

# PokémonKG: Rappresentazione, CSP, Ricerca e Classificazione in un Sistema a Base di Conoscenza

Componenti del gruppo:

- **Vito Zaza** matr: 719282

Link del progetto: <https://github.com/ZaVit00/PokemonKG>

## Introduzione e sintesi complessiva del progetto

Il progetto ha come obiettivo la realizzazione di un sistema a base di conoscenza (KBS) che integri rappresentazione semantica, ragionamento vincolato, ricerca su spazi di stati e apprendimento automatico.

Il dominio di riferimento è quello dei Pokémon. A tal fine abbiamo adottato e successivamente **esteso** l'ontologia esistente **PokémonKG** (v1.0.0), disponibile all'indirizzo

<https://pokemonkg.org/ontology/version/1.0.0/>.

In coerenza con le buone pratiche e con le indicazioni del testo di riferimento, **non** abbiamo ridefinito un'ontologia da zero: abbiamo invece **riutilizzato** la base di conoscenza esistente, arricchendola con nuove quadruple (infatti pokémonKG utilizza delle quadruple e non delle triple dove il quarto elemento è la fonte da dove i dati provengono) per coprire i compiti richiesti dal progetto.

### *Estensione della Knowledge Base*

L'ontologia PokémonKG, pur risultando completa nel dominio di base (tipi, specie, evoluzioni, mosse), si è rivelata parzialmente insufficiente per i compiti che ci eravamo prefissati e che volevamo modellare.

Per questo motivo, l'ontologia è stata **arricchita semanticamente** attraverso l'aggiunta di nuove quadruple RDF, ottenute da fonti esterne.

Tale arricchimento, tuttavia, non è stato sempre immediato: sebbene le fonti utilizzate, in particolare PokeAPI, fossero strutturate, presentavano **sintassi e modelli concettuali differenti** rispetto a PokémonKG.

Questo ha reso necessario un intervento manuale di mappatura semantica, rallentando il processo e impendendo un processo completamente automatico di arricchimento della conoscenza da fonti strutturate esterne.

Va osservato che, se le fonti esterne avessero adottato la stessa ontologia di riferimento di PokémonKG, il processo di integrazione sarebbe risultato notevolmente più agevole garantendo l'interoperabilità semantica.

Nel capitolo dedicato alla Knowledge Base verranno presentati nel dettaglio i principali arricchimenti introdotti.

Questa estensione ha reso la base di conoscenza **più ricca e funzionale** rispetto agli obiettivi di ragionamento e apprendimento perseguiti nel progetto.

### *Modellazione CSP (Con vincoli rigidi e vincoli flessibili per un problema di ottimizzazione)*

A partire dai Pokémon presenti nella KB e dai loro tipi, sono stati modellati **tre** distinti problemi di **Constraints Satisfaction Problem (CSP)**.

I dettagli formali relativi alla definizione delle variabili, dei domini e dei vincoli saranno approfonditi nel capitolo dedicato.

Per la risoluzione di questi problemi sono state adottate diverse strategie, scelte in funzione della natura del problema e della complessità computazionale (e anche alla volontà di sperimentare tecniche alternative per risolvere lo stesso problema):

- **Ricerca sistematica** nello spazio degli stati, modellando il CSP come un problema di ricerca su grafo (con ricerca sistematica con DFS); I CSP sappiamo dalla teoria che possono essere risolti in questo modo incapsulando nel nodo generico la logica dell'assegnazione di valori alle variabili e con un metodo di ricerca per visitare in modo sistematico l'intero albero;
- **Ricerca locale**, per esplorare in modo efficiente, seppur non in modo completo, lo spazio delle soluzioni, accettando la possibilità di trovare solo ottimi locali senza la garanzia di osservare dei massimi globali (la ricerca locale non offre questa garanzia)
- **Approcci greedy**, per ottenere assegnamenti validi e di buona qualità con un costo computazionale ridotto effettuando delle scelte miopi (sostanzialmente scelte che migliorando SOLO la situazione locale)
- **Solver industriali**, come quelli offerti dalla libreria *Google OR-Tools*, capaci di gestire vincoli combinatori complessi in modo ottimizzato e soprattutto in modo efficiente.

Abbiamo affrontato diversi compiti:

1. **Assegnamento squadre** → un CSP globale che, partendo da tutti i pokémon disponibili nella KG, genera e assegna una squadra **rispettando vincoli globali** imposti sulla base delle statistiche dei pokémon che compongono la squadra.
2. **Ordinamento contro avversari** → un CSP combinatorio che determina l'ordine **ottimale** dei nostri Pokémon contro la squadra avversaria, massimizzando una funzione di vantaggio definita sui moltiplicatori di tipo. A questo livello il ragionamento considera solo i moltiplicatori di tipo, senza entrare nel dettaglio delle mosse per una questione di semplicità (includere le mosse avrebbe reso complicato il processo)
3. **Generazione dei Set di Mosse (quadruple)** -> Per ciascun Pokémon è stato generato un numero prefissato (**num\_set**) di quadruple di mosse candidate, selezionate in base al tipo primario e secondario e alla compatibilità con mosse di altri tipi (es. molti Pokémon di tipo Fuoco possono apprendere mosse di tipo Lotta).

Le informazioni relative alle mosse, incluse statistiche, tipo, categoria, danno, precisione, PP massimi, sono state ottenute direttamente dalla Knowledge Graph, tramite interrogazioni SPARQL.

Data la complessità combinatoria del problema, si è deciso di strutturare la generazione delle mosse su due livelli distinti di CSP che abbiamo denotato come locale e globale:

- **CSP locale**: per ciascun Pokémon, vengono generate alcune quadruple di mosse che soddisfano vincoli rigidi (es. copertura di categorie fisico/speciale, danno minimo complessivo, PP totali, soglia di precisione totale, danno totale). Questo livello produce un insieme di soluzioni candidate valide, da cui poi selezionare nel passo successivo.
- **CSP globale**: una volta ottenute le quadruple candidate per ogni Pokémon, si seleziona un'unica quadrupla per ciascun individuo, con l'obiettivo di massimizzare una funzione di qualità basata su vincoli flessibili (preferenze). Ogni vincolo flessibile è pesato mediante un coefficiente numerico che ne indica l'importanza relativa nella valutazione complessiva della soluzione.

In questo secondo livello, non essendo presenti vincoli rigidi da soddisfare obbligatoriamente, il problema si configura come un CSP di ottimizzazione basato esclusivamente su preferenze, dove l'obiettivo è identificare la configurazione che massimizza la qualità complessiva dell'assegnamento.

### *Ricerca su Spazi di Stati*

Una volta assegnato a ciascun Pokémon un set di mosse, lo **scontro 1-vs-1** è stato modellato come un **problema di ricerca su spazio di stati**, in cui ogni stato rappresenta una configurazione possibile del combattimento.

L'obiettivo è individuare la **sequenza minima di mosse** necessaria a mandare KO l'avversario, considerando una serie di **vincoli dinamici** progettati per rendere la simulazione più realistica e meno banale.

In particolare, sono stati introdotti meccanismi come **PP decrescenti, cure dell'avversario a turni regolari e moltiplicatori di danno progressivi**.

La struttura dello spazio e la presenza di vincoli implicano la necessità di utilizzare **strategie di ricerca sistematica** con **pruning** e gestione degli stati visitati, al fine di evitare cicli o soluzioni inefficaci.

I dettagli formali, gli algoritmi impiegati e le ottimizzazioni adottate saranno approfonditi nel capitolo dedicato.

### *Apprendimento non supervisionato*

Parallelamente, è stato affrontato un compito di apprendimento non supervisionato con K-Means, mirato a raggruppare i Pokémon in Archetipi/SubArchetipi.

- I cluster prodotti sono stati valutati qualitativamente da un esperto, che ha giudicato la clusterizzazione **coerente con i macro-ruoli noti** (attaccanti, tank, bilanciati, ecc.).
- Questo processo ha arricchito ulteriormente la KB con nuove relazioni semantiche.

### *Apprendimento supervisionato*

Il dataset arricchito con archetipi è stato poi utilizzato per compiti di **classificazione supervisionata**, con l'obiettivo di predire l'archetipo di un Pokémon a partire dalle sue feature.

- Sono stati sperimentati diversi modelli (Decision Tree, Random Forest, Regressione Logistica, Dummy Classifier).
- È stata adottata una procedura rigorosa di valutazione di 10-fold-cross validation con medie e deviazione standard delle metriche su più run in modo da ottenere, in genere, una stima del comportamento futuro del nostro modello.
- I risultati hanno mostrato che il sistema è in grado di generalizzare bene e distinguere correttamente i macro-ruoli dei Pokémon, nonostante la difficoltà intrinseca del compito.

## Pokemon KG: Una knowledge Graph basato sul dominio di pokémon

Come detto, il progetto si fonda sull'utilizzo dell'ontologia **PokémonKG** (versione 1.0.0), disponibile all'indirizzo <https://pokemonkg.org/ontology/version/1.0.0/>.

Tale ontologia, sviluppata secondo i linguaggi standard **OWL** e **RDF**, si propone di rappresentare formalmente il dominio dei Pokémons, includendo entità come specie, tipi, mosse, evoluzioni e altre informazioni strutturali.

Per importare la KG, bisogna scaricare il file “poke-a.nq” una notazione che aderisce al modello di dati RDF. Ciascun file viene poi importato manualmente all'interno di un repository dedicato in **GraphDB**, un triplestore conforme a OWL 2.

L'uso di GraphDB ha permesso di eseguire interrogazioni SPARQL, navigare la base di conoscenza ed esplorare sia le classi ontologiche che le relazioni istanziate.

Come tipico delle ontologie OWL, PokémonKG include sia una **componente terminologica (TBox)** che una **componente asserzionale (ABox)**.

- La **TBox** definisce il vocabolario concettuale: classi come Species, Move, SpecialMove, StatusMove etc, e proprietà come hasType, hasMove etc. Questa parte fornisce lo schema semantico su cui si basa il grafo che sostanzialmente corrisponde all'ontologia, cioè la specifica formale dei simboli utilizzati;
- La **ABox**, invece, contiene le istanze effettive, ovvero i Pokémons concreti e le loro relazioni: ad esempio, l'indicazione che *Charizard* è un Pokémon di tipo *Fuoco*, che può usare la mossa *Flamethrower* espressa tramite quadruple soggetto-verbo-oggetto-origine del dato (da dove è stato preso il dato. Questo quarto elemento è chiamato named graph nella KG)

### Arricchimento semantico della base di conoscenza

L'ontologia originale PokémonKG, pur offrendo una base concettuale ben strutturata, si è rivelata **incompleta** per soddisfare pienamente i requisiti del nostro progetto.

Per questo motivo, è stato realizzato un **intervento di arricchimento semantico** che ha coinvolto entrambe le componenti principali di un'ontologia OWL (cioè ABox e TBox)

Questo arricchimento ha seguito un approccio formale e controllato, attraverso la definizione di:

- nuove proprietà ontologiche (DatatypeProperty, ObjectProperty) inserite nella TBox
- nuove quadruple RDF (N-Quads) contenenti asserzioni nella ABox,
- grafi nominati che distinguono le fonti dei dati importati (es. bulbapedia, pokeapi, archetipi-clustering),
- collegamenti semantici esplicativi con entità esterne tramite owl:sameAs per risolvere bug nella base di conoscenza.

Le fonti utilizzate per l'arricchimento sono:

- **PokeAPI** per completare i metadati delle mosse,
- **Bulbapedia** per la tabella dei moltiplicatori tra tipi,
- un **dataset CSV derivato dal clustering** per assegnare archetipi e sub-archetipi ai Pokémons.

Nei paragrafi successivi, verranno descritti nel dettaglio i tre macro-interventi eseguiti:

### **Primo intervento => Arricchimento delle Mosse con danni, precisione, pp**

L'ontologia originale PokémonKG include già una rappresentazione strutturata (e ben fatta) delle mosse (classificate come PhysicalMove, SpecialMove, StatusMove) e in molti casi fornisce anche parametri quantitativi come la potenza (basePower), la precisione (accuracy) e i punti azione (basePowerPoints o anche chiamati PP nei giochi pokémon).

Tuttavia, durante l'esplorazione del grafo tramite GraphDB, è emerso che **numerose mosse erano incomplete**, ovvero **prive di uno o più di questi valori**. Altre mosse invece, avevano questi parametri settati o in alcuni casi per mosse con accuracy sempre garantita, questa non era specificata nella KG risultando in problemi. Questo per noi si è rivelato un grossissimo problema perché la parte dei CSP si basa anche su questi parametri.

Per identificare tali casi (o anomalie), è stata utilizzata la seguente query SPARQL, che seleziona tutte le mosse incomplete (cioè mosse mancanti di uno o più valori (almeno una tra quelle indicate)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX pokemonkg: <https://pokemonkg.org/ontology#>
SELECT DISTINCT ?move ?typeClass
  (BOUND(?power) AS ?hasPower)
  (BOUND(?accuracy) AS ?hasAccuracy)
  (BOUND(?pp) AS ?hasPP)
WHERE {
?move rdf:type ?typeClass .
VALUES ?typeClass {
  pokemonkg:SpecialMove
  pokemonkg:PhysicalMove
  pokemonkg>StatusMove
}
OPTIONAL { ?move pokemonkg:basePower ?power. }
OPTIONAL { ?move pokemonkg:accuracy ?accuracy. }
OPTIONAL { ?move pokemonkg:basePowerPoints ?pp. }
FILTER( !BOUND(?power) || !BOUND(?accuracy) || !BOUND(?pp) )
}
ORDER BY ?typeClass ?move
```

Una volta individuate le mosse incomplete, per ciascuna di esse è stato effettuato un recupero automatico dei metadati mancanti tramite [PokeAPI](#), accedendo via API http con script Python (**main\_aricchimento\_mosse.py**) al relativo endpoint di PokeAPI.

Questa operazione non è stata semplice perché le URI delle mosse dentro PokémonKG, non corrispondono fedelmente ai nomi delle mosse in PokeAPI (come d'altronde è lecito aspettarsi).

Ad esempio la mossa <https://pokemonkg.org/instance/move/ice-shard> corrisponde alla mossa ice-shard di PokeAPI. Quindi in generale, si è osservato come la parte finale dell'URI corrispondeva al nome della mossa di PokeAPI. Il nome della mossa è cruciale essendo un parametro da inserire nella URL per ottenere i metadati della mossa es: <https://pokeapi.co/api/v2/move/flamethrower>

Tuttavia, per alcune mosse si è reso necessario, comunque, un intervento manuale a causa di una asimmetria da nomi delle mosse in PokeAPI e uri normalizzate delle mosse di PokemonKG.

In particolare nel processo di arricchimento: per ogni mossa, se uno dei tre valori (power, accuracy, pp) risultava mancante, vengono generate delle quadruple (per aderire al formato di pokmenon KG) e indicando come quarto parametro la fonte ovvero PokeAPI. In taluni casi è stato necessario impostare un valore a 0 per evitare NONE.

Il risultato di questo processo automatico (e manuale successivamente per mosse difficili da trovare) è stato salvato nel file:

**“mosse\_aricchite.nq”**

che contiene le quadruple RDF arricchite, nel formato N-Quads, pronte per essere ricaricate nel grafo via GraphDB con fonte inserita. Le prime 10 quadruple da mosse\_aricchite.nq:

```
<https://pokemonkg.org/instance/move/acrobatics> <https://pokemonkg.org/ontology#accuracy>
"100"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/acrobatics> <https://pokemonkg.org/ontology#basePower>
"55"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/acrobatics> <https://pokemonkg.org/ontology#basePowerPoints>
"15"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/air-cutter> <https://pokemonkg.org/ontology#accuracy>
"95"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/air-cutter> <https://pokemonkg.org/ontology#basePower>
"60"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/air-cutter> <https://pokemonkg.org/ontology#basePowerPoints>
"25"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/air-slash> <https://pokemonkg.org/ontology#basePowerPoints>
"15"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/amnesia> <https://pokemonkg.org/ontology#accuracy>
"100"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .

