

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 specifica 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

```
1      INSERT INTO exercise (ID_user , ID_exercise, InsertionDate , CompletionDate ,
2      Evaluation , Feedback)
3      VALUES
4      (
5          FLOOR(904 + RAND() * (1003 - 904 + 1)), -- Random user ID
6          FLOOR(1 + RAND() * (84 - 1 + 1)), -- Random exercise ID
7          DATE.SUB(NOW(), INTERVAL FLOOR(RAND() * 365) DAY), -- Random date within
the past year
8          DATE.SUB(NOW(), INTERVAL FLOOR(RAND() * 365) DAY), -- Random completion
date within the past year
9          FLOOR(RAND() * (100 - 0 + 1)), -- Random evaluation between 0 and 100
10         CASE
11             WHEN RAND() <= 0.33 THEN -1
12             WHEN RAND() <= 0.66 THEN 0
13             ELSE 1
14         END -- Random feedback (-1, 0, or 1)
15     );
```

Listing 1: Generazione Casuale esecuzione esercizio

```
1      DELIMITER //
2
3      CREATE PROCEDURE GenerateUsers()
4      BEGIN
5          DECLARE i INT DEFAULT 1;
6          DECLARE j INT;
7
8          WHILE i <= 10 DO
9              SET j = 1;
10             WHILE j <= 10 DO
11                 INSERT INTO user (ID_Therapist) VALUES (i);
12                 SET j = j + 1;
13             END WHILE;
14             SET i = i + 1;
15         END WHILE;
16     END //
17
18     DELIMITER ;
19
20     CALL GenerateUsers();
21
```

Listing 2: Generazione degli utenti

```

1      INSERT INTO patientcondition (ID_condition , ID_patient , Severity ,
2      WritingSeverity , ReadingSeverity)
3      SELECT
4          FLOOR(1 + RAND() * 12) as ID_condition , — Random condition ID (1 to 12)
5          u.ID as ID_patient ,
6          1 as Severity ,
7          FLOOR(1 + RAND() * 10) as WritingSeverity ,
8          FLOOR(1 + RAND() * 10) as ReadingSeverity
9      FROM
10         user u
11      WHERE
12         u.ID_Therapist != 0;

```

Listing 3: Generazione delle severity (tutti gli user)

```

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

```

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

```

1      INSERT INTO patientcondition (ID_condition , ID_patient , Severity ,
2      WritingSeverity , ReadingSeverity)
3      SELECT
4          FLOOR(1 + RAND() * 12) as ID_condition , — Random condition ID (1 to 12)
5          u.ID as ID_patient ,
6          1 as Severity ,
7          FLOOR(1 + RAND() * 10) as WritingSeverity ,
8          FLOOR(1 + RAND() * 10) as ReadingSeverity
9      FROM
10         user u
11      WHERE
12         u.ID_Therapist != 0
13      ORDER BY
14         RAND() — Randomly order the users and pick the first one
15      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: exercise_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'utente.

```
1      def select_exercises_not_done(ID: int) -> list:
2  connessione = Connector()
3  lista = []
4  cursor = None
5  try:
6      if connessione.get_connection() is not None:
7          cursor = connessione.get_connection().cursor(dictionary=True)
8          query = """
9              SELECT *
10             FROM exercise_glossary eg
11             WHERE NOT EXISTS (
12                 SELECT 1
13                 FROM exercise e
14                 WHERE e.ID_exercise = eg.ID_exercise AND e.ID_user = %s);
15             """
16
17         parametro = (ID,)
18         cursor.execute(query, parametro)
19
20         records = cursor.fetchall()
21
22         for record in records:
23             esercizio = Exercise(record["ID_Exercise"],
24                                   record["Difficulty"],
25                                   record["Target"],
26                                   record["Type"],
27                                   None,
28                                   None,
29                                   None)
30             lista.append(esercizio)
31         return lista
32
33     except mysql.connector.Error as e:
34         print("Error while connecting to MySQL ", e)
35         return list()
36     finally:
37         if connessione.get_connection() is not None:
38             if cursor is not None:
39                 cursor.close()
40             connessione.get_connection().close()
```

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

