato generato l'esercizio e inserito nel set
- **ID**: id unico del'esercizio
- **exerciseDifficulty**: difficolta dell'esercizio
- **target**: indica a quale tipologia di patologie è meglio dedicata l'esercizio
- **type**: indica il tipo di esercizio(lettura, scrittura...)
- **lastEvaluation**: indicante l'ultima vlalutazioen dell'esercizio se questo è stato già eseguito dal paziente
- **lasCompletionDate**: indicante l'ultima data in cui è stato fatto l'esercizio
- **lastFeedback**: 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

def __str__(self):
    return "Exercise ID: {}, Exercise Type: {}".format(self._exerciseID, self.
_exType)

def __repr__(self):
    return "ID: {}".format(self._exerciseID)
def getFitnessValue(self) -> float:
    return self._fitnessValue
def setFitnessValue(self, value: float) -> None:
    self._fitnessValue = value
def getGeneration(self) -> int:
    return self._generation
def setGeneration(self, value: int) -> None:
    self._generation = value
def getExerciseID(self) -> int:
    return self._exerciseID
def getExerciseDifficulty(self) -> int:
    return self._exerciseDifficulty
def getExerciseTarger(self) -> str:
    return self._target
def getExerciseType(self) -> str:
    return self._exType
def getLastEvaluation(self) -> int | None:
    return self._lastEvaluation
def getLastCompletionDate(self) -> str | None:
    return self._lastCompletionDate
def getLastFeedback(self) -> int:
    return self._lastFeedback
```

7 Sviluppo del nostro algoritmo genetico

Per lo sviluppo del nostro porgetto abbiamo optato per una soluzioen con Algoritmo Genetico trovandoci più comodi nella sua implementaizone per la suddivisione dei compiti tra i vari membri del team.

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

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

7.1 Selezione

Nella fase di selezione abbiamo implementato 3 tipi diversi di algoritmi:

- **Roulette Wheel**: Sfruttiamo il valore della funzione di fintess di ogni elemento per impostare la probabilità che venga scelto, tramite un sorteggio randomico dii un numero verrà prellevato l'elemento.

```
1  import random
2  import GA.Selection.selectionUtility as su
3  from GA.Individual.exerciseIndividual import Individual
4  from GA.Population.exercisePopulation import Population
5  from GA.Evaluation.fitness import individualFitness as evaluate
6
7          def rouletteWheel(population: Population):
8              user = population.getUser()
9              totalFitness = 0
10             for i in population.getIndividuals():
11                 evaluate(i, user)
12                 totalFitness += i.fitness()
13
14             newIndividuals = list()
15             while len(newIndividuals) < len(population):
16                 value = random.random() * totalFitness
17                 comulativeFitness = 0
18                 for i in population.getIndividuals():
19                     comulativeFitness += i.fitness()
20                     if comulativeFitness >= value:
21                         newIndividuals.append(Individual(*i.getList()))
22                         break
23             return newIndividuals
```

- **Rank Selection**: Sfrutta il rank di ogni elemento per suddividere una torta di selezione in partizioni proporzionare al rank dell'elemento, il rank va dal più basso al più alto proporzionata alla percentuale di possibilità di essere selezionato, questa metodologia riesce ad escludere meno gli elementi rispetto all aroulette siccome la percentuale è proporzionata agli elementi e non al valore fitness. Per individuare i rank degli elementi viene richiamata la funzione findRank.

```
import random
import GA.Selection.selectionUtility as su
from GA.Individual.exerciseIndividual import Individual
from GA.Population.exercisePopulation import Population
from GA.Evaluation.fitness import individualFitness as evaluate

        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
```

```
            import random
from GA.Individual.exerciseIndividual import Individual


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

```

- **item Random Selection**: Questa selezione è completamente casuale infatti non seguendo alcun tipo di logica può generare dei dati completamente indefficienti.