<https://pokemonkg.org/instance/move/ancient-power> <https://pokemonkg.org/ontology#basePowerPoints>
"5"^^<http://www.w3.org/2001/XMLSchema#integer> <https://pokeapi.co/> .
```

Una osservazione importante: in questo caso non è stato necessario modificare la **TBox**, poiché le proprietà basePower, accuracy e basePowerPoints erano già definite all'interno dell'ontologia originale di PokémonKG. L'intervento si è quindi limitato a **popolare la ABox** con i valori mancanti, mantenendo la coerenza semantica del grafo.

### Correzioni semantiche via owl:sameAs

Durante la fase di arricchimento, sono emerse alcune incongruenze o duplicazioni concettuali nella parte asserzionale (Abox) del grafo originale. In particolare, alcuni URI di mosse risultavano **duplicati** differendo solo per la parte finale dell'URI (cioè il nome della mossa) ad esempio poisonpowder e poison-powder in realtà sono il medesimo concetto.

Questo errore è probabilmente dipeso da a scraping automatico di fonti sul web per ottenere statistiche delle mosse.

Per correggere questi casi, è stato necessario creare un file dedicato:

**“mosse\_sameas.nq”**

contenente **triple owl:sameAs** per riallineare concetti che fanno riferimento alla stessa entità ma sono rappresentati con URI differenti. Di seguito si riportano le sei mosse individuate come duplicati.

```
<https://pokemonkg.org/instance/move/poisonpowder> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/poison-powder> <https://pokemonkg.org/dataset/manual-alignment> .

<https://pokemonkg.org/instance/move/grasswhistle> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/grass-whistle> <https://pokemonkg.org/dataset/manual-alignment> .

<https://pokemonkg.org/instance/move/featherdance> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/feather-dance> <https://pokemonkg.org/dataset/manual-alignment> .

<https://pokemonkg.org/instance/move/conversion2> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/conversion-2> <https://pokemonkg.org/dataset/manual-alignment> .

<https://pokemonkg.org/instance/move/sonicboom> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/sonic-boom> <https://pokemonkg.org/dataset/manual-alignment> .

<https://pokemonkg.org/instance/move/smellingsalt> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/smelling-salt> <https://pokemonkg.org/dataset/manual-alignment> .

<https://pokemonkg.org/instance/move/faint-attack> <http://www.w3.org/2002/07/owl#sameAs>
<https://pokemonkg.org/instance/move/feint-attack> <https://pokemonkg.org/dataset/manual-alignment> .
```

Infatti, grazie all'utilizzo del motore inferenziale OWL e del predicato owl:sameAs, questi concetti apparentemente distinti sono stati riconosciuti come equivalenti durante l'esecuzione delle query e del reasoning, garantendo una maggiore uniformità semantica e migliorando l'affidabilità delle interrogazioni SPARQL.

Importante: Non abbiamo previsto uno script automatico per caricare i file su GraphDB, pertanto, abbiamo semplicemente deciso di caricare manualmente i file importandoli tramite la finestra grafica di GraphDB. Una volta importati i file delle mosse possiamo eseguire questa query SPARQL per popolare le strutture dati del nostro sistema basato su conoscenza.

Si mostra la query utilizzata per ottenere i dati delle mosse. Una volta che le mosse sono state arricchite semanticamente non abbiamo problemi di nessun tipo per i tre parametri numerici.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX pokemonkg: <https://pokemonkg.org/ontology#>

SELECT DISTINCT ?move ?basePower ?accuracy ?pp ?moveType ?catMove
WHERE {
  ?move rdf:type pokemonkg:Move .
  ?move pokemonkg:basePower ?basePower .
  ?move pokemonkg:accuracy ?accuracy .
  ?move pokemonkg:basePowerPoints ?pp .
  ?move pokemonkg:hasType ?moveType .
  ?move rdf:type ?catMove .
  VALUES ?catMove { pokemonkg:PhysicalMove pokemonkg:SpecialMove pokemonkg>StatusMove }
}
ORDER BY ?move
```

## Modellazione dei moltiplicatori di danno tra tipi

L'ontologia originale PokéMonKG non includeva alcuna rappresentazione strutturata **dell'efficacia relativa dei tipi, ovvero quanto un tipo di mossa sia efficace contro un tipo difensivo** (il classico moltiplicatore dei giochi pokémon: 0.5, 1, 1.5, 2.0)

Si tratta di un'informazione cruciale per i compiti di ragionamento nel dominio PokéMon (es. calcolo vantaggi, assegnamento strategico delle mosse, valutazione CSP).

Per colmare questa lacuna è stato realizzato un arricchimento sia della TBox che della ABox.

### *Estensione della TBox – DamageMultiplierRelation*

Per modellare il concetto di “relazione di moltiplicatore di danno”, è stata definita una nuova classe ontologica OWL chiamata:

```
pokemonkg:DamageMultiplierRelation rdf:type owl:Class ;  
    rdfs:label "Relazione di Moltiplicatore di Danno"@it ;  
    rdfs:comment "Modella l'efficacia di una mossa di un certo tipo contro un tipo difensivo"@it .
```

A questa classe sono state associate tre nuove proprietà ontologiche, anch'esse definite nella TBox all'interno del file **schema\_moltiplicatori.ttl**:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .  
@prefix owl: <http://www.w3.org/2002/07/owl#> .  
@prefix pokemonkg: <https://pokemonkg.org/ontology#> .  
@prefix pkgj: <https://pokemonkg.org/instance/> .
```

```
#####  
# Classe per la relazione di moltiplicatore di danno  
#####  
pokemonkg:DamageMultiplierRelation rdf:type owl:Class ;  
    rdfs:label "Relazione di moltiplicatore di danno tra tipi"@it .
```

```
#####  
# Proprietà oggetto  
# Nota: i valori di queste proprietà sono URI di tipi predefiniti  
# (namespace PokéType_*), non istanze di una classe.  
# Pertanto, non viene specificato un rdfs:range formale.  
#####  
pokemonkg:moveType rdf:type owl:ObjectProperty ;  
    rdfs:domain pokemonkg:DamageMultiplierRelation ;  
    rdfs:label "Tipo della mossa"@it .
```

```
pokemonkg:targetType rdf:type owl:ObjectProperty ;  
    rdfs:domain pokemonkg:DamageMultiplierRelation ;
```

```

rdfs:label "Tipo del bersaglio"@it .

#####
# Proprietà dato per il moltiplicatore
#####
pokemonkg:damageMultiplierValue rdf:type owl:DatatypeProperty ;
    rdfs:domain pokemonkg:DamageMultiplierRelation ;
    rdfs:range xsd:decimal ;
    rdfs:label "Valore del moltiplicatore"@it .

```

Questa estensione ha permesso di modellare ogni **relazione di efficacia tipo→tipo** come un'entità reificata, arricchita da metadati.

### Popolamento della ABox – N-Quads tabella\_moltiplicatori.nq

I dati concreti sono stati estratti da una tabella CSV  $18 \times 18$  ottenuta da fonti ufficiali (Bulbapedia) e trasformati in quadruple RDF tramite lo script main\_tabella\_moltiplicatori.py (quindi creata, salvata in locale e trasformata con codice python). Per assicurare la correttezza sintattica non abbiamo usato i nomi dei tipi ma le URI direttamente.

Ogni cella della matrice è stata convertita in un'entità DamageMultiplierRelation con URI univoco. Un esempio del contenuto del file:

Il file risultante è:

**tabella\_moltiplicatori.nq**

contenente tutte le 324 relazioni tra i tipi ( $18 \times 18$ ), caricate come quadruple RDF all'interno di GraphDB nel grafo.

```

https://pokemonkg.org/instance/damageRel_005> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <https://pokemonkg.org/ontology#DamageMultiplierRelation>
<https://pokemonkg.org/dataset/bulbapedia> .
<https://pokemonkg.org/instance/damageRel_005>
<https://pokemonkg.org/ontology#moveType>
<https://pokemonkg.org/ontology#PokéType:Normal>
<https://pokemonkg.org/dataset/bulbapedia> .
<https://pokemonkg.org/instance/damageRel_005>
<https://pokemonkg.org/ontology#targetType>
<https://pokemonkg.org/ontology#PokéType:Ground>
<https://pokemonkg.org/dataset/bulbapedia> .
<https://pokemonkg.org/instance/damageRel_005>
<https://pokemonkg.org/ontology#damageMultiplierValue>
"1.0"^^<http://www.w3.org/2001/XMLSchema#decimal>
<https://pokemonkg.org/dataset/bulbapedia> .

```

Una volta caricato a mano questo file su GraphDB, le relazioni sono interrogabili tramite SPARQL. Un esempio di query per recuperare tutti i moltiplicatori moveType → targetType è il seguente. Questa query in realtà è proprio necessaria per popolare la struttura dati interna al sistema basata su conoscenza.

```
# Definizione dei prefissi per abbreviare le URI usate
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX pokemonkg: <https://pokemonkg.org/ontology#>

# Selezione dei tipi di attacco, tipi bersaglio e valori di moltiplicatore
SELECT ?attackerType ?defenderType ?multiplier
WHERE {
  # Individua le relazioni che rappresentano il moltiplicatore di danno tra tipi
  # DamageMultiplierRelation è una classe
  ?rel rdf:type pokemonkg:DamageMultiplierRelation ;
    # Tipo di mossa che attacca
    pokemonkg:moveType ?attackerType ;
    # Tipo del bersaglio difensore
    pokemonkg:targetType ?defenderType ;
    # Valore numerico del moltiplicatore di danno
    pokemonkg:damageMultiplierValue ?multiplier .
}
```

## *Introduzione di Archetipi e SubArchetipi*

Per migliorare la descrizione semantica dei Pokémon all'interno del grafo, è stato introdotto un ulteriore livello di astrazione basato su Archetipi comportamentali e relative Subcategorie (Offensivo, Tank, Bilanciato/Balanced, Elite/Leggendario) e Tank Fisico/ Speciale e, Offensivo Speciale/ Fisico. Questo arricchimento ha richiesto un intervento sia sulla componente terminologica (TBox) dell'ontologia, sia sulla componente asserzionale (ABox).

Estensione della TBox: nuove proprietà OWL. Nella TBox sono state definite due nuove proprietà di tipo owl:DatatypeProperty (quindi proprietà tra due individui) (DataType si contrappone ad ObjectType)

```
@prefix pokemonkg: <https://pokemonkg.org/ontology#> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .  
  
pokemonkg:hasArchetype rdf:type owl:DatatypeProperty ;  
    rdfs:domain pokemonkg:Pokemon ;  
    rdfs:range xsd:string ;  
    rdfs:label "Archetipo"@it ;  
    rdfs:comment "Archetipo comportamentale del Pokémon appreso via clustering"@it .  
  
pokemonkg:hasSubArchetype rdf:type owl:DatatypeProperty ;  
    rdfs:domain pokemonkg:Pokemon ;  
    rdfs:range xsd:string ;  
    rdfs:label "Subarchetipo"@it ;  
    rdfs:comment "Subcategoria dell'archetipo, appresa via clustering"@it .
```

Queste proprietà sono ora parte integrante dell'ontologia e sono state dichiarate nel file [\*archetipi\\_pokemon.ttl\*](#).

Popolamento della ABox: inserimento delle quadruple: **I dati relativi agli Archetipi e SubArchetipi sono stati ottenuti tramite clustering non supervisionato**, successivamente **validati da un esperto umano** e salvati in un file CSV. Il file main\_arricchimento\_archetipi.py si occupa di:

1. Leggere il CSV con i risultati del clustering;
2. Associare ogni Pokémon al proprio URI nella KG;
3. Generare quadruple RDF come in figura;
4. Scrivere tutte le quadruple RDF nel file Archetipi\_pokemon.nq.

```
<https://pokemonkg.org/instance/pokemon/lucario>  
<https://pokemonkg.org/ontology#hasArchetype>  
"Attaccante"^^<http://www.w3.org/2001/XMLSchema#string>  
<https://pokemonkg.org/dataset/archetipi-clustering> .  
  
<https://pokemonkg.org/instance/pokemon/lucario>  
<https://pokemonkg.org/ontology#hasSubArchetype>
```

"Veloce"^^<<http://www.w3.org/2001/XMLSchema#string>> <<https://pokemonkg.org/dataset/archetipi-clustering>> .

Durante il processo è stato eseguito anche un controllo di consistenza tra i Pokémon presenti nel CSV e quelli effettivamente presenti nella KG. Le eventuali discrepanze sono state loggiate a scopo diagnostico e in futuro migliorabili. Un esempio di contenuto del file Archetipi\_pokemon.nq che verrà importato manualmente dentro GraphDB

Pokémon URI	Predicato	Valore	Fonte
<.../lucario>	hasArchetype	"Attaccante"	< <a href="https://pokemonkg.org/dataset/archetipi-clustering">https://pokemonkg.org/dataset/archetipi-clustering</a> >
<.../lucario>	hasSubArchetype	"Attaccante Speciale"	< <a href="https://pokemonkg.org/dataset/archetipi-clustering">https://pokemonkg.org/dataset/archetipi-clustering</a> >
<.../blissey>	hasArchetype	"Tank"	< <a href="https://pokemonkg.org/dataset/archetipi-clustering">https://pokemonkg.org/dataset/archetipi-clustering</a> >
<.../blissey>	hasSubArchetype	"Tank Speciale"	< <a href="https://pokemonkg.org/dataset/archetipi-clustering">https://pokemonkg.org/dataset/archetipi-clustering</a> >

Con la seguente query (prima non possibile) ora possiamo ottenere gli archetipi dei pokémon

```
PREFIX poke: <https://pokemonkg.org/ontology#>

SELECT ?pokemon ?archetipo ?subarchetipo

WHERE {
  ?pokemon poke:hasArchetype ?archetipo .
  OPTIONAL { ?pokemon poke:hasSubArchetype ?subarchetipo }}
```

35	pkgi:pokemon/exeggute	"Balanced"	
36	pkgi:pokemon/exeggutor	"Attaccante"	"Attaccante Speciale"
37	pkgi:pokemon/fearow	"Attaccante"	"Attaccante Fisico"
38	pkgi:pokemon/flareon	"Attaccante"	"Attaccante Fisico"
39	pkgi:pokemon/gastly	"Balanced"	
40	pkgi:pokemon/gengar	"Attaccante"	"Attaccante Speciale"
41	pkgi:pokemon/geodude	"Balanced"	
42	pkgi:pokemon/gloom	"Balanced"	
43	pkgi:pokemon/golbat	"Attaccante"	"Attaccante Fisico"
44	pkgi:pokemon/goldeen	"Balanced"	
45	pkgi:pokemon/golduck	"Attaccante"	"Attaccante Speciale"
46	pkgi:pokemon/golem	"Tank"	"Tank Fisico"
47	pkgi:pokemon/graveler	"Balanced"	

## Constraint Satisfaction Problem

### Primo problema -> Generazione della Squadra Personale (CSP)

**La prima fase del processo di modellazione tramite CSP riguarda la generazione di squadre Pokémon personali**, dove ogni membro della squadra è scelto in modo da rispettare vincoli strategici legati alla varietà e compatibilità dei tipi. Questo problema è stato formulato come un problema di soddisfacimento di vincoli (CSP), in cui:

- Le variabili sono gli slot della squadra ( $\text{slot}_0, \text{slot}_1, \dots, \text{slot}_5$ ).
- I domini sono insiemi di Pokémon compatibili con i tipi desiderati per ciascuno slot.
- I vincoli garantiscono la diversità tipologica e impediscono duplicati.

Generare squadre Pokémon con i seguenti vincoli globali

- almeno 3 tipi primari distinti;
- massimo 2 Pokémon con lo stesso tipo primario;
- almeno 2 Pokémon con doppio tipo;
- tutti i Pokémon devono essere diversi;
- rispetto tra tipo primario richiesto e, optionalmente, tipo secondario ammesso per ciascuno slot.

La squadra è generata tramite il metodo:

```
GeneratoreSquadre.genera_squadra_personale(  
    tipi_strategici=TipoPokemonHelper.genera_tipi_strategici([...]),  
    squadra_precedente=set()  
)
```

Il parametro `tipi_strategici` è una mappa che associa a ciascun tipo primario desiderato un insieme (eventualmente vuoto) di tipi secondari ammessi. La struttura permette un controllo dettagliato delle combinazioni di tipi, ad esempio:

```
{  
    "Fire": {"Fighting", "Flying"},  
    "Water": set(),  
}
```

L'algoritmo si comporta come segue:

1. Filtra tutti i Pokémon compatibili con i tipi richiesti, escludendo quelli già presenti nella squadra avversaria.
2. Costruisce un CSP con sei variabili ( $\text{slot}_0 \dots \text{slot}_5$ ), ciascuna con dominio ridotto secondo i tipi compatibili. Questa è una scelta strategica ma rischiosa: Assegnare un dominio di certe

mosse per un determinato slot riduce sicuramente il tempo necessario per trovare la soluzione ma ci pone nel rischio di non riuscire mai a trovare una soluzione. Ecco perché cicliamo per 10 volte fino a trovare una soluzione. In questo caso non ci sono vincoli flessibili ma solo vincoli rigidi.

### 3. Applica i seguenti vincoli:

- **AllDifferentConstraint:** tutti i Pokémon devono essere distinti (come è logico aspettarsi)
- **Vincolo di diversità globale:** minimo 3 tipi primari, massimo 2 Pokémon con stesso tipo, almeno 2 con doppio tipo.

Il file Python **generazione\_squadra.py** contiene la classe GeneratoreSquadre che incapsula tutta la logica di filtraggio e risoluzione CSP. In particolare:

- Il metodo `_filtra_pokemon_per_tipo` costruisce i domini delle variabili CSP filtrando i Pokémon secondo il tipo primario richiesto e gli eventuali tipi secondari ammessi.
- Il metodo `_vincoli_generali_squadra` impone vincoli globali di diversità.
- È previsto anche un metodo `genera_squadra_capo_palestra` per creare squadre a tema, basate su un unico tipo, ma che richiedano almeno 2 Pokémon con doppio tipo; Attualmente era previsto in passato ma ora è inutilizzato.

Nell'esperimento sono stati testati vincoli come:

```
# Slot 0: Pokémon con tipo primario ma senza secondario, ..... # Slot 5: Pokémon con tipo secondario tra 5 opzioni (stessa logica per tutti) (i tipi vengono estratti in modo casuale per garantire un certo grado di randomicità tra le squadre prodotte dall'algoritmo)
```

```
TipPokemonHelper.genera_tipi_strategici([0, 0, 1, 2, 3, 4])
```

In questo modo si guida il solver verso una squadra varia, evitando configurazioni monotone o troppo simili tra loro.

```
--- Squadra personale 1 ---
pokemon_0: https://pokemonq.org/instance/pokemon/thundurus → tipi: ontology#PokéType:Flying/ontology#PokéType:Electric
pokemon_1: https://pokemonq.org/instance/pokemon/morgrem → tipi: ontology#PokéType:Dark/ontology#PokéType:Fairy
pokemon_2: https://pokemonq.org/instance/pokemon/silicobra → tipi: ontology#PokéType:Ground
pokemon_3: https://pokemonq.org/instance/pokemon/latios → tipi: ontology#PokéType:Psychic/ontology#PokéType:Dragon
pokemon_4: https://pokemonq.org/instance/pokemon/steelix → tipi: ontology#PokéType:Steel/ontology#PokéType:Ground
pokemon_5: https://pokemonq.org/instance/pokemon/turtonator → tipi: ontology#PokéType:Dragon/ontology#PokéType:Fire

--- Squadra personale 2 ---
pokemon_0: https://pokemonq.org/instance/pokemon/lanturn → tipi: ontology#PokéType:Water/ontology#PokéType:Electric
pokemon_1: https://pokemonq.org/instance/pokemon/aegislash → tipi: ontology#PokéType:Ghost/ontology#PokéType:Steel
pokemon_2: https://pokemonq.org/instance/pokemon/talonflame → tipi: ontology#PokéType:Flying/ontology#PokéType:Fire
pokemon_3: https://pokemonq.org/instance/pokemon/rookidee → tipi: ontology#PokéType:Flying
pokemon_4: https://pokemonq.org/instance/pokemon/zamazenta → tipi: ontology#PokéType:Steel/ontology#PokéType:Fighting
pokemon_5: https://pokemonq.org/instance/pokemon/thievul → tipi: ontology#PokéType:Dark
```

## Problema due => Ordinamento della squadra per il match-up

Una volta generate le due squadre Pokémon, è stato necessario affrontare il problema di ordinare strategicamente i nostri Pokémon in modo da massimizzare l'efficacia contro gli avversari, basandosi esclusivamente sui tipi (senza ancora considerare le mosse effettive, che verranno assegnate successivamente).

Obiettivo del problema: Dato un ordine fisso dei Pokémon avversari, si desidera determinare l'assegnamento ottimale dei nostri sei Pokémon agli slot da 0 a 5, tale da massimizzare la somma dei moltiplicatori di danno (derivati dal confronto tra tipi).

Formalizzazione come CSP: Il problema è stato modellato come un problema di soddisfacibilità con ottimizzazione, con le seguenti caratteristiche:

- Variabili: ciascuno dei sei slot da riempire;
- Dominio: tutti i nostri Pokémon disponibili (senza ripetizioni);
- Vincoli: ogni Pokémon può essere assegnato a un solo slot (vincolo di *tutti diversi*);
- Funzione obiettivo: massimizzare la somma dei punteggi ottenuti da ogni assegnamento (moltiplicatori di tipo).

Matrice dei punteggi di efficacia

Cella [i][j] = max\_eff(nostro\_i → avversario\_j) - max\_eff(avversario\_j → nostro\_i)

Valore > 0 → vantaggio nostro | Valore < 0 → vantaggio avversario

	lanturn	aegislash	talonflame	rookidee	zamazenta	thievul
thundurus	0	0	1	1.5	0.5	0
morgrem	0	0	0	0	-1	1.75
silicobra	0	1	-1	-1	1	0
latios	0.5	-1	0	0	0	-1
steelix	0	1.5	-1.5	0.5	0	0
turtonator	0	1	0	0	1	0

## Soluzione con OR-Tools

Per risolvere il problema in modo efficiente, è stato adottato Google OR-Tools, una suite open-source di strumenti di ottimizzazione a livello industriale. In particolare, è stato utilizzato il modulo CP-SAT Solver, progettato per problemi di soddisfacibilità con vincoli e obiettivi lineari.

L'utilizzo di OR-Tools ha permesso di:

- Definire formalmente le variabili, i vincoli e la funzione obiettivo;

- Eseguire una ricerca sistematica ottimale, garantendo la massima qualità della soluzione;
- Ottenere soluzione in tempi computazionali contenuti, considerando che in questo problema le possibili assegnazioni sono  $6! = 720$  permutazioni possibili.

### Approccio comparativo con algoritmo greedy

Parallelamente, è stato implementato anche un algoritmo greedy, che assegna iterativamente il miglior Pokémon disponibile per ciascuno slot avversario. Sebbene SEMPRE non ottimale, si è rivelato un utile baseline euristico per il confronto.

File coinvolti

- problema\_scontro.py: costruzione della matrice dei punteggi tra squadre.
- solver\_scontro.py: implementazione dell'algoritmo di assegnamento ottimale con OR-Tools (assegnamento ottimale) e della strategia greedy (assegnamento\_greedy).

I moltiplicatori di tipo utilizzati nella costruzione della matrice di punteggi derivano direttamente dalla Knowledge Graph, dove sono stati inseriti manualmente nella sezione "Arricchimento: Moltiplicatori").

```
--- MATCHUP OTTIMALE ---
Avversario 0: lanturn   ↔ Nostro latios (score = 0.50 -> Moderatamente avvantaggiato)
Avversario 1: aegislash   ↔ Nostro steelix (score = 1.50 -> Fortemente avvantaggiato)
Avversario 2: talonflame   ↔ Nostro turtonator (score = 0.00 -> Neutro)
Avversario 3: rookidee   ↔ Nostro thundurus (score = 1.50 -> Fortemente avvantaggiato)
Avversario 4: zamazenta   ↔ Nostro silicobra (score = 1.00 -> Fortemente avvantaggiato)
Avversario 5: thievul   ↔ Nostro morgrem (score = 1.75 -> Fortemente avvantaggiato)
Punteggio totale punteggio assegnamento ottimale: 6.25
```

```
--- MATCHUP GREEDY ---
Avversario 0: lanturn   ↔ Nostro latios (score = 0.50 -> Moderatamente avvantaggiato)
Avversario 1: aegislash   ↔ Nostro steelix (score = 1.50 -> Fortemente avvantaggiato)
Avversario 2: talonflame   ↔ Nostro thundurus (score = 1.00 -> Fortemente avvantaggiato)
Avversario 3: rookidee   ↔ Nostro morgrem (score = 0.00 -> Neutro)
Avversario 4: zamazenta   ↔ Nostro silicobra (score = 1.00 -> Fortemente avvantaggiato)
Avversario 5: thievul   ↔ Nostro turtonator (score = 0.00 -> Neutro)
Punteggio totale punteggio assegnamento greedy: 4.00
```

Avversario	assegnamento_ottimale	assegnamento_greedy	Differenza assegnamento_greedy - assegnamento_ottimale
lanturn	latios	latios	+0.00
aegislash	steelix	steelix	+0.00
talonflame	turtonator	thundurus	+1.00
rookidee	thundurus	morgrem	-1.50
zamazenta	silicobra	silicobra	+0.00
thievul	morgrem	turtonator	-1.75
-----	-----	-----	-----
Totale	6.25	4.00	-2.25

--- Squadra 1 ordinata secondo assegnamento ottimale ---

pokemon\_0: <https://pokemonkg.org/instance/pokemon/latios> → tipi: ontology#PokéType:Psychic/ontology#PokéType:Dragon  
 pokemon\_1: <https://pokemonkg.org/instance/pokemon/steelix> → tipi: ontology#PokéType:Steel/ontology#PokéType:Ground  
 pokemon\_2: <https://pokemonkg.org/instance/pokemon/turtonator> → tipi: ontology#PokéType:Dragon/ontology#PokéType:Fire  
 pokemon\_3: <https://pokemonkg.org/instance/pokemon/thundurus> → tipi: ontology#PokéType:Flying/ontology#PokéType:Electric  
 pokemon\_4: <https://pokemonkg.org/instance/pokemon/silicobra> → tipi: ontology#PokeType:Ground  
 pokemon\_5: <https://pokemonkg.org/instance/pokemon/morgrem> → tipi: ontology#PokéType:Dark/ontology#PokéType:Fairy

--- Squadra 1 ordinata secondo assegnamento greedy ---

pokemon\_0: <https://pokemonkg.org/instance/pokemon/latios> → tipi: ontology#PokéType:Psychic/ontology#PokéType:Dragon  
 pokemon\_1: <https://pokemonkg.org/instance/pokemon/steelix> → tipi: ontology#PokéType:Steel/ontology#PokéType:Ground  
 pokemon\_2: <https://pokemonkg.org/instance/pokemon/thundurus> → tipi: ontology#PokéType:Flying/ontology#PokéType:Electric  
 pokemon\_3: <https://pokemonkg.org/instance/pokemon/morgrem> → tipi: ontology#PokéType:Dark/ontology#PokeType:Fairy  
 pokemon\_4: <https://pokemonkg.org/instance/pokemon/silicobra> → tipi: ontology#PokéType:Ground  
 pokemon\_5: <https://pokemonkg.org/instance/pokemon/turtonator> → tipi: ontology#PokéType:Dragon/ontology#PokéType:Fire