```
1 def select_done_exercises(ID: int) -> dict:
2     connessione = Connector()
3     cursor = None
4     esercizi = {}
5     try:
6         if connessione.get_connection() is not None:
7             cursor = connessione.get_connection().cursor(dictionary=True)
8             query = """
9                 SELECT
10                    e.ID_exercise AS ExerciseID ,
11                    eg.Difficulty AS ExerciseDifficulty ,
12                    eg.Type AS ExerciseType ,
13                    eg.Target AS ExerciseTarget ,
14                    e.Evaluation AS ExerciseEvaluation ,
15                    e.CompletionDate AS ExerciseCompletionDate ,
16                    e.Feedback AS ExerciseFeedback ,
17                    DATE.FORMAT(e.CompletionDate , '%Y-%m-%d') AS Exercis
18                    eCompletionDate ,
19                    e.Evaluation AS ExerciseEvaluation ,
20                    e.Feedback AS ExerciseFeedback
21                FROM
22                    exercise e
23                JOIN
24                    exercise_glossary eg ON e.ID_exercise = eg.ID_exercise
25                WHERE
26                    e.ID_user = %s
27                ORDER BY
28                    e.CompletionDate DESC
29                LIMIT 50;
30            """
31            parametro = (ID,)
32            cursor.execute(query , parametro)
33
34            records = cursor.fetchall()
35
36            for record in records:
37                esercizio = Exercise(record["ExerciseID"],
38                                    record["ExerciseDifficulty"],
39                                    record["ExerciseTarget"],
40                                    record["ExerciseType"],
41                                    record["ExerciseEvaluation"],
42                                    record["ExerciseCompletionDate"],
43                                    record["ExerciseFeedback"])
44                esercizi[record["ExerciseID"]] = esercizio
45            return esercizi
46
47        except mysql.connector.Error as e:
48            print("Error while connecting to MySQL ", e)
49            return dict()
50        finally:
51            if connessione.get_connection() is not None:
52                if cursor is not None:
53                    cursor.close()
54                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.

```

1 def select_random_exercise(n: int, ID: int) -> list[Exercise]:
2     connessione = Connector()
3     lst = list()
4     cursor = None
5     try:
6         if connessione.get_connection() is not None:
7             cursor = connessione.get_connection().cursor(dictionary=True)
8             query = """
9                 SELECT
10                    eg_random.ID_exercise,
11                    eg_random.Difficulty,
12                    eg_random.Target,
13                    eg_random.Type,
14                    DATEFORMAT(e.CompletionDate, '%Y-%m-%d') AS ExerciseCompletionDate,
15                    e.Evaluation,
16                    e.Feedback
17                FROM (
18                    SELECT *
19                    FROM exercise_glossary
20                    ORDER BY RAND()
21                    LIMIT %s
22                ) AS eg_random
23                LEFT JOIN exercise e ON eg_random.ID_exercise = e.ID_exercise
24                AND e.ID_user = %s
25                AND e.InsertionDate = (
26                    SELECT MAX(InsertionDate)
27                    FROM exercise
28                    WHERE ID_exercise = eg_random.ID_exercise
29                    AND ID_user = %s
30                )
31                ORDER BY e.InsertionDate DESC;
32            """
33            parametro = (n, ID, ID,)
34            cursor.execute(query, parametro)
35            records = cursor.fetchall()
36
37            if records is not None:
38                for record in records:
39                    esercizio = Exercise(record["ID_exercise"],
40                                         record["Difficulty"],
41                                         record["Target"],
42                                         record["Type"],
43                                         record["Evaluation"],
44                                         record["ExerciseCompletionDate"],
45                                         record["Feedback"])
46                    lst.append(esercizio)
47
48            return lst
49
50     except mysql.connector.Error as e:
51         print("Error while connecting to MySQL ", e)
52         return list()
53     finally:
54         if connessione.get_connection() is not None:
55             if cursor is not None:
56                 cursor.close()
57                 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.