```
import random
import GA.Selection.selectionUtility as su
from GA.Individual.exerciseIndividual import Individual
from GA.Population.exercisePopulation import Population
from GA.Evaluation.fitness import individualFitness as evaluate

def randomSelection(population: Population):
    newP = []
    while len(newP) < len(population):
        newP.append(random.choice(population))
    return newP

```

7.2 Crossover

La fase Crossover utilizza degli algoritmi per la generazione di nuovi elementi partendo dagli elementi scelti nella fase di Selezione.

Le tipologie che sono state create di Crossover non solo per tipologia ma anche per il numero di elementi da usare.

Le mitodologie di Crossover eseguite su 3 elementi sono:

- **nPoint**: Questo Crossover va a modificare n punti di 3 liste di esercizi, ogni lista è sarà suddivisa in sottoliste di meno elementi queste'ultime rappresentano i punti che saranno combinati con gli altri.

```
1  from random import random
2  from random import randint
3  from GA.Crossover.crossoverUtility import divide
4  from GA.Individual.exerciseIndividual import Individual
5
6
7  def nPoint(i1: Individual, i2: Individual, i3: Individual, n: int) -> tuple[Individual
       , Individual, Individual]:
8      if len(i1) != len(i2) != len(i3):
9          raise ValueError("Invalid individual length! They have to be the same.")
10     elif n > len(i1):
11         raise ValueError("Invalid n value, must be less than the lenght of the
       individual!")
12
13     dividedl1, dividedl2, dividedl3 = divide(i1.getList(), n), divide(i2.getList(), n)
       , divide(i3.getList(), n)
14
15     newl1 = list()
16     newl2 = list()
17     newl3 = list()
18
19     for i, triplet in enumerate(zip(dividedl1, dividedl2, dividedl3)):
20         if i % 3 == 0:
21             newl1.extend(triplet[0])
22             newl2.extend(triplet[1])
23             newl3.extend(triplet[2])
24         elif i % 3 == 1:
25             newl1.extend(triplet[2])
26             newl2.extend(triplet[0])
27             newl3.extend(triplet[1])
28         elif i % 3 == 2:
29             newl1.extend(triplet[1])
30             newl2.extend(triplet[2])
31             newl3.extend(triplet[0])
32
33     i1.setList(newl1)
34     i2.setList(newl2)
35     i3.setList(newl3)
36
37     return i1, i2, i3
```

- **nPointReverse**: Questo Crossover va a modificare n punti di 3 liste di esercizi, ogni lista è sarà suddivisa in sottoliste di meno elementi queste'ultime rappresentano i punti che saranno combinati con gli altri ma in ordine diverso a nPoint.

```
from random import random
from random import randint
from GA.Crossover.crossoverUtility import divide
from GA.Individual.exerciseIndividual import Individual

        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**: Questo Crossover va a modificare n punti di 3 liste di esercizi, ogni lista è sarà suddivisa in sottoliste di meno elementi queste'ultime rappresentano i punti che saranno combinati con gli altri ma in ordine casuale.

```
from random import random
from random import randint
from GA.Crossover.crossoverUtility import divide
from GA.Individual.exerciseIndividual import Individual

            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])

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

    return i1, i2, i3

```

Le metodologie di Crossover eseguite su 2 elementi sono:

- **nPoint**: Questo Crossover va a modificare n punti di 2 liste di esercizi, ogni lista sarà suddivisa in sottoliste di meno elementi queste'ultime rappresentano i punti che saranno combinati con gli altri.

```
1  from random import random
2  from GA.Crossover.crossoverUtility import divide
3  from GA.Individual.exerciseIndividual import Individual
4
5  def nPoint(i1: Individual, i2: Individual, n: int) -> tuple[Individual, Individual]:
6      if len(i1) != len(i2):
7          raise ValueError("Invalid individual length! They have to be the same.")
8      elif n > len(i1):
9          raise ValueError("Invalid n value, must be less than the lenght of the
       individual!")
10
11     dividedI1, dividedI2 = divide(i1.getList(), n), divide(i2.getList(), n)
12
13     newI1 = list()
14     newI2 = list()
15
16     for i, pair in enumerate(zip(dividedI1, dividedI2)):
17         if i % 2 == 0:
18             newI1.extend(pair[0])
19             newI2.extend(pair[1])
20         else:
21             newI1.extend(pair[1])
22             newI2.extend(pair[0])
23
24     i1.setList(newI1)
25     i2.setList(newI2)
26
27     return i1, i2
28
```