```

2025-09-10 18:10:02,408 [DEBUG] Starting new HTTP connection (1): localhost:7200
2025-09-10 18:10:03,087 [DEBUG] http://localhost:7200 "POST /repositories/pokemonKG HTTP/1.1" 200 None
  
```

Ovviamente per scegliere l'ordinamento migliore abbiamo optato per la soluzione prodotta da or tools essendo quella che ha la certezza di massimizzare la funzione di qualità definita internamente.

## Problema tre => Modellazione CSP Locale per l'Assegnamento delle Mosse

Una volta ordinate le squadre tramite il match-up basato sui moltiplicatori di tipo, viene avviata la **fase di generazione dei set di mosse (quadruple) per ogni Pokémon della squadra personale** (la squadra avversaria non viene modellata in questo problema). Questo processo è stato modellato come un problema di soddisfacimento di vincoli che abbiamo definito locale, poiché ripetuto individualmente per ciascun Pokémon, con l'obiettivo di produrre fino a NUM\_SET\_MOSSE quadruple valide. In taluni casi il solver potrebbe trovare meno set di NUM\_SET\_MOSSE in tal caso considerando NUM\_SET\_MOSSE come un upper bound alla profondità degli alberi che vedremo in seguito

Set di mosse generati dal CSP locale (colonne separate per mossa)

Pokémon	Set	Mossa 1	Mossa 2	Mossa 3	Mossa 4	Totale Set
latios	Set 1	confusion	draco-meteor	smog	low-sweep	4
	Set 2	heal-block	dragon-pulse	poison-tail	secret-sword	
	Set 3	prismatic-laser	dragon-breath	gastro-acid	rolling-kick	
	Set 4	mirror-coat	dragon-tail	venom-drench	focus-punch	
steelix	Set 1	steel-wing	earth-power	freeze-shock	rock-slide	4
	Set 2	heavy-slam	mud-shot	ice-hammer	rock-throw	
	Set 3	gyro-ball	mud-bomb	blizzard	smack-down	

**Obiettivo del CSP LOCALE:** Il CSP locale assegna quattro mosse a ciascun Pokémon, rispettando **vincoli rigidi** (solo vincoli rigidi niente flessibili in questa fase) che garantiscono la qualità e varietà del set. Le mosse vengono selezionate ovviamente a partire dalla Knowledge Graph che abbiamo arricchito in precedenza.

L'intero problema del CSP locale, essendo un problema di modesta complessità computazionale, è implementato utilizzando la libreria [python-constraint](#), che consente la definizione dichiarativa di un CSP con variabili, domini e vincoli. In particolare ogni quadrupla è composta da 4 slot, ciascuno con un dominio costruito in modo semantico sulla base di determinate mosse:

- Slot 0: mosse del tipo primario (obbligatorio).
- Slot 1: mosse del tipo secondario (se presente), altrimenti si ripete il tipo primario.
- Slot 2: mosse di un tipo compatibile con il primario o secondario.
- Slot 3: altro tipo compatibile diverso dal precedente; se non disponibile, si usa una fusione dei domini precedenti (senza duplicati).

I tipi compatibili sono definiti tramite la funzione `ottiene_tipi_compatibili()` basata su regole di sinergia tra tipi Pokémon.

## Vincoli RIGIDI del CSP

1. Tutte le mosse devono essere diverse tra loro (AllDifferentConstraint).
2. Almeno una mossa di categoria fisica e una speciale.
3. Danno totale (base power)  $\geq 200$ .
4. PP totali  $\geq 25$ .
5. Precisione totale  $\geq 250$ .

Queste soglie sono state definite internamente al codice e non derivano da conoscenza pregressa.

Tali vincoli rigidi sono imposti attraverso funzioni vincolo dedicate (`_vincolo_categoria`, `_vincolo_danni`, ecc.) e hanno lo scopo di evitare la generazione di set deboli, ridondanti o troppo rischiosi. Ogni funzione valuta aggregati specifici (es. somma dei PP o danni) rispetto a soglie definite internamente; se un assegnamento non soddisfa i vincoli viene semplicemente scartato; ci accontentiamo solo della prima soluzione.

Dettagli di esecuzione: Questo processo è eseguito iterativamente per ogni Pokémon nella squadra personale, mantenendo la coerenza dell'ordine determinato nella fase di match-up. I set generati vengono poi passati a una fase successiva in cui **un CSP globale selezionerà il miglior set per ogni Pokémon** (sezione successiva).

## Problema quattro => Assegnazione Globale dei Set di Mosse

Una volta che, per ogni Pokémon della squadra, sono state generate NUM\_SET\_MOSSE (o anche un numero inferiore non causa problemi) possibili quadruple di mosse candidate tramite un **CSP locale** (vedi sezione precedente), il passo successivo è quello di selezionare per ciascun Pokémon **un solo set di mosse**, in modo da massimizzare la **funzione di qualità** complessiva dell'intera squadra.

Questo problema viene modellato come un **CSP di ottimizzazione** in cui:

- Ogni variabile rappresenta un Pokémon e il suo dominio è l'insieme dei NUM\_SET\_MOSSE **indici corrispondenti alle quadruple generate localmente; quindi lavoro su degli indici numerici che corrispondono a delle liste (vedesi codice per maggior chiarezza)**
- Non ci sono vincoli rigidi: ogni possibile assegnamento completo è lecito (problema di ottimizzazione puro senza vincoli rigidi)
- L'obiettivo è **massimizzare una funzione di qualità basata** su vincoli **flessibili**, ciascuno ponderato con un peso numerico specificato (es. danno medio, copertura di tipi, bilanciamento tra categorie fisiche e speciali, ecc.).

Nota Bene: Sostanzialmente si tratta di assegnare un numero fisso di indici numerici che vanno da 0 a NUM\_SET\_MOSSE – 1. Questo è il modo migliore a cui abbiamo pensato per risolvere il problema astraendo sulla complessità di lavora con set diversi per ciascun pokémon. Infatti, CIASCUN POKEMON ha un certo numero associato di set ed è nostro compito trovare l'assegnazione totale che massimizza una funzione definita internamente.

### ► Approccio 1: Ricerca sistematica con Or Tools e DFS

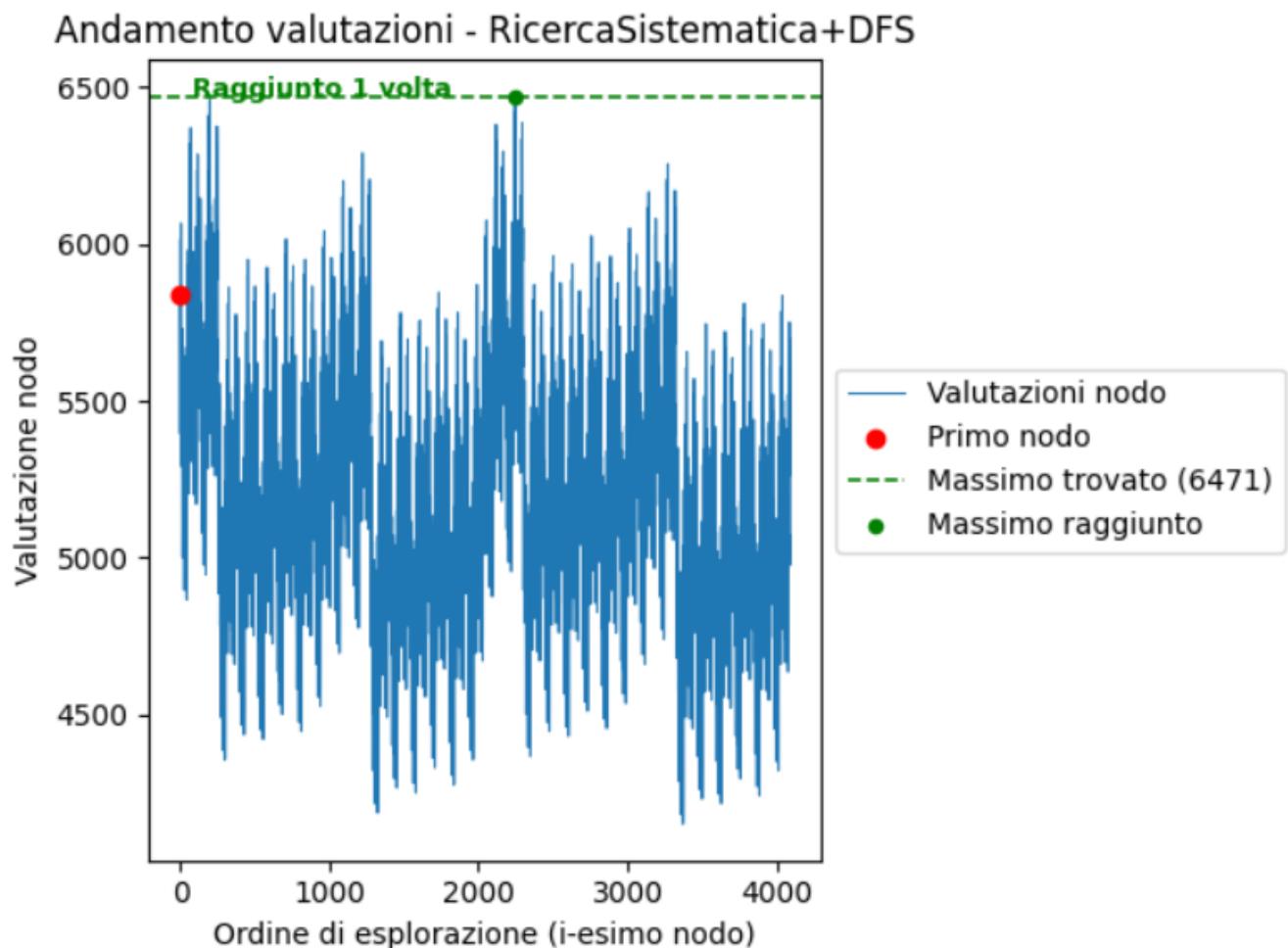
Nel file assegnazione\_mosse\_globale.py, l'algoritmo esplora **tutte le possibili assegnazioni totali** dei set di mosse (ad esempio, per 6 Pokémon con 10 set ciascuno:  $10^6$  combinazioni).

Sono stati sperimentati due metodi di ricerca sistematica:

- **dfs\_globale**: una **ricerca in profondità (DFS)** che esplora lo spazio delle assegnazioni parziali, accumulando passo dopo passo i punteggi fino a trovare la combinazione con valore massimo della funzione di qualità.
- **risvoli\_con\_ortools**: un approccio alternativo che utilizza **Google OR-Tools**, una libreria di livello industriale per la risoluzione efficiente di CSP e problemi combinatori. L'approccio è leggermente differente con OR-tools rispetto a tutti gli altri metodi ma comunque estremamente funzionante

L'approccio sistematico garantisce il **ritrovamento del massimo globale (e anche capire quante soluzioni esistono con tale valore di massimo)**, ma il costo computazionale può crescere esponenzialmente con l'aumentare dei Pokémon o dei set candidati. Difatti, il DFS deve scendere fino a profondità 6 (cioè deve arrivare ad una assegnazione totale per poterla valutare e determinare se rappresenta o meno un miglioramento rispetto a quanto trovato. Sappiamo che la DFS ha una ottima complessità spaziale ma ciò non significa che la sua complessità temporale sia buona perché questa è  $B^D$  ovvero esponenziale rispetto alla profondità dell'albero.

Mentre il DFS su CSP lavora con assegnazioni parziali che vengono via via incrementate fino ad arrivare ad assegnazioni totali, l'approccio che vediamo adesso lavora esclusivamente nel mondo delle assegnazioni totali.



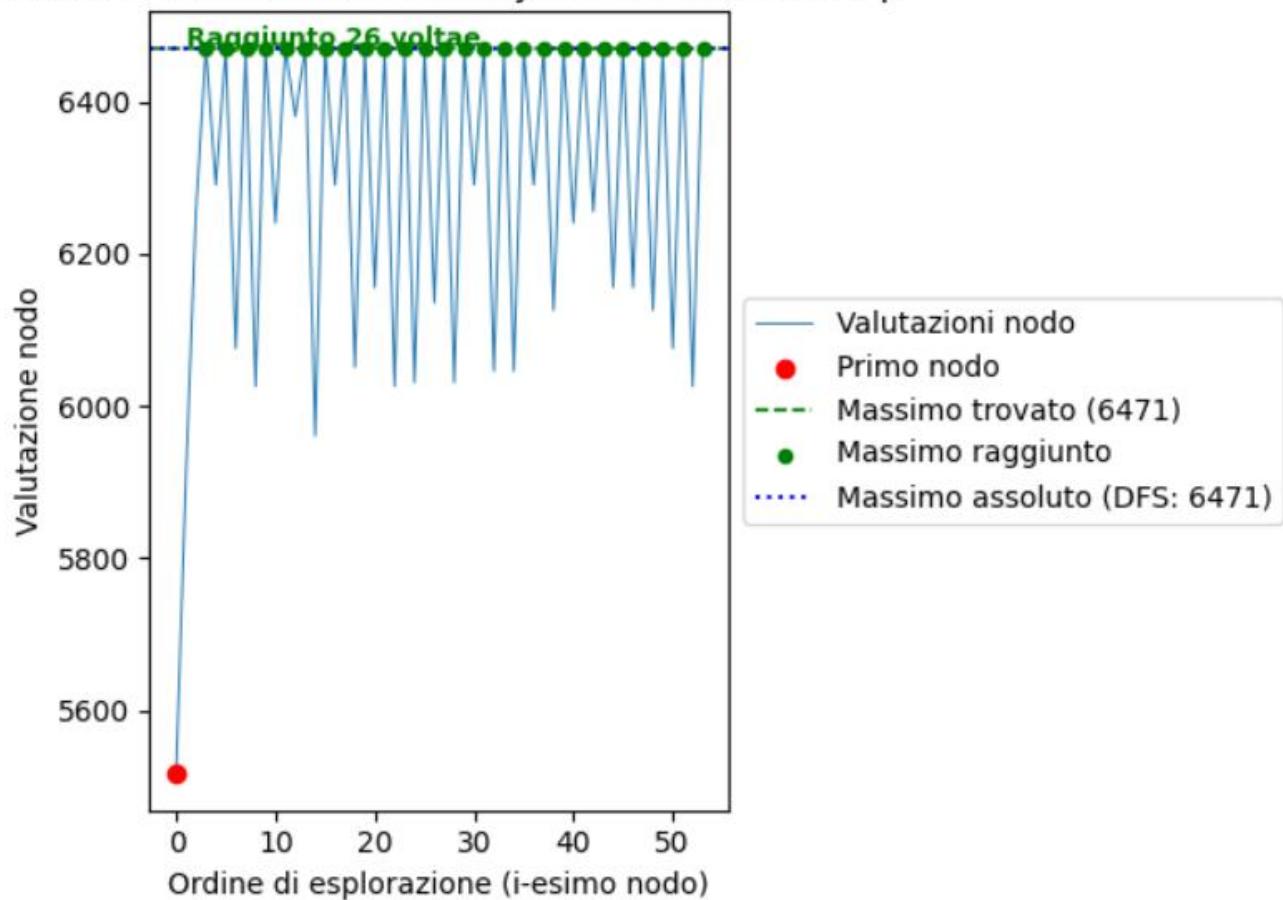
#### ► Approccio 2: Ricerca locale stocastica

Nel file `nodo_ricerca_locale.py`, è stato implementato un secondo approccio in cui si esplora lo spazio delle **assegnazioni totali** tramite metodi di **ricerca locale** (ad esempio Greedy + Random, Simulated Annealing, Beam Search, ecc.).