```
1 def informationUser(ID: int) -> dict:
2     connessione = Connector()
3     cursor = None
4     patologie = {}
5     try:
6         if connessione.get_connection() is not None:
7             cursor = connessione.get_connection().cursor(dictionary=True)
8             query = """
9                 SELECT
10                    c.Name,
11                    pc.Severity ,
12                    pc.WritingSeverity ,
13                    pc.ReadingSeverity
14                FROM
15                    patientcondition pc
16                JOIN
17                    'condition' c ON pc.ID_condition = c.ID_condition
18                WHERE
19                    pc.ID_patient = %s;
20                """
21            parametro = (ID,)
22            cursor.execute(query, parametro)
23
24            records = cursor.fetchall()
25
26            if len(records) == 0:
27                return dict()
28            else:
29                for record in records:
30                    tupla = (record["Severity"],
31                            record["WritingSeverity"],
32                            record["ReadingSeverity"])
33                    patologie[record["Name"]] = tupla
34            return patologie
35
36     except mysql.connector.Error as e:
37         print("Error while connecting to MySQL ", e)
38         return dict()
39     finally:
40         if connessione.get_connection() is not None:
41             if cursor is not None:
42                 cursor.close()
43                 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 fitness 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 base della quantità di tempo passata dalla data odierna alla data di completamento del esercizio.
- **difficultBasedFitness**: Restituisce un valore sulla base della difficoltà 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 problematiche del paziente, se l'esercizio coincide con le problematiche del paziente.
- **targetBasedFitness**: Restituisce un valore in 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 Population 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 Population

```

1 from GA.Individual.user import User
2 from GA.Individual.exerciseIndividual import Individual
3
4
5 class Population:
6     def __init__(self, u: User, *args: Individual):
7         self._user = u
8         self._entireFitness = 0
9         self._currGen = 0
10        self._individuals = list()
11        for i in args:
12            self._individuals.append(i)
13
14        def __len__(self):
15            return len(self._individuals)
16
17        def __str__(self):
18            return f"Population for user {self._user}: {self._individuals}"
19
20        def __getitem__(self, item: int):
21            if self._individuals[item] is not None:
22                return self._individuals[item]
23            else:
24                return None
25
26        def totalFitness(self) -> int:
27            self._entireFitness = 0
28            for individual in self._individuals:
29                self._entireFitness += individual.fitness()
30            return self._entireFitness
31
32        def replaceIndividual(self, oldIndividual: Individual, newIndividual: Individual):
33            if oldIndividual in self._individuals:
34                index = self._individuals.index(oldIndividual)
35                self._individuals[index] = newIndividual
36            else:
37                raise ValueError(f"Invalid Individual_ {Individual}")
38
39        def removeIndividual(self, i: Individual):
40            if i in self._individuals:
41                self._individuals.remove(i)
42            else:
43                raise ValueError("Individual not found in the population")
44
45        def getIndividuals(self) -> list[Individual]:
46            return self._individuals
47
48        def setIndividuals(self, newIndividuals: list[Individual]):
49            self._individuals = newIndividuals
50
51        def getUser(self) -> User:
52            return self._user
53
54        def incrementGeneration(self):
55            self._currGen += 1
56
57        def getGeneration(self) -> int:
58            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

```
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
7
8     def __str__(self):
9         return f"User ID: {self._ID}"
10
11     def __repr__(self):
12         return f"User ID: {self._ID}"
13
14     def getID(self) -> int:
15         return self._ID
16
17     def getConditions(self) -> dict:
18         return self._conditions.copy()
19
20     def getExercises(self) -> dict:
21         return self._exercises.copy()
```

Conditions è un dizionario caratterizzato da:

- **Key:** nome della patologia
- **value:** una tupla che indica la gravità della patologia nei tre aspetti Severity(gravità generale), SeverityWrite(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