- **nPointReverse**: Questo Crossover va a modificare n punti di 2 liste di esercizi, ogni lista è sarà suddivisa in sottoliste di meno elementi queste'ultime rappresentano i punti che saranno combinati con gli altri ma in ordine diverso a nPoint.

```
from random import random
from GA.Crossover.crossoverUtility import divide
from GA.Individual.exerciseIndividual import Individual

def nPointReverse(i1: Individual, i2: Individual, n: int) -> tuple[Individual, Individual]:
    if len(i1) != len(i2):
        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!")

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

    newI1 = []
    newI2 = []

    for i, pair in enumerate(zip(dividedI1, dividedI2)):
        if i % 2 == 0:
            newI1.extend(pair[1])
            newI2.extend(pair[0])
        else:
            newI1.extend(pair[0])
            newI2.extend(pair[1])

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

    return i1, i2
```

- **nPointRandom**: Questo Crossover va a modificare n punti di 2 liste di esercizi, ogni lista è sarà suddivisa in sottoliste di meno elementi queste'ultime rappresentano i punti che saranno combinati con gli altri ma in ordine casuale.

```python
from random import random
from GA.Crossover.crossoverUtility import divide
from GA.Individual.exerciseIndividual import Individual

def nPointRandom(i1: Individual, i2: Individual, n: int,  crossoverProbability: float) -> tuple[Individual, Individual]:
    if len(i1) != len(i2):
        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 = divide(i1.getList(), n), divide(i2.getList(), n)

    newI1 = []
    newI2 = []

    for i, pair in enumerate(zip(dividedI1, dividedI2)):
        if random() < crossoverProbability:
            newI1.extend(pair[1])
            newI2.extend(pair[0])
        else:
            newI1.extend(pair[0])
            newI2.extend(pair[1])

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

    return i1, i2
```

- **unifromCrossover**: Questo Crossover va a modificare n punti di 2 liste di esercizi, ogni sottolista viene suddivisa nei suoi singoli di esercizi e queste suddivisioni potrebbero essere scambiate tra le 2 liste .

```python
from random import random
from GA.Crossover.crossoverUtility import divide
from GA.Individual.exerciseIndividual import Individual

def uniformCrossover(i1: Individual, i2: Individual, crossoverProbability: float) -> tuple[Individual, Individual]:
    if len(i1) != len(i2):
        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, dividedI2 = divide(i1.getList(), len(i1)), divide(i2.getList(), len(i2))

    newI1 = []
    newI2 = []

    for i, pair in enumerate(zip(dividedI1, dividedI2)):
        if random() < crossoverProbability:
            newI1.extend(pair[1])
        else:
            newI1.extend(pair[0])

        if random() < crossoverProbability:
            newI2.extend(pair[0])
        else:
            newI2.extend(pair[1])

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

    return i1, i2
```

Queste tipologie di Crossover sono eseguite da:

- **executeCrossover**: Questa funzione esegue il Crossover su una popolazione, la popolazioen rappresenta un insieme di informazioni inerenti all'utente e ad una lista di esercizi.

```
def executeCrossover(p: Population, crossoverType: crossover, *args) -> list[
    Individual]:
    newIndividuals = list()
    individuals = p.getIndividuals()
    size = len(p)

    if size % 2 != 0:
        raise ValueError("The size of the population must be even!")
    else:
        for _ in range(round(len(p) / 2)):
            i1 = random.choice(individuals)
            p.removeIndividual(i1)
            i2 = random.choice(individuals)
            p.removeIndividual(i2)
            i1, i2 = crossoverType(i1, i2, *args)
            newIndividuals.append(i1)
            newIndividuals.append(i2)

        return newIndividuals
```