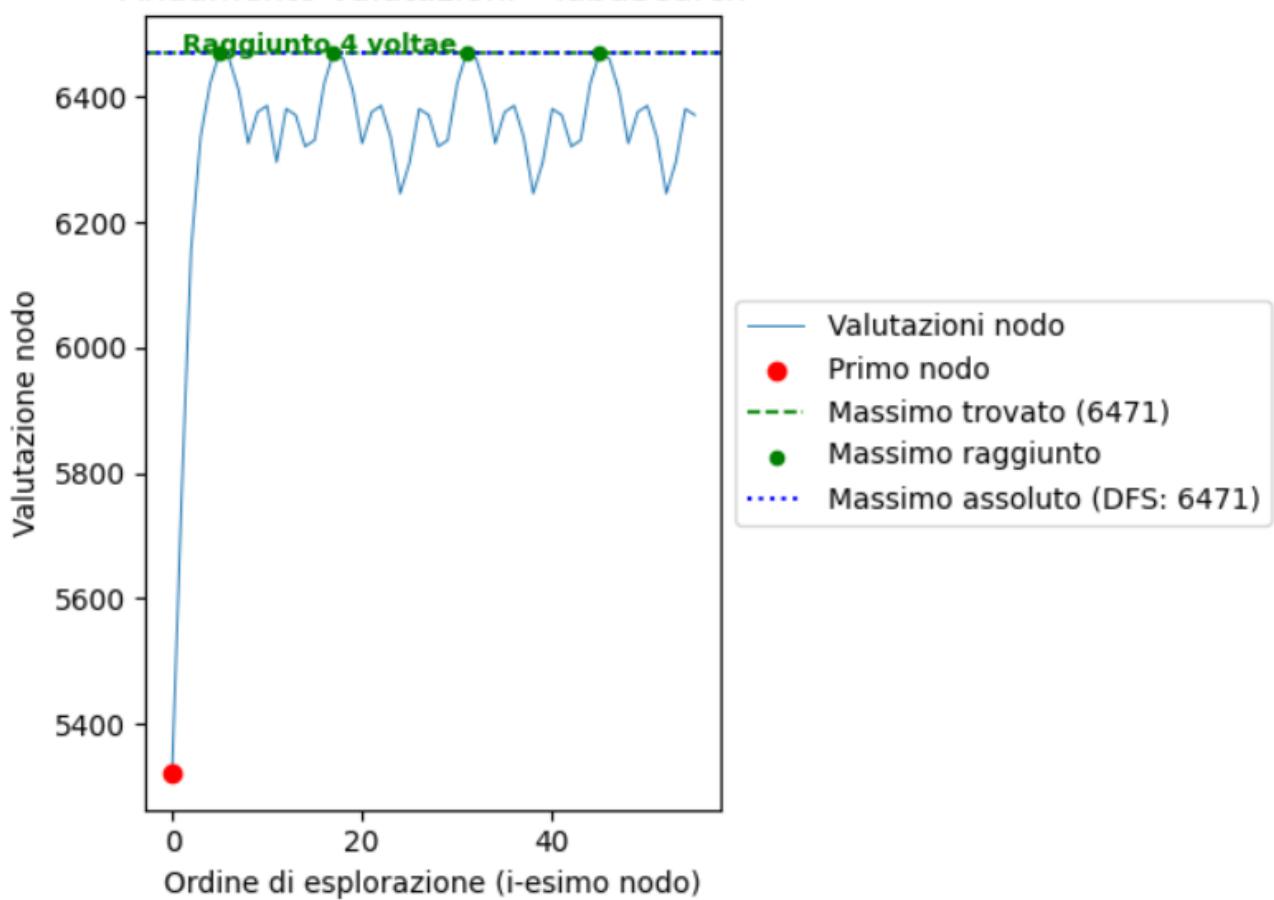
Questi metodi non costruiscono la soluzione incrementale, ma partono da una configurazione iniziale completa e la modificano in modo iterativo (il modo può cambiare da algoritmo ad algoritmo). Questo rende il processo molto più veloce, ma non garantisce la scoperta del massimo globale.

Plottiamo di seguito i valori dei nodi trovati e delle funzioni di valutazioni applicate a tali nodi:

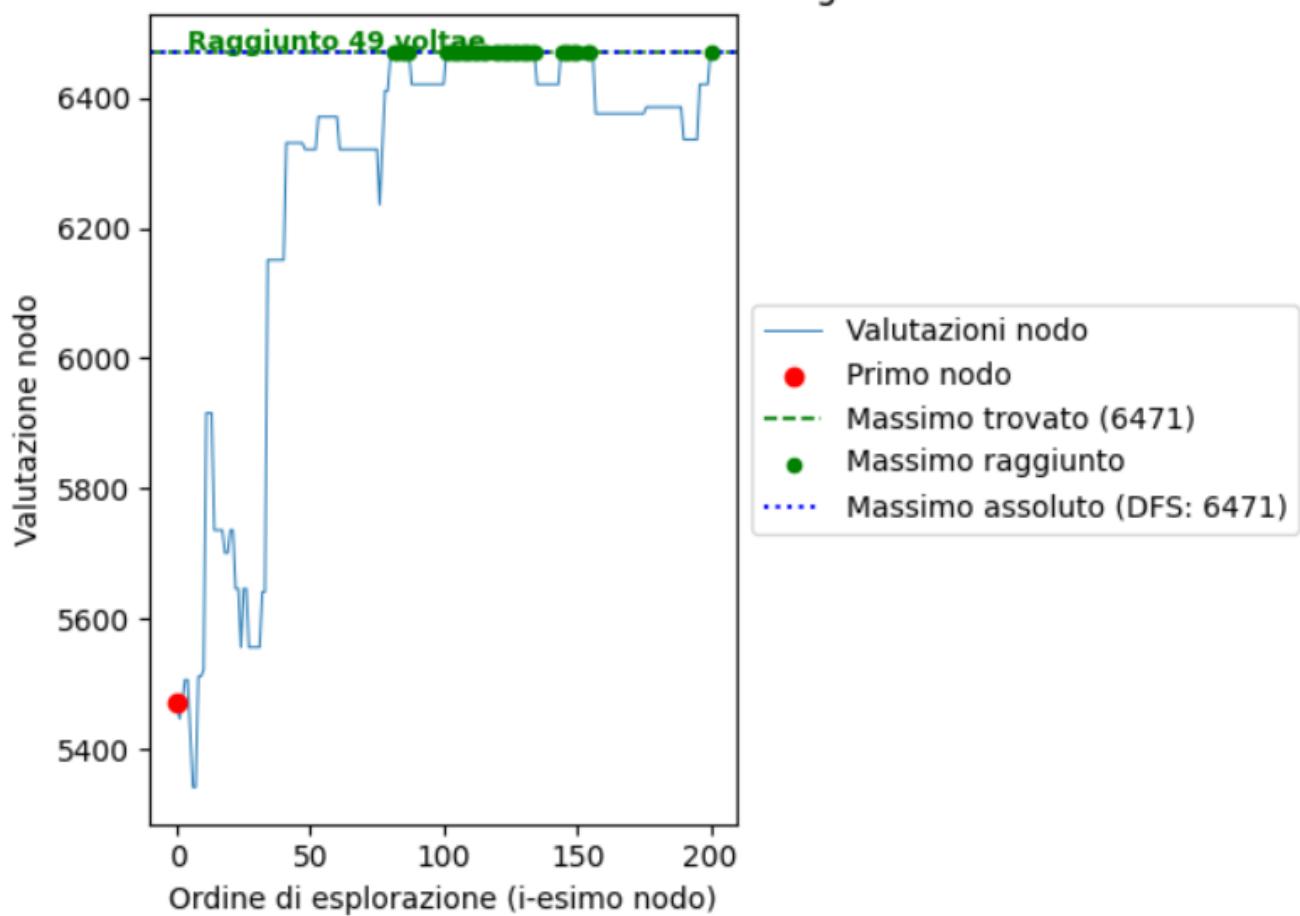
Andamento valutazioni - GreedyAscent+RandomStep



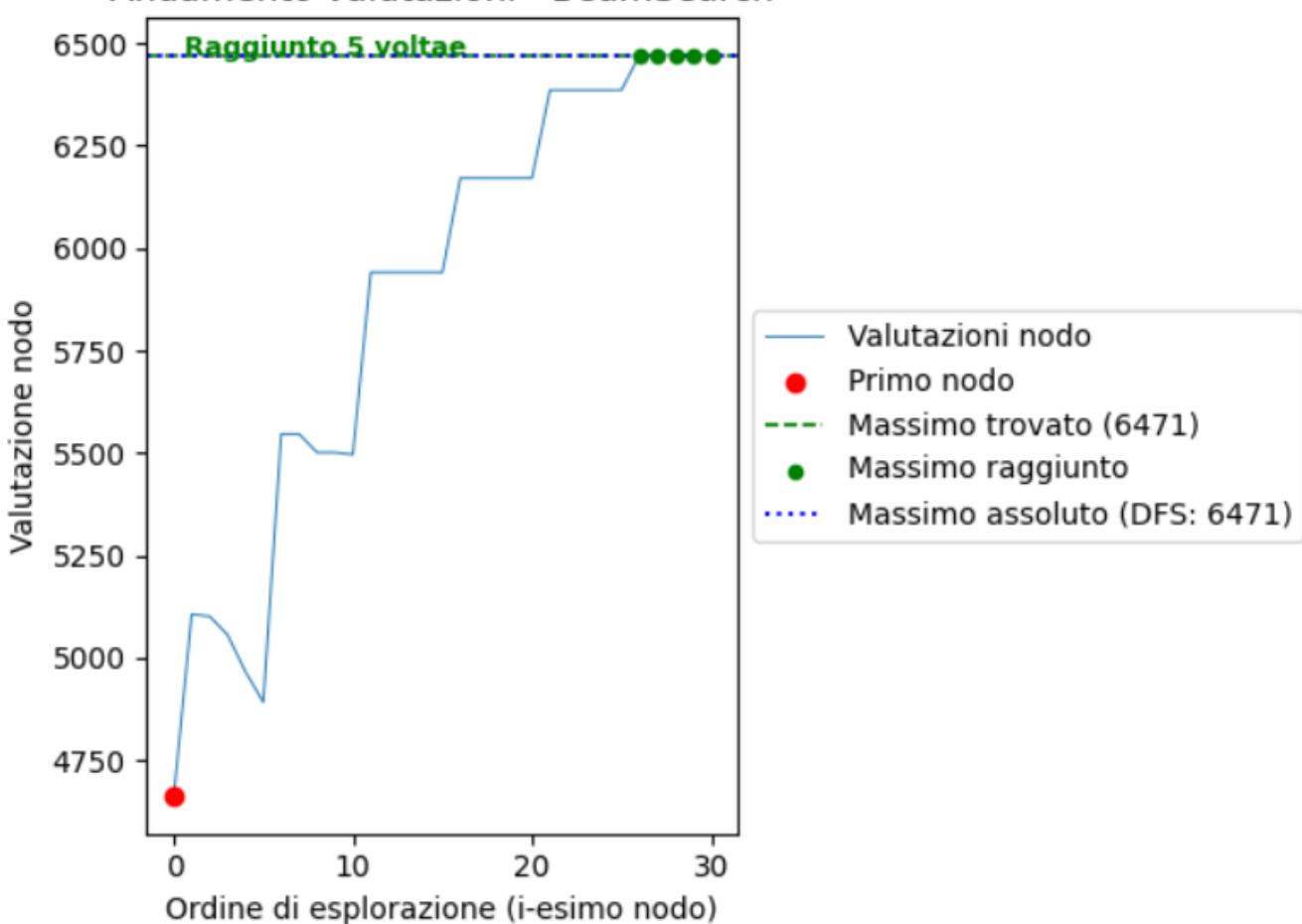
Andamento valutazioni - TabuSearch

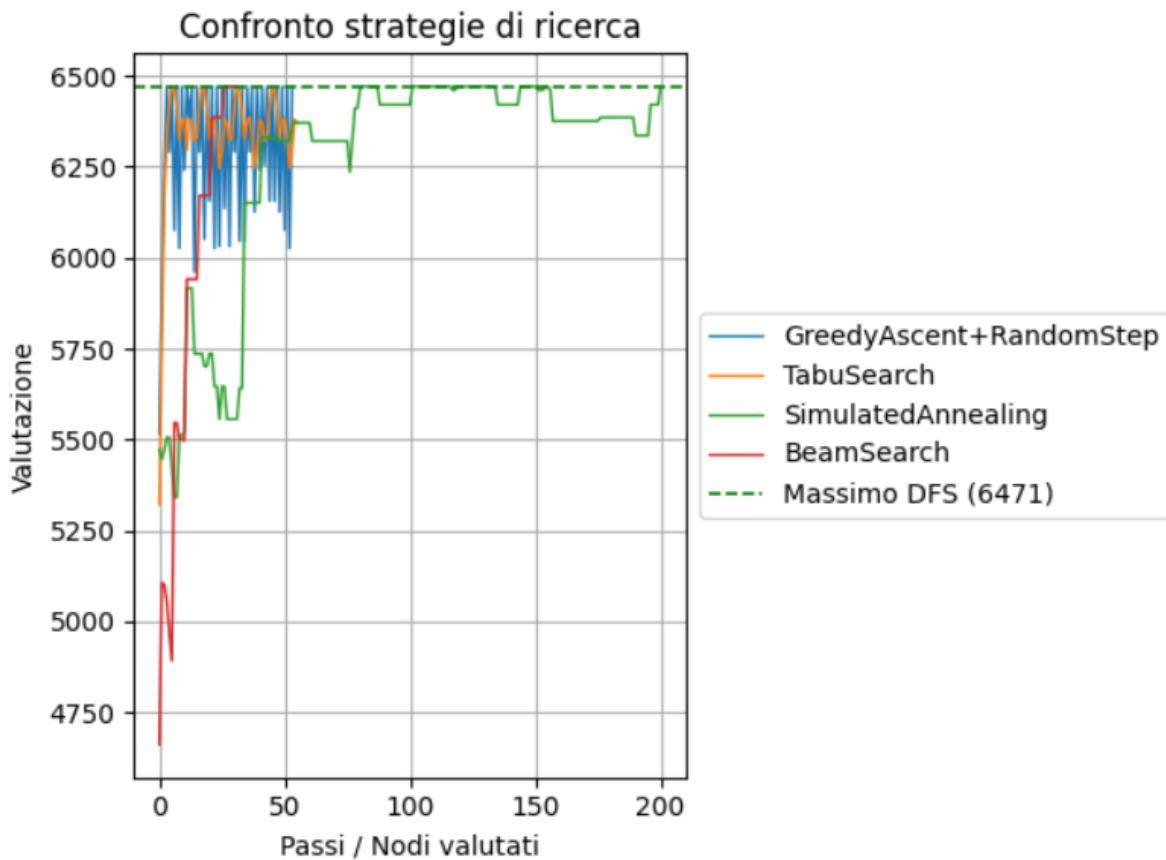


### Andamento valutazioni - SimulatedAnnealing



### Andamento valutazioni - BeamSearch





Sorprendentemente, nel 99% delle run da noi eseguite la **ricerca locale ha prodotto soluzioni equivalenti a quelle trovate via DFS**, mostrando un buon equilibrio tra qualità e tempi di calcolo. Un confronto rispetto ai tempi di calcolo è visto nella figura: si noti che hanno raggiunto lo stesso valore del massimo e i metodi di ricerca sono stati in grado di trovare il massimo più volte.

Una osservazione importante è che grazie alla ricerca sistematica siamo in grado di capire quante “soluzioni” con funzione di valutazione massima esistono nel nostro sistema (questo cambia da run a run ovviamente) mentre, con la ricerca locale per via della loro proprietà di non esplorare l’intero spazio delle possibili soluzioni in modo sistematico, non hanno garanzie di ritrovamento della soluzione con valore massimo anche se questa esiste.

I vari metodi di ricerca locale sono stati da me migliorati inserendo loro dei contatori speciali per evitare che rimanessero intrappolato negli STESSI massimi locali che in realtà rappresentavano un massimo globale (ma ciò l’algoritmo di ricerca locale non sa e non può sapere)

#### Confronto valori finali delle strategie:

- GreedyAscent+RandomStep: valore massimo trovato = 6471, tempo = 0.0177s
- TabuSearch: valore massimo trovato = 6471, tempo = 0.0147s
- RicercaSistematica+DFS: valore massimo trovato = 6471, tempo = 0.0676s
- OttimizzazioneGlobale: valore massimo trovato = 6471, tempo = 0.1425s
- SimulatedAnnealing: valore massimo trovato = 6471, tempo = 0.0092s
- BeamSearch: valore massimo trovato = 6471, tempo = 0.0107s

Questo passaggio è cruciale, in quanto determina **QUALE** l'assegnazione finale dei set di mosse che verranno poi utilizzati nello scontro 1-vs-1 simulato sullo spazio di stati.

#### Assegnazione delle mosse dal solver globale

Pokémon	Mossa 1	Mossa 2	Mossa 3	Mossa 4
latios	prismatic-laser	dragon-breath	gastro-acid	rolling-kick
steelix	steel-wing	earth-power	freeze-shock	rock-slide
turtonator	dragon-tail	blue-flare	inferno	water-spout
thundurus	dragon-ascent	electroweb	leaf-blade	secret-sword
silicobra	mud-shot	thousand-arrows	blast-burn	smack-down
morgrem	sucker-punch	light-of-ruin	shadow-bone	confusion

Anche in questo caso abbiamo utilizzato l'assegnamento di mosse prodotto da OR – Tools che in realtà è (ovviamente) equivalente a quello che generebbe la DFS.

## Simulazione dello scontro 1 vs 1 e ricerca in spazi di stati

La fase conclusiva dell'intero processo consiste nella simulazione e pianificazione **ottimale** di uno scontro 1 contro 1, utilizzando le mosse generate e assegnate nelle fasi precedenti. Lo scontro viene modellato come un **problema di ricerca su spazio di stati**, con l'obiettivo di **mandare KO l'avversario nel minor numero di turni**.

L'elemento centrale della modellazione è la classe StatoCombattimento, che implementa l'interfaccia NodoSpazioStati e rappresenta un singolo **istante della battaglia**. Questa classe è compatibile con le strategie di **ricerca sistematica** definite nel modulo spazio\_stati.py e consente di integrare diverse euristiche (qualora in futuro ci fosse la necessità) vincoli dinamici e logiche di pruning.

Nel codice si sono implementati BFS, DFS, IDDFS (iterative deepening DFS). Quale risulta essere il migliore? Non esiste un algoritmo migliore ma esiste solo l'algoritmo che meglio si adatta al problema. **Vogliamo ottenere la soluzione con il minor numero di mosse e poiché il costo dell'arco è unitario, la profondità della soluzione rappresenta il numero di mosse applicate.** Quindi una visita in ampiezza risulta essere la più indicata in questo scopo piuttosto che una visita in profondità. Visto comunque, che potrebbero essere generati molti stati contemporaneamente si è deciso di utilizzare la IDDFS che combina l'ottimalità della DFS (ricorsione) con il limite di profondità che difatti fa comportare la IDDFS come se fosse una BFS

### Modellazione dello Stato

Lo stato di un combattimento è caratterizzato da:

- hp\_avversario: HP rimanenti dell'avversario (intero, es. 600 iniziali).
- pp: lista dei PP rimanenti per ciascuna delle quattro mosse disponibili.
- turni\_rimanenti\_per Riposo: contatore che regola la frequenza con cui l'avversario si cura completamente.
- cure\_rimaste: numero massimo di cure totali disponibili per l'avversario.
- moltiplicatori: lista di float che rappresentano il moltiplicatore applicato a ciascuna mossa (aumenta in modo variabile ad ogni cura eseguita dall'avversario).
- set\_mosse: oggetto di tipo SetMosse, contenente le quattro mosse assegnate al Pokémon.
- ultime\_mosse\_usate: una coda FIFO di due elementi che impedisce l'uso consecutivo di una stessa mossa più di due volte. Significa che un pokémon è abilitato all'uso di una STESSA mossa al massimo per due turni consecutivi; al terzo turno il pokémon deve cambiare mossa

### Regole dinamiche e transizioni

Le regole che governano l'evoluzione degli stati sono le seguenti:

- Ogni transizione corrisponde all'uso di una mossa (con PP > 0 e basePower > 0).
- Il danno inflitto è **deterministico**, calcolato come basePower \* moltiplicatore corrente.
- Le mosse di tipo status (basePower = 0) vengono **prunate** in fase di generazione dei successori perché creerebbero solo loop.

- Ogni  $X$  turni (es. `TURNI_PER_RIPOSO = 4`), se l'avversario ha ancora cure disponibili:
  - Gli HP vengono ripristinati al massimo.
  - I moltiplicatori di danno per ogni mossa vengono incrementati in modo variabile secondo la formula:
  - `moltiplicatore_base = 0.5`  

$$\text{boost_aggiuntivo} = (\text{CURE_TOTALI} - \text{nuove_cure_rimaste}) * 0.25$$
  - Il contatore dei turni di cura viene resettato.
  - Il numero di cure rimaste viene decrementato.
- Non è consentito utilizzare la stessa mossa **più di due volte consecutivamente** (spam control).

Il combattimento termina con il raggiungimento dello **stato goal**, definito da `hp_avversario <= 0`.

### Strategia di Ricerca

Per trovare il **piano ottimale** (cioè la sequenza di mosse che porta al KO nel minor numero di turni), si utilizzano strategie di **ricerca sistematica**. In particolare, il modulo `spazio_stati.py` fornisce le seguenti implementazioni:

- `ricerca_dfs(stato_iniziale)`: visita in profondità senza limite, utile per trovare *una* soluzione.
- `ricerca_bfs(stato_iniziale, n_solutions)`: visita in ampiezza che esplora tutte le soluzioni più brevi.
- `ricerca_iddfs(stato_iniziale, profondita_massima)`: visita con profondità crescente, ideale per garantire il percorso di cammino minimo alias quello alla profondità più vicina alla radice

L'invocazione tipica per eseguire il calcolo è la seguente:

```
stato_iniziale = StatoCombattimento(
```

```
    hp_avversario=600,
    pp=[5, 5, 5, 5],
    turni_rimanenti_per_riposo=4,
    cure_totali=3,
    moltiplicatori=[1.0, 1.0, 1.0, 1.0],
    set_mosse=set_mosse
)
```

```
piani = RicercaSpazioStati.ricerca_iddfs(stato_iniziale, profondita_massima=50)
```

== RICERCA 1-VS-1: OGNI POKÉMON CON IL RISPECTIVO AVVERSARIO ==

--- ✅ Scontro 1: <https://pokemonkg.org/instance/pokemon/latios> vs <https://pokemonkg.org/instance/pokemon/lanturn> ---

Mosse a disposizione con PP originali:

```
[Mossa(move='https://pokemonkg.org/instance/move/prismatic-laser', basePower=160, accuracy=100, pp=10, tipo_mossa='https://pokemonkg.org/ontology#PokéType:Psychic', categoria_mossa='https://pokemonkg.org/ontology#SpecialMove'),
Mossa(move='https://pokemonkg.org/instance/move/dragon-breath', basePower=60, accuracy=100, pp=20, tipo_mossa='https://pokemonkg.org/ontology#PokéType:Dragon', categoria_mossa='https://pokemonkg.org/ontology#SpecialMove'),
Mossa(move='https://pokemonkg.org/instance/move/gastro-acid', basePower=0, accuracy=100, pp=10, tipo_mossa='https://pokemonkg.org/ontology#PokéType:Poison', categoria_mossa='https://pokemonkg.org/ontology#StatusMove'),
Mossa(move='https://pokemonkg.org/instance/move/rolling-kick', basePower=60, accuracy=85, pp=15, tipo_mossa='https://pokemonkg.org/ontology#PokéType:Fighting', categoria_mossa='https://pokemonkg.org/ontology#PhysicalMove')]
```

- Tipi avversario: ['<https://pokemonkg.org/ontology#PokéType:Water>', '<https://pokemonkg.org/ontology#PokéType:Electric>']
- Moltiplicatori calcolati tenendo conto tipi avversario: [1.0, 1.0, 1.0, 1.0]
- PP di partenza [3, 10, 10, 10]

✅ IDDFS ha trovato 1 percorso/i.

▣ Percorso soluzione 1 (KO in 7 mosse):

```
Turno 0: StatoCombattimento(HP=600, PP=[3, 10, 10, 10], Cure=3, TurniRiposo=4, Mult=[1.0, 1.0, 1.0, 1.0])
Turno 1: StatoCombattimento(HP=440, PP=[2, 10, 10, 10], Cure=3, TurniRiposo=3, Mult=[1.0, 1.0, 1.0, 1.0])
Turno 2: StatoCombattimento(HP=380, PP=[2, 9, 10, 10], Cure=3, TurniRiposo=2, Mult=[1.0, 1.0, 1.0, 1.0])
Turno 3: StatoCombattimento(HP=320, PP=[2, 8, 10, 10], Cure=3, TurniRiposo=1, Mult=[1.0, 1.0, 1.0, 1.0])
Turno 4: StatoCombattimento(HP=600, PP=[2, 8, 10, 9], Cure=2, TurniRiposo=4, Mult=[1.75, 1.75, 1.75, 1.75])
Turno 5: StatoCombattimento(HP=320, PP=[1, 8, 10, 9], Cure=2, TurniRiposo=3, Mult=[1.75, 1.75, 1.75, 1.75])
Turno 6: StatoCombattimento(HP=40, PP=[0, 8, 10, 9], Cure=2, TurniRiposo=2, Mult=[1.75, 1.75, 1.75, 1.75])
Turno 7: StatoCombattimento(HP=-65, PP=[0, 7, 10, 9], Cure=2, TurniRiposo=1, Mult=[1.75, 1.75, 1.75, 1.75])
```

Questo scontro si immagini replicato per ogni pokémon. Alla fine, una tabella riassuntiva descrive il numero di passi minimo per sconfiggere il pokémon avversario

Algoritmo di ricerca utilizzato: IDDFS

Scontro #	Soluzione trovata	Lunghezza
1	SI	7
2	SI	7
3	SI	2
4	SI	10
5	SI	2
6	SI	3

## Apprendimento NON supervisionato

Nell'apprendimento non supervisionato non disponiamo di una *feature target* esplicita da predire. L'obiettivo principale diventa quindi quello di **identificare gruppi omogenei di esempi**, mettendo in evidenza le similitudini tra le osservazioni presenti nel dataset.

In questo contesto, l'algoritmo non cerca di classificare i dati in categorie già note, ma di **scoprire automaticamente delle strutture nascoste**, individuando pattern che permettono di distinguere insiemi di Pokémon con caratteristiche comuni. Il dataset di partenza che abbiamo modificato è il dataset (<https://www.kaggle.com/datasets/abcsds/pokemon>) che contiene 800 pokémon di cui alcuni andrebbero eliminati per estensioni future (pokémon mega evoluti, forme alternative di leggendari etc.).

L'obiettivo iniziale del clustering era quello di suddividere i Pokémon in gruppi omogenei sulla base delle loro **statistiche individuali (HP, Attack, Speed, Sp. Atk, Sp. Def)**, con l'idea di ottenere raggruppamenti che riflettessero i diversi livelli di forza. Prima di applicare l'algoritmo, era lecito aspettarsi che il modello producesse una distinzione tra Pokémon “più forti” e “meno forti” e che, aumentando il numero di cluster, emergessero sottocategorie simili a quelle esistenti nel contesto competitivo.

Tuttavia, i risultati preliminari (non riportati nella documentazione ma visualizzati da me) hanno mostrato che l'applicazione diretta del K-Means a queste feature grezze generava cluster poco interpretabili (a livello semantico da parte del supervisore).

Questo è un chiaro sintomo della **bassa qualità informativa delle feature scelte**, poiché basarsi unicamente su valori numerici elementari non è sufficiente a distinguere in modo significativo le diverse tipologie di Pokémon neanche nel mondo reale.

Per superare questo limite, si è proceduto a una fase di **ingegnerizzazione delle feature**, ovvero alla costruzione di nuove variabili più **semanticamente rappresentative** dei Pokémon. La sostituzione delle semplici statistiche con queste feature derivate ha rappresentato un vero e proprio salto di qualità, consentendo al clustering di catturare meglio le differenze tra archetipi e rendendo i raggruppamenti molto più interpretabili. Di seguito si riportano le nuove feature sulle quali andiamo a lavorare; tutte le altre feature vengono scartate in quanto inutili al problema.