```
1 from GA.Individual.exercise import Exercise
2
3
4 class Individual:
5     def __init__(self, *args: Exercise):
6         self._individualFitness = 0
7         self._exercises = list()
8         for ex in args:
9             self._exercises.append(ex)
10
11     def __len__(self):
12         return len(self._exercises)
13
14     def __iter__(self):
15         return iter(self._exercises)
16
17     def __str__(self):
18         return f"Individual {id(self)}: {self._exercises}"
19
20     def __repr__(self):
21         return f"\n!-{id(self)}: {self._exercises}"
22
23     def fitness(self) -> float:
24         self._individualFitness = 0
25         for ex in self._exercises:
26             self._individualFitness += ex.getFitnessValue()
27         return self._individualFitness
28
29     def getExerciseByIndex(self, i: int) -> Exercise | None:
30         if self._exercises[i] is not None:
31             return self._exercises[i]
32         else:
33             return None
34
35     def replaceExercise(self, oldEx: Exercise, newEx: Exercise, gen: int):
36         newEx.setGeneration(gen)
37         index = self._exercises.index(oldEx)
38         self._exercises[index] = newEx
39
40     def getList(self) -> list[Exercise]:
41         return self._exercises
42
43     def setList(self, lst: list):
44         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 dell'esercizio
- **exerciseDifficulty**: difficoltà 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 valutazione dell'esercizio se questo è stato già eseguito dal paziente
- **lastCompletionDate**: indicante l'ultima data in cui è stato fatto l'esercizio
- **lastFeedback**: Feedback lasciato dal paziente all'esercizio

```
1 class Exercise:
2     def __init__(self, ID: int, exerciseDifficulty: int, target: str, exType: str,
3         lastEvaluation: int | None, lastCompletionDate: str | None,
4         lastFeedback: int | None):
5         self._fitnessValue = 0
6         self._generation = 0
7         self._exerciseID = ID
8         self._exerciseDifficulty = exerciseDifficulty
9         self._target = target
10        self._exType = exType
11        self._lastEvaluation = lastEvaluation
12        self._lastCompletionDate = lastCompletionDate
13        self._lastFeedback = lastFeedback
14
15    def __str__(self):
16        return "Exercise ID: {}, Exercise Type: {}".format(self._exerciseID, self._exType)
17
18    def __repr__(self):
19        return "ID: {}".format(self._exerciseID)
20
21    def getFitnessValue(self) -> float:
22        return self._fitnessValue
23
24    def setFitnessValue(self, value: float) -> None:
25        self._fitnessValue = value
26
27    def getGeneration(self) -> int:
28        return self._generation
29
30    def setGeneration(self, value: int) -> None:
31        self._generation = value
32
33    def getExerciseID(self) -> int:
34        return self._exerciseID
35
36    def getExerciseDifficulty(self) -> int:
37        return self._exerciseDifficulty
38
39    def getExerciseTarget(self) -> str:
40        return self._target
41
42    def getExerciseType(self) -> str:
43        return self._exType
44
45    def getLastEvaluation(self) -> int | None:
46        return self._lastEvaluation
47
48    def getLastCompletionDate(self) -> str | None:
49        return self._lastCompletionDate
50
51    def getLastFeedback(self) -> int:
52        return self._lastFeedback
```

7 Sviluppo del nostro algoritmo genetico

Per lo sviluppo del nostro progetto abbiamo optato per una soluzione con Algoritmo Genetico trovandoci più comodi nella sua implementazione per la suddivisione dei compiti tra i vari membri del team. 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 e Mutazione.

7.1 Selezione

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

- **Roulette Wheel:** Sfruttiamo il valore della funzione di fitness di ogni elemento per impostare la probabilità che venga scelto, tramite un sorteggio randomico di un numero verrà prelevato 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        cumulativeFitness = 0
18        for i in population.getIndividuals():
19            cumulativeFitness += i.fitness()
20            if cumulativeFitness >= value:
21                newIndividuals.append(Individual(*i.getList()))
22                break
23
24    return newIndividuals
25
```

- Rank Selection: Sfrutta il rank di ogni elemento per suddividere una torta di selezione in partizioni proporzionate 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.