- **execute3Crossover**: Questa funzioen esegue il Crossover su 3 popolazioen che invece di 2 così da effettuare combinazioni più rillevanti.

```
from GA.Population.exercisePopulation import Population
from GA.Individual.exerciseIndividual import Individual
import GA.Crossover.crossover3Methods as crossover
import random

def execute3Crossover(p: Population, crossoverType: crossover, *args) -> list[
    Individual]:
    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)
            i1, i2, i3 = crossoverType(i1, i2, i3, *args)
            newIndividuals.append(i1)
            newIndividuals.append(i2)
            newIndividuals.append(i3)

        return newIndividuals
```

Tutte queste funzionalità vengono effettuate sfruttando la funzione divide che alla ricezioen di ua lista di esercizi essa viene suddivisa in una lista di liste formate dagli esercizi della lista originale, questa fuznioen permette la suddivisione della lista in punti che poi saranno usati dai Crossover.

7.3 Mutazione

La mutazione è quella fase degli algoritmi GA che esegue una modifica al singolo dato, tale modifica può portare sia ad un vanaggio che ad un svantaggio, dipende dal tipo di algoritmo che viene usato, quelli da noi sviluppati sono:

- **randomSingleMutation**: Questa funzione può eseguire una mutazione su un esercizio di una popolazione. Se viene effettuate richiama la funzieon mutateEX.

```
1  import random
2  from GA.Population.exercisePopulation import Population
3  import GA.Mutation.mutationUtility as mu
4
5  def randomSingleMutation(p: Population, mutationRate: float) -> Population:
6      if random.random() < mutationRate:
7          i = random.choice(p.getIndividuals())
8          p.replaceIndividual(i, mu.mutateEx(i, p))
9
10     return p
```

- **randomIndividualMutation**: Questa fuznione può eseguire una mutazione sull'intero set di esercizi di una popolazione, se la mutazioen accade richiama la funzione mutateIndividual.

```
1  import random
2  from GA.Population.exercisePopulation import Population
3  import GA.Mutation.mutationUtility as mu
4
5  def randomIndividualMutation(p: Population, mutationRate: float) -> Population:
6      individuals = p.getIndividuals()
7
8      if random.random() < mutationRate:
9          i = random.choice(individuals)
10         p.replaceIndividual(i, mu.mutateIndividual(i, p))
11
12     return p
```

- **worstIndividualMutation**: Questa funzione può eseguire una mutazione ma con la scelta del set di esercizi da cambiare, infatti viene individuato il set di esercizi con la fitness minore, successivamente viene richiamata la funzioen mutateIndividual.

```
import random
from GA.Population.exercisePopulation import Population
import GA.Mutation.mutationUtility as mu

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

```

Queste 3 tipologie di mutazioni per funzionare utilizzano:

- **mutateEX**: Questa fuznione esegue uan ricerca casuale nel set di esercizi che ha ricevuto e sul DataBase per poi cambiare l'esercizio.

```
import random
import DataBase.DBExercise as db
from GA.Population.exercisePopulation import Population
from GA.Individual.exerciseIndividual import Individual
from GA.Evaluation.fitness import individualFitness as evaluate

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**: esegue una ricerca casuale nel DataBase di un numero di esercizi per un intero set per poi sostituirli e ricalcolare il valore fitness.

```
1  import random
2  import DataBase.DBExercise as db
3  from GA.Population.exercisePopulation import Population
4  from GA.Individual.exerciseIndividual import Individual
5  from GA.Evaluation.fitness import individualFitness as evaluate
6
7  def mutateIndividual(i: Individual, p: Population):
8      userId = p.getUser().getID()
9      gen = p.getGeneration()
10     newIndividual = Individual(*db.select_random_exercise(len(i.getList()), userId))
11     evaluate(newIndividual, p.getUser())
12     for ex in newIndividual.getList():
13         ex.setGeneration(gen)
14
15     return newIndividual
```