```
df["OFF_FISICO"] = df["Attack"] + df["Speed"]
df["OFF_SPECIALE"] = df["Sp. Atk"] + df["Speed"]
df["TANK_FISICO"] = df["HP"] + df["Defense"]
df["TANK_SPECIALE"] = df["HP"] + df["Sp. Def"]
features = ["OFF_FISICO", "OFF_SPECIALE", "TANK_FISICO", "TANK_SPECIALE"]
```

OSS\_IMPORTANTISSIMA: prima ancora di applicare il k-means sul dataset, questo viene diviso in **training set e test set**.

La divisione serve a garantire che il modello non sia valutato sugli stessi dati usati per la costruzione dei cluster, simulando l'arrivo di Pokémon “nuovi”.

Viene poi applicata la standardizzazione (StandardScaler): tutte le feature hanno media 0 e deviazione standard 1, così da essere confrontabili e non falsare la distanza euclidea usata da K-Means. Infatti, l'algoritmo k-means utilizza le distanze euclidea che sono sensibili alla scala di valori utilizzati.

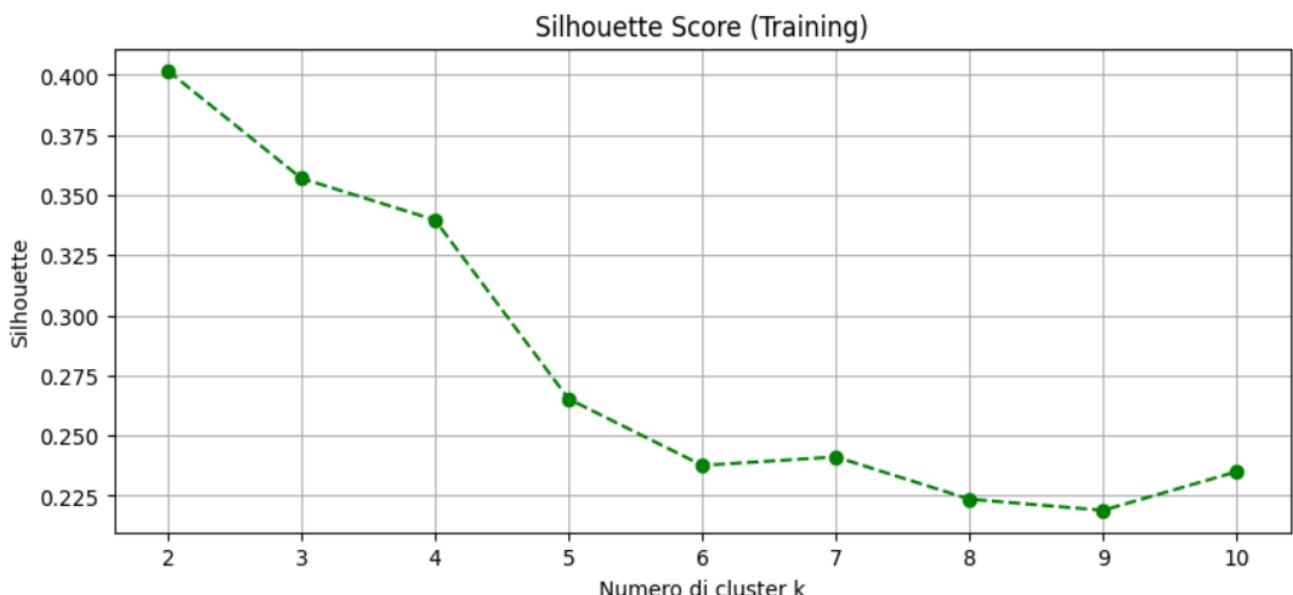
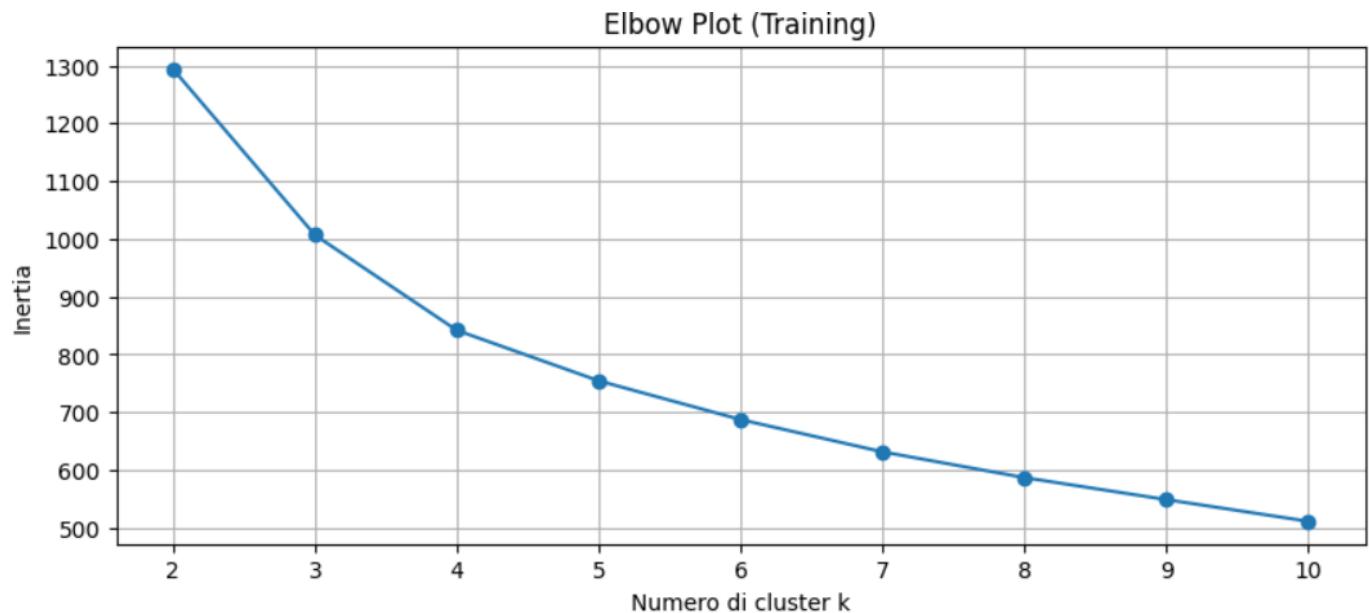
Come anticipato l'algoritmo non supervisionato è il k-means con parametri:

- Inizializzazione: k-means++ per convergenza stabile
- Parametri esplicativi: random\_state=42 per la riproducibilità
- n\_init=10 per la robustezza.
- Altri parametri chiave: è stato usato il valore di default max\_iter=300.

Un aspetto fondamentale nell'applicazione del K-Means è la scelta del iper-parametro k, cioè il numero di cluster in cui suddividere i dati. Per determinare K abbiamo utilizzato due metriche ampiamente studiate:

- Elbow Method (Inertia): misura la compattezza dei cluster. Man mano che aumentiamo k, l'inerzia diminuisce, ma oltre un certo punto la riduzione diventa marginale: il "gomito" della curva suggerisce un buon compromesso.
- Silhouette Score: valuta la separazione dei cluster, confrontando la distanza media tra punti dello stesso cluster e punti di cluster diversi. Un valore più alto indica cluster meglio definiti.

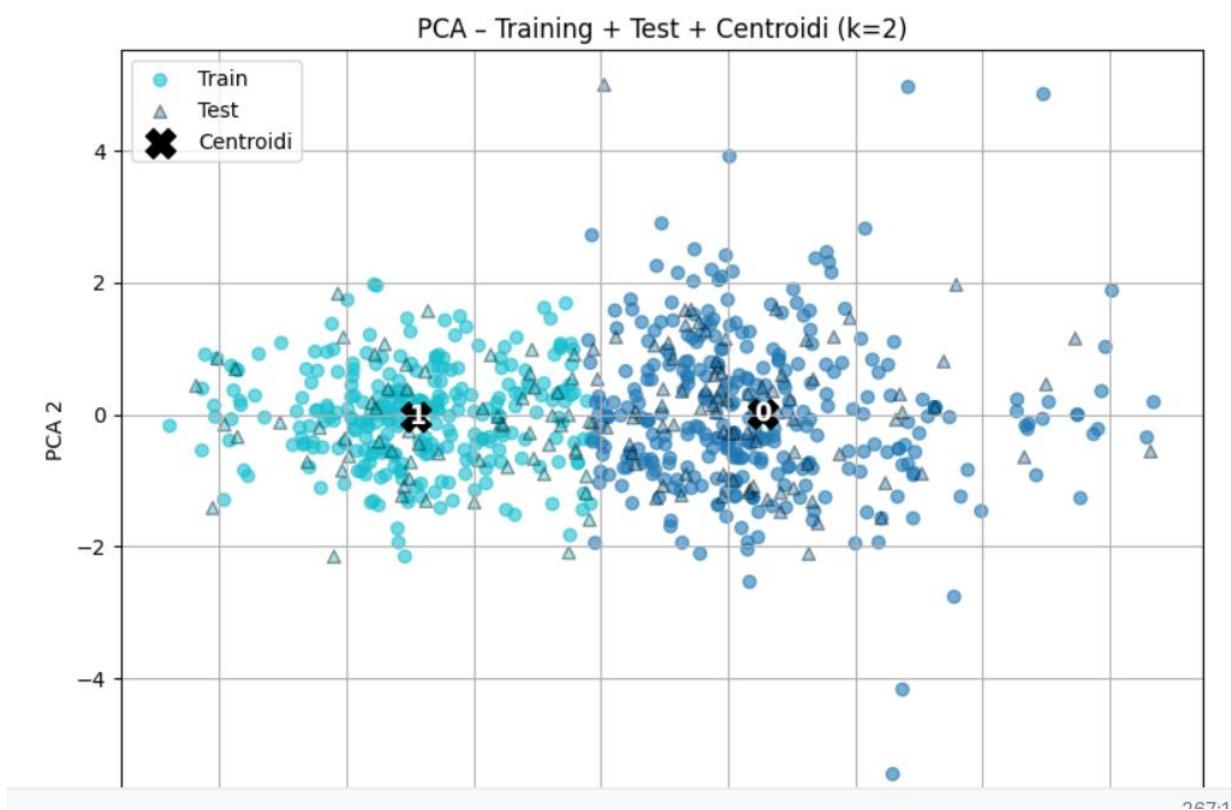
Per eseguire ciò abbiamo eseguito il K-means per vari valori di K ks = list(range(2, 11)) e plottate le metriche su un grafico:



Dal confronto dei due grafici emergono alcune osservazioni:

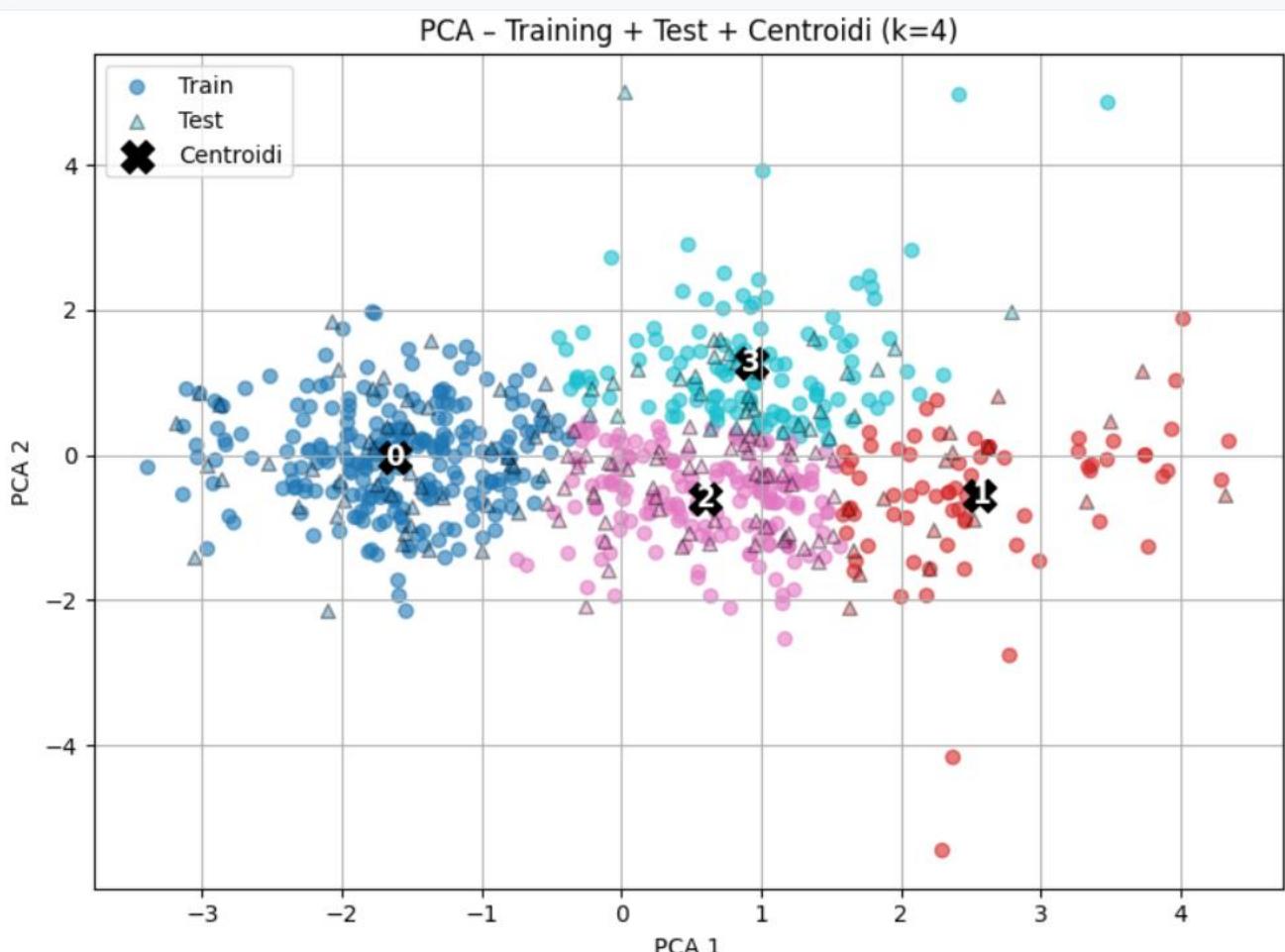
- Elbow Plot: la curva mostra una forte diminuzione dell'inerzia tra  $k=2$  e  $k=4$ , dopodiché la riduzione diventa più graduale. Questo suggerisce che il punto di "gomito" si colloca intorno a **k=4**, valore che rappresenta un buon compromesso tra compattezza dei cluster e semplicità del modello.
- Silhouette Score: il valore più alto si registra per  $k=2$  ( $\sim 0.40$ ), ma ciò significherebbe dividere tutti i Pokémon solo in due grandi gruppi, con una perdita di dettaglio. Per valori superiori, lo score cala progressivamente, ma si mantiene ancora accettabile intorno a  $k=3-4$ , garantendo una buona separazione dei cluster.

OSS: cosa sarebbe K = 2? Molto semplicemente con k = 2, stiamo suddividendo i pokémon in due gruppi: Pokemon forti e pokémon deboli. Poco utile predire se un pokémon è forte o debole quindi aumentiamo il numero di cluster.



Quando invece  $K = 4$  otteniamo questa suddivisione:

Alcuni gruppi (es. cluster 0 e 2) appaiono più compatti, altri (es. cluster 1) più dispersi. Sono presenti zone di sovrapposizione (soprattutto tra cluster 2 e 3), segno che certi Pokémon condividono caratteristiche ibride come d'altronde coerente nel dominio pokémon.



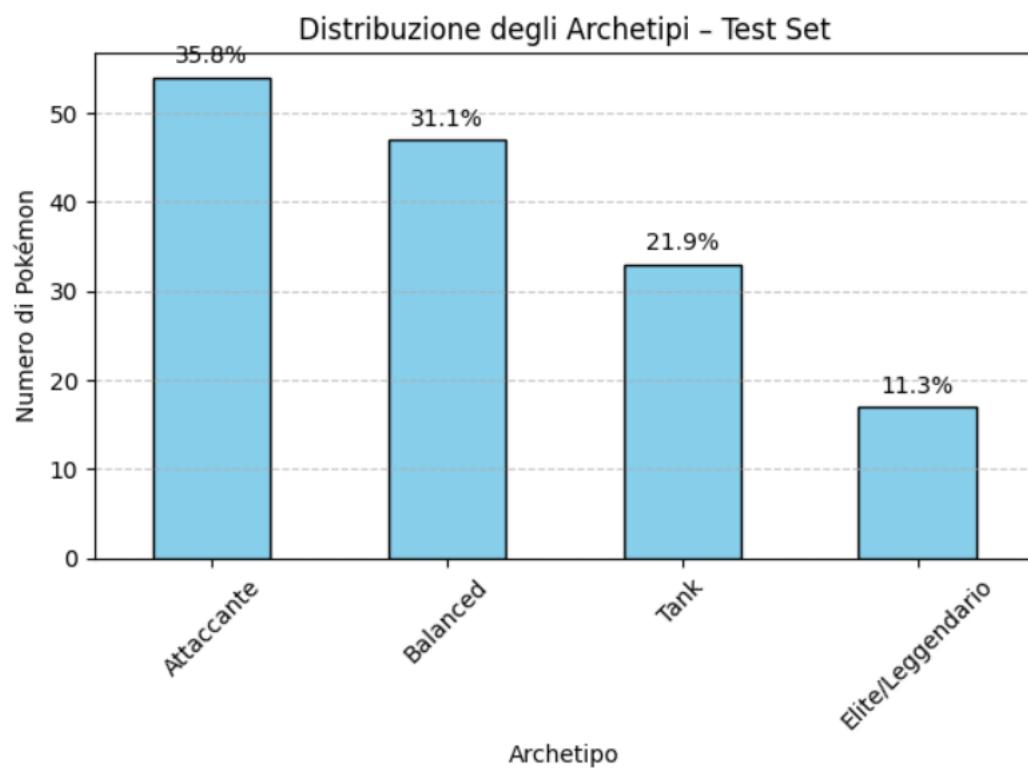
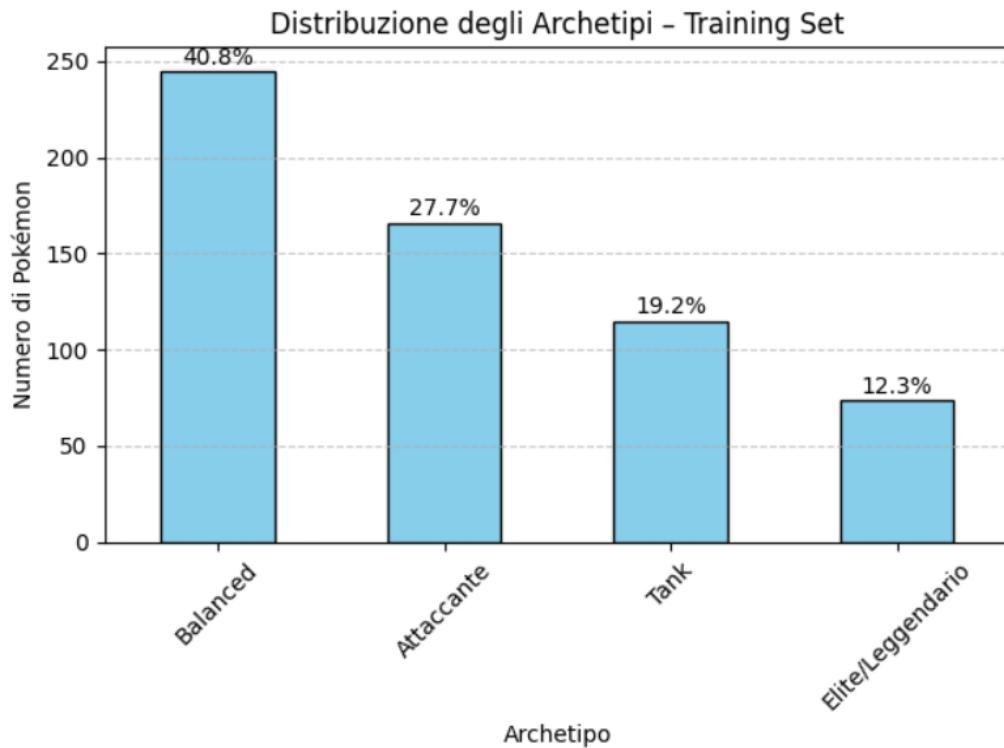
La vera semantica ai cluster deve essere attribuita dall'esperto (il supervisore umano) che analizzando i quattro centroidi (essendo delle medie) è in grado di determinare la semantica dei raggruppamenti. In particolare, abbiamo optato per questa suddivisione (screen preso dai commenti nel mio codice).

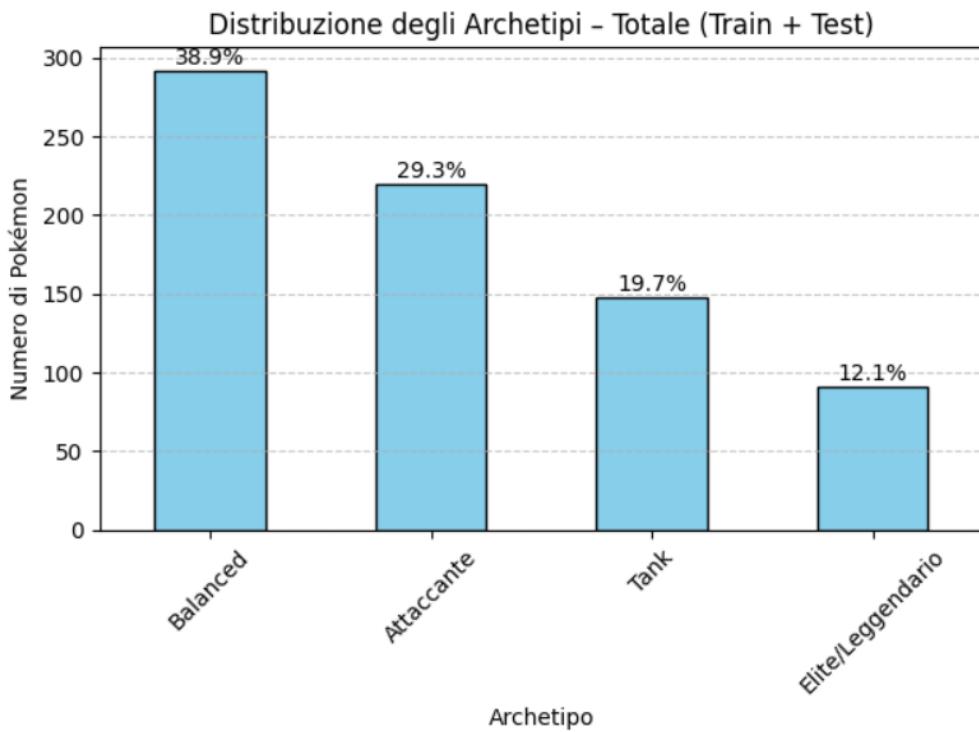
## Tabella Archetipi e SubArchetipi

Cluster	Archetipo	Possibili SubArchetipi
0	Bilanciati	—
1	Attaccante	Attaccante Fisico, Attaccante Speciale
2	Elite/Leggendario	—
3	Tank	Tank Difensivo, Tank Speciale

Una volta scelto k possiamo riapplicare k means sul training ottenendo il modello che viene utilizzato per clusterizzare gli esempi di test.

In questa fase è stato dunque possibile analizzare la distribuzione delle “classi” presenti nel training e test set su cui i successivi modelli supervisionati andranno addestrati.





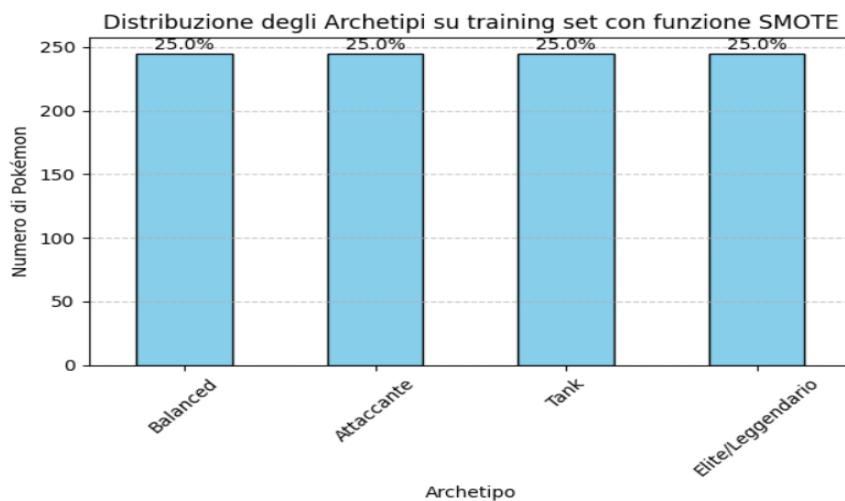
In origine, il training set presenta una forte asimmetria: archetipi come \*Bilanciato\* e \*Attaccante\* sono molto più numerosi rispetto a \*Tank\* ed \*Elite/Leggendario\*.

Questa situazione riflette fedelmente il mondo Pokémon, ma può influenzare negativamente i modelli di classificazione, che tenderebbero a favorire le classi più popolose. Per evitare che i modelli supervisionati trascurassero i cluster più piccoli, è stata applicata la tecnica **SMOTE** sul solo training set.

- k\_neighbors=5: valore standard che rappresenta un buon compromesso tra diversità e coerenza dei campioni sintetici.
- random\_state=42: fissato per garantire la riproducibilità dei risultati.

Questa scelta ha permesso di ridurre il bias verso le classi maggioritarie senza contaminare il test set. Il nuovo training set è mostrato in figura.

Lo smote deve essere applicato al train set? NO, perché il test set è un surrogato degli esempi del mondo reale e nel contesto del mondo pokémon sappiamo che in modo naturale le classi sono sempre state sbilanciate.



Una porzione di visualizzazione del nuovo dataset prodotto dove sono state eliminate tutte le feature irrilevanti ed è stata aggiunta la colonna archetipo (sub-archetipo non è necessario)