```

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 rankSelection(population: Population):
8         user = population.getUser()
9         for i in range(len(population)):
10             evaluate(population[i], user)
11
12         sortedIndividuals = sorted(population.getIndividuals(), key=lambda ind: ind.fitness())
13
14         newPopulation = list()
15         size = len(sortedIndividuals)
16         subdivision = (size * (size+1))/2
17
18         while len(newPopulation) < len(population):
19             newPopulation.append(su.find_rank(sortedIndividuals, subdivision))
20         return sortedIndividuals
21

```

```

1         import random
2 from GA.Individual.exerciseIndividual import Individual
3
4
5 def find_rank(individuals: list[Individual], subdivision: float) -> Individual:
6     num = random.randint(1, 100)
7     prec = 0
8     for i in range(1, len(individuals)):
9         if prec < num <= (100/subdivision)*i:
10             return Individual(*individuals[i].getList())
11         else:
12             prec = (100/subdivision)*i
13

```

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

```

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 randomSelection(population: Population):
8     newP = []
9     while len(newP) < len(population):
10         newP.append(random.choice(population))
11     return newP
12

```

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 metodologie 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[
8     Individual, Individual, Individual]:
9     if len(i1) != len(i2) != len(i3):
10         raise ValueError("Invalid individual length! They have to be the same.")
11     elif n > len(i1):
12         raise ValueError("Invalid n value, must be less than the length of the
13             individual!")
14
15     dividedI1, dividedI2, dividedI3 = divide(i1.getList(), n), divide(i2.getList(),
16         n), divide(i3.getList(), n)
17
18     newList1 = list()
19     newList2 = list()
20     newList3 = list()
21
22     for i, triplet in enumerate(zip(dividedI1, dividedI2, dividedI3)):
23         if i % 3 == 0:
24             newList1.extend(triplet[0])
25             newList2.extend(triplet[1])
26             newList3.extend(triplet[2])
27         elif i % 3 == 1:
28             newList1.extend(triplet[2])
29             newList2.extend(triplet[0])
30             newList3.extend(triplet[1])
31         elif i % 3 == 2:
32             newList1.extend(triplet[1])
33             newList2.extend(triplet[2])
34             newList3.extend(triplet[0])
35
36     i1.setList(newList1)
37     i2.setList(newList2)
38     i3.setList(newList3)
39
40     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.

```
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 nPointReverse(i1: Individual, i2: Individual, i3: Individual, n: int)
8     -> tuple[Individual, Individual, Individual]:
```

```

7     """
8     Performs n-point crossover with perfect interleaving but starting with a
      switch.
9     :param i1: The first individual.
10    :param i2: The second individual.
11    :param i3: The third individual.
12    :param n: Number of points.
13    :return: The two new individuals.
14    """
15    if len(i1) != len(i2) != len(i3):
16        raise ValueError("Invalid individual length! They have to be the same.")
17    elif n > len(i1):
18        raise ValueError("Invalid n value, must be less than the lenght of the
      individual!")
19
20    dividedI1, dividedI2, dividedI3 = divide(i1.getList(), n), divide(i2.getList(),
      n), divide(i3.getList(), n)
21
22    newI1 = []
23    newI2 = []
24    newI3 = []
25
26    for i, triplet in enumerate(zip(dividedI1, dividedI2, dividedI3)):
27        if i % 3 == 0:
28            newI1.extend(triplet[1])
29            newI2.extend(triplet[2])
30            newI3.extend(triplet[0])
31        elif i % 3 == 1:
32            newI1.extend(triplet[0])
33            newI2.extend(triplet[1])
34            newI3.extend(triplet[2])
35        elif i % 3 == 2:
36            newI1.extend(triplet[2])
37            newI2.extend(triplet[0])
38            newI3.extend(triplet[1])
39
40    i1.setList(newI1)
41    i2.setList(newI2)
42    i3.setList(newI3)
43
44    return i1, i2, i3
45

```

- 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.

```

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      def nPointRandom(i1: Individual, i2: Individual, i3: Individual, n:
      int, crossoverProbability: float) -> tuple[Individual, Individual, Individual]
7      """
8      Performs n-point crossover with random interleaving.
9      :param i1: The first individual.
10     :param i2: The second individual.
11     :param i3: The third individual.
12     :param n: Number of points.
13     :param crossoverProbability: The probability of crossover.

```

```

14         :return: The two new individuals.
15         """
16         if len(i1) != len(i2) != len(i3):
17             raise ValueError("Invalid individual length! They have to be the same.")
18         elif n > len(i1):
19             raise ValueError("Invalid n value, must be less than the lenght of the
20             individual!")
21         elif crossoverProbability < 0 or crossoverProbability > 1:
22             raise ValueError("Invalid crossoverProbability! Must be between 0 and 1")
23
24         dividedI1, dividedI2, dividedI3 = divide(i1.getList(), n), divide(i2.getList(),
25         , n), divide(i3.getList(), n)
26
27         newI1 = []
28         newI2 = []
29         newI3 = []
30
31         for triplet in zip(dividedI1, dividedI2, dividedI3):
32             if random() < crossoverProbability:
33                 i = randint(0, 2)
34                 if i % 3 == 0:
35                     newI1.extend(triplet[0])
36                     newI2.extend(triplet[1])
37                     newI3.extend(triplet[2])
38                 elif i % 3 == 1:
39                     newI1.extend(triplet[2])
40                     newI2.extend(triplet[0])
41                     newI3.extend(triplet[1])
42                 elif i % 3 == 2:
43                     newI1.extend(triplet[1])
44                     newI2.extend(triplet[2])
45                     newI3.extend(triplet[0])
46
47         i1.setList(newI1)
48         i2.setList(newI2)
49         i3.setList(newI3)
50
51         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      """
7      Performs n-point crossover with perfect interleaving.
8      :param i1: The first individual.
9      :param i2: The second individual.
10     :param n: Number of points.
11     :return: The two new individuals.
12     """
13     if len(i1) != len(i2):
14         raise ValueError("Invalid individual length! They have to be the same.")
15     elif n > len(i1):
16         raise ValueError("Invalid n value, must be less than the lenght of the
17         individual!")

```

```

17
18     dividedl1, dividedl2 = divide(i1.getList(), n), divide(i2.getList(), n)
19
20     newl1 = list()
21     newl2 = list()
22
23     for i, pair in enumerate(zip(dividedl1, dividedl2)):
24         if i % 2 == 0:
25             newl1.extend(pair[0])
26             newl2.extend(pair[1])
27         else:
28             newl1.extend(pair[1])
29             newl2.extend(pair[0])
30
31     i1.setList(newl1)
32     i2.setList(newl2)
33
34     return i1, i2
35

```

- **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.

```

1  from random import random
2  from GA.Crossover.crossoverUtility import divide
3  from GA.Individual.exerciseIndividual import Individual
4
5  def nPointReverse(i1: Individual, i2: Individual, n: int) -> tuple[Individual,
6  Individual]:
7      """
8      Performs n-point crossover with perfect interleaving but starting with a
9      switch.
10      :param i1: The first individual.
11      :param i2: The second individual.
12      :param n: Number of points.
13      :return: The two new individuals.
14      """
15      if len(i1) != len(i2):
16          raise ValueError("Invalid individual length! They have to be the same.")
17      elif n > len(i1):
18          raise ValueError("Invalid n value, must be less than the length of the
19          individual!")
20
21      dividedl1, dividedl2 = divide(i1.getList(), n), divide(i2.getList(), n)
22
23      newl1 = []
24      newl2 = []
25
26      for i, pair in enumerate(zip(dividedl1, dividedl2)):
27          if i % 2 == 0:
28              newl1.extend(pair[1])
29              newl2.extend(pair[0])
30          else:
31              newl1.extend(pair[0])
32              newl2.extend(pair[1])
33
34      i1.setList(newl1)
35      i2.setList(newl2)
36
37      return i1, i2
38

```

- **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.