The screenshot shows a data visualization interface with a toolbar at the top containing icons for search, filter, sort, and edit, followed by "Edit in Data Wrangler" and "CSV". The main area displays a table with the following data:

	OFF_FISICO	OFF_SPECIALE	TANK_FISICO	TANK_SPECIALE	Archetipo
1	96	96	96	96	96 Balanced
2	130	160	160	160	165 Attaccante
3	175	170	145	150	Attaccante
4	105	122	80	100	Balanced
5	165	105	165	150	Tank
6	124	124	183	193	Tank
7	184	107	136	116	Attaccante

Osservazione: Si noti che in questa fase abbiamo costruito anche altri due dataset, molto diverso dai precedenti: [pokemon\\_train\\_con\\_distanze.csv](#) e [pokemon\\_test\\_con\\_distanze.csv](#). Con questi dataset vogliamo sfruttare le distanze dai centroidi K-Means come feature continue per addestrare un classificatore supervisionato (in questo caso, sarà la regressione logistica multiclasse), utile per prevedere il cluster di appartenenza di ciascun Pokémon sfruttando direttamente le distanze euclidee.

## Apprendimento Supervisionato

In questa sezione applichiamo gli algoritmi supervisionati per classificare i Pokémon in uno dei quattro archetipi individuati precedentemente tramite clustering:

- Tank
- Bilanciati
- Offensivi
- Elite / Leggendari

L'obiettivo sarà capire quanto bene i modelli supervisionati (di base e ensemble) riescono a distinguere i Pokémon nei 4 archetipi, utilizzando solo le quattro feature numeriche principali:

- `OFF\_FISICO`
- `OFF\_SPECIALE`
- `TANK\_FISICO`
- `TANK\_SPECIALE`

Tutti i modelli sono stati valutati mediante **10-fold cross validation**, così da osservare in media il comportamento del modello al variare del validation set.

Per ciascun valore dell'iper-parametro testato, la procedura è stata ripetuta su 10 diverse suddivisioni dei dati: ad ogni run sono stati calcolati i valori delle metriche, e successivamente è stata calcolata la media complessiva.

Al termine, oltre alla media finale, è stata considerata anche la **deviazione standard**, utile per misurare il grado di incertezza del modello e valutare la stabilità delle prestazioni rispetto a variazioni nei dati di validazione.

La metrica di errore presa come riferimento è la F1 – macro che bilancia precisione e recall ed è molto utile quando abbiamo degli esempi da predire sbilanciati (difatti il test set è sbilanciato)

In generale dobbiamo aspettarci che le prestazioni calino leggermente su il test, essendo sbilanciato, ma questo non è affatto una criticità poiché:

- il training bilanciato con smote serve a dare al modello la capacità di riconoscere tutte le classi, senza pregiudizi su classi più frequenti
- il test sbilanciato serve a verificare come il modello si comporta in uno scenario realistico ovvero in uno scenario in cui le classi sono sbilanciate come il dominio pokémon.

## K-NN

Innanzitutto, essendo gli esempi stati raggruppati sulla base di distanze euclidee si è sperimentato l'utilizzo dell'algoritmo k-Nearest-Neighbors, che utilizza esattamente la distanza euclidea per assegnare la classe più frequente nei  $k$  vicini la classe da assegnare. Per questo ci è sembrata la scelta più giustificabile vedendo un comportamento simile al k-means.

Di seguito si riportano le metriche calcolate con la ten fold cross validation per valori di  $k = \text{list}(\text{range}(1, 31, 2))$  #  $k$  dispari da 1 a 29

Top-10 per F1-macro in CV:									
k	acc_mean	acc_std	prec_mean	prec_std	rec_mean	rec_std	f1_mean	f1_std	
1	0.975510	0.016580	0.976629	0.015843	0.975375	0.016586	0.975393	0.016621	
3	0.972449	0.018843	0.973813	0.017991	0.972375	0.018709	0.972242	0.018840	
15	0.971429	0.009998	0.972443	0.009629	0.971500	0.010037	0.971430	0.009977	
5	0.971429	0.017556	0.972727	0.016461	0.971458	0.017611	0.971305	0.017595	
17	0.970408	0.009627	0.971304	0.009426	0.970458	0.009681	0.970430	0.009658	
13	0.970408	0.013265	0.971877	0.012282	0.970417	0.013405	0.970318	0.013311	
21	0.969388	0.014431	0.970271	0.014097	0.969458	0.014447	0.969338	0.014469	
23	0.969388	0.013690	0.970097	0.013220	0.969500	0.013704	0.969323	0.013714	
11	0.968367	0.013265	0.969751	0.012581	0.968333	0.013340	0.968262	0.013251	
9	0.968367	0.014028	0.969625	0.013390	0.968375	0.014132	0.968226	0.014185	
19	0.967347	0.014286	0.968469	0.014068	0.967375	0.014454	0.967360	0.014345	
27	0.967347	0.014286	0.967803	0.014252	0.967417	0.014373	0.967231	0.014438	
7	0.967347	0.018139	0.968452	0.017740	0.967417	0.018138	0.967193	0.018212	
29	0.965306	0.015272	0.965921	0.015105	0.965375	0.015283	0.965250	0.015310	
25	0.965306	0.012245	0.966045	0.012211	0.965333	0.012380	0.965154	0.012406	

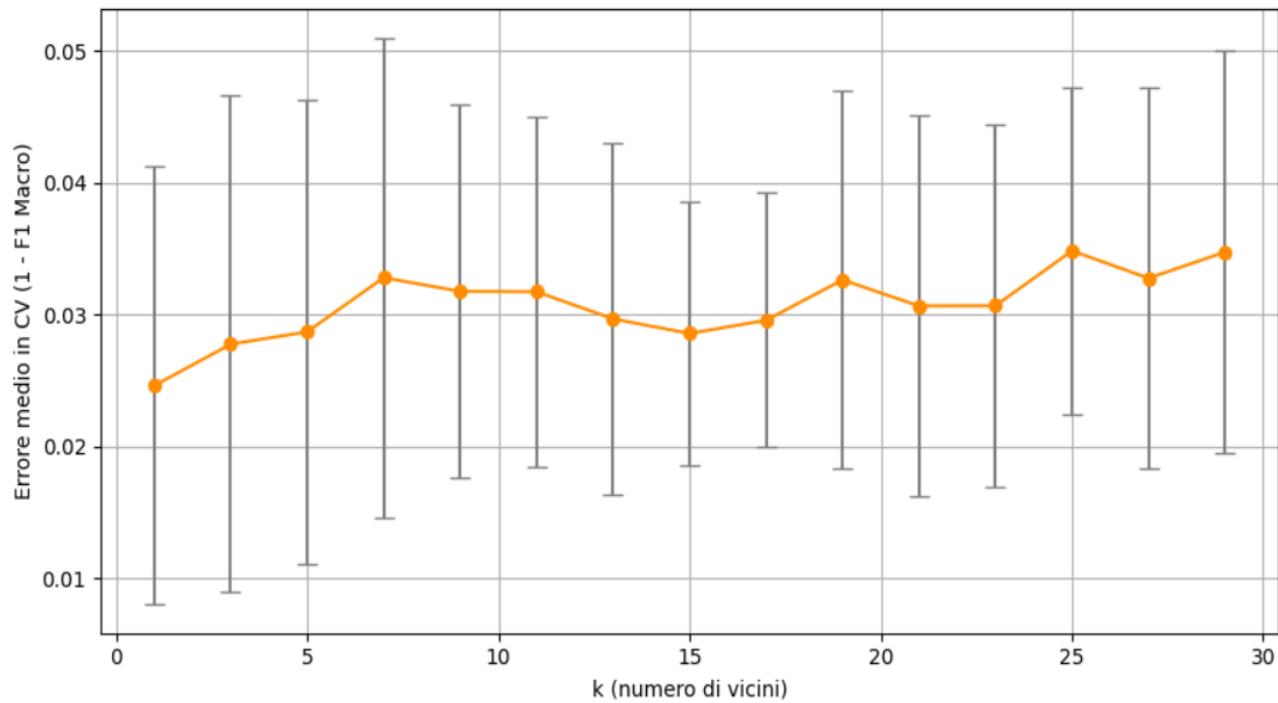
1 row	v	1 rows x 4 cols	Static	
♦ Accuracy      ♦ Precisione      ♦ Richiamo      ♦ F1-score      ♦				
Media ± std su tutti i k	$0.969 \pm 0.014$	$0.970 \pm 0.014$	$0.969 \pm 0.014$	$0.969 \pm 0.014$

Le metriche (Accuracy, Precisione, Richiamo e F1-score) sono tutte molto elevate e stabili, con valori medi intorno a 0.97 e una deviazione standard di circa  $\pm 0.014$ , segno di un modello robusto e poco sensibile alla suddivisione del dataset. Per valori di  $k$  intorno a 13–17 si ottengono i risultati migliori in termini di equilibrio tra performance e stabilità. Nel complesso, il modello KNN mostra un comportamento coerente e affidabile su tutto l'intervallo analizzato.

Questo conferma che la scelta di  $k=13$  è appropriata, poiché garantisce alte prestazioni mantenendo al tempo stesso una bassa variabilità.

La medesima situazione è visibile attraverso il grafico ottenuto plottando questi valori

KNN — 10-Fold CV: Errore medio  $\pm$  Deviazione Standard