```

1 from random import random
2 from GA.Crossover.crossoverUtility import divide
3 from GA.Individual.exerciseIndividual import Individual
4
5 def nPointRandom(i1: Individual, i2: Individual, n: int, crossoverProbability:
6     float) -> tuple[Individual, Individual]:
7     """
8     Performs n-point crossover with random interleaving.
9     :param i1: The first individual.
10    :param i2: The second individual.
11    :param n: Number of points.
12    :param crossoverProbability: The probability of crossover.
13    :return: The two new individuals.
14    """
15    if len(i1) != len(i2):
16        raise ValueError("Invalid individual length! They have to be the same.")
17    elif n > len(i1):
18        raise ValueError("Invalid n value, must be less than the lenght of the
19        individual!")
20    elif crossoverProbability < 0 or crossoverProbability > 1:
21        raise ValueError("Invalid crossoverProbability! Must be between 0 and 1")
22
23    dividedI1, dividedI2 = divide(i1.getList(), n), divide(i2.getList(), n)
24
25    newI1 = []
26    newI2 = []
27
28    for i, pair in enumerate(zip(dividedI1, dividedI2)):
29        if random() < crossoverProbability:
30            newI1.extend(pair[1])
31            newI2.extend(pair[0])
32        else:
33            newI1.extend(pair[0])
34            newI2.extend(pair[1])
35
36    i1.setList(newI1)
37    i2.setList(newI2)
38
39    return i1, i2

```

- **uniformCrossover**: 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 .

```

1 from random import random
2 from GA.Crossover.crossoverUtility import divide
3 from GA.Individual.exerciseIndividual import Individual
4
5 def uniformCrossover(i1: Individual, i2: Individual, crossoverProbability: float)
6     -> tuple[Individual, Individual]:
7     """
8     Performs uniform crossover.
9     :param i1: The first individual.

```



```

9      :param i2: The second individual.
10     :param crossoverProbability: The probability of crossover.
11     :return: The two new individuals.
12     """
13
14     if len(i1) != len(i2):
15         raise ValueError("Invalid individual length! They have to be the same.")
16     elif crossoverProbability < 0 or crossoverProbability > 1:
17         raise ValueError("Invalid crossoverProbability! Must be between 0 and 1")
18
19     dividedI1, dividedI2 = divide(i1.getList(), len(i1)), divide(i2.getList(), len
20                               (i2))
21
22     newI1 = []
23     newI2 = []
24
25     for i, pair in enumerate(zip(dividedI1, dividedI2)):
26         if random() < crossoverProbability:
27             newI1.extend(pair[1])
28         else:
29             newI1.extend(pair[0])
30
31         if random() < crossoverProbability:
32             newI2.extend(pair[0])
33         else:
34             newI2.extend(pair[1])
35
36     i1.setList(newI1)
37     i2.setList(newI2)
38
39     return i1, i2

```

Queste tipologie di Crossover sono eseguite da:

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

```

1  from GA.Population.exercisePopulation import Population
2  from GA.Individual.exerciseIndividual import Individual
3  import GA.Crossover.crossover3Methods as crossover
4  import random
5
6
7  def execute3Crossover(p: Population, crossoverType: crossover, *args) -> list[
8      Individual]:
9      """
10     Executes the crossover on a population taking 3 individuals instead of 2.
11     :param p: The starting population.
12     :param crossoverType: The crossover algorithm we want to perform.
13     :param args: The arguments to pass to the crossover method as needed.
14     :return: The new population.
15     """
16     newIndividuals = list()
17     individuals = p.getIndividuals()
18     size = len(p)
19
20     if size % 3 != 0:
21         raise ValueError("The size of the population must be divisible by 3!")
22     else:
23         for _ in range(round(len(p) / 3)):
24             i1 = random.choice(individuals)

```

```

24         p.removeIndividual(i1)
25         i2 = random.choice(individuals)
26         p.removeIndividual(i2)
27         i3 = random.choice(individuals)
28         i1, i2, i3 = crossoverType(i1, i2, i3, *args)
29         newIndividuals.append(i1)
30         newIndividuals.append(i2)
31         newIndividuals.append(i3)
32
33     return newIndividuals
34

```

- **execute3Crossover**: Questa funzione esegue il Crossover su 3 popolazioni che invece di 2 così da effettuare combinazioni più rilevanti.

```

1
2

```

Tutte queste funzionalità vengono effettuate sfruttando la funzione `divide` che alla ricezione di una lista di esercizi essa viene suddivisa in una lista di liste formate dagli esercizi della lista originale, questa funzione permette la suddivisione della lista in punti che poi saranno usati dal 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 vantaggio che ad uno 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 effettuata richiama la funzione `mutateEX`.

```

1 import random
2 from GA.Population.exercisePopulation import Population
3 import GA.Mutation.mutationUtility as mu
4
5
6 def randomSingleMutation(p: Population, mutationRate: float) -> Population:
7     """
8     Randomly mutates a gene in an Individual in a Population.
9     :param p: The Population to mutate.
10    :param mutationRate: Probability of mutation.
11    :return: The population after mutation.
12    """
13    if random.random() < mutationRate:
14        i = random.choice(p.getIndividuals())
15        p.replaceIndividual(i, mu.mutateEx(i, p))
16
17    return p
18
19

```

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

```

1 import random
2 from GA.Population.exercisePopulation import Population
3 import GA.Mutation.mutationUtility as mu
4
5
6     def randomIndividualMutation(p: Population, mutationRate: float) ->
7         Population:
8         """
9         Randomly mutates an entire Individual in a Population.
10        :param p: The Population to mutate.
11        :param mutationRate: Probability of mutation.
12        :return: The population after mutation.
13        """
14        individuals = p.getIndividuals()
15
16        if random.random() < mutationRate:
17            i = random.choice(individuals)
18            p.replaceIndividual(i, mu.mutateIndividual(i, p))
19
20        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 funzione **mutateIndividual**.

```

1 import random
2 from GA.Population.exercisePopulation import Population
3 import GA.Mutation.mutationUtility as mu
4
5
6     def worstIndividualMutation(p: Population, mutationRate: float) ->
7         Population:
8         """
9         Mutates the worst Individual in a Population.
10        :param p: The Population to mutate.
11        :param mutationRate: Probability of mutation.
12        :return: The population after mutation.
13        """
14        individuals = p.getIndividuals()
15        minF = individuals[0].fitness()
16        indice = 0
17        for index, individual in enumerate(individuals):
18            if minF > individual.fitness():
19                minF = individual.fitness()
20                indice = index
21
22        if random.random() < mutationRate:
23            i = p.getIndividuals()[indice]
24            p.replaceIndividual(i, mu.mutateIndividual(i, p))
25
26        return p

```

Queste 3 tipologie di mutazioni per funzionare utilizzano:

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

```

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
8 def mutateEx(i: Individual, p: Population) -> Individual:
9     """
10     Mutates a random exercise in an Individual.
11     :param i: The Individual to mutate.
12     :param p: The Population of the individual.
13     :return: The new Individual after mutation.
14     """
15     userId = p.getUser().getID()
16     gen = p.getGeneration()
17     oldEx = random.choice(i.getList())
18     newEx = db.select_random_exercise(1, userId)[0]
19     i.replaceExercise(oldEx, newEx, gen)
20     evaluate(i, p.getUser())
21
22     return i
23

```

- 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
8     def mutateIndividual(i: Individual, p: Population):
9         """
10         Mutates the entire Individual.
11         :param i: The Individual to mutate.
12         :param p: The Population of the Individual.
13         :return: The new Individual after mutation.
14         """
15         userId = p.getUser().getID()
16         gen = p.getGeneration()
17         newIndividual = Individual(*db.select_random_exercise(len(i.getList()), userId))
18         evaluate(newIndividual, p.getUser())
19         for ex in newIndividual.getList():
20             ex.setGeneration(gen)
21
22         return newIndividual
23

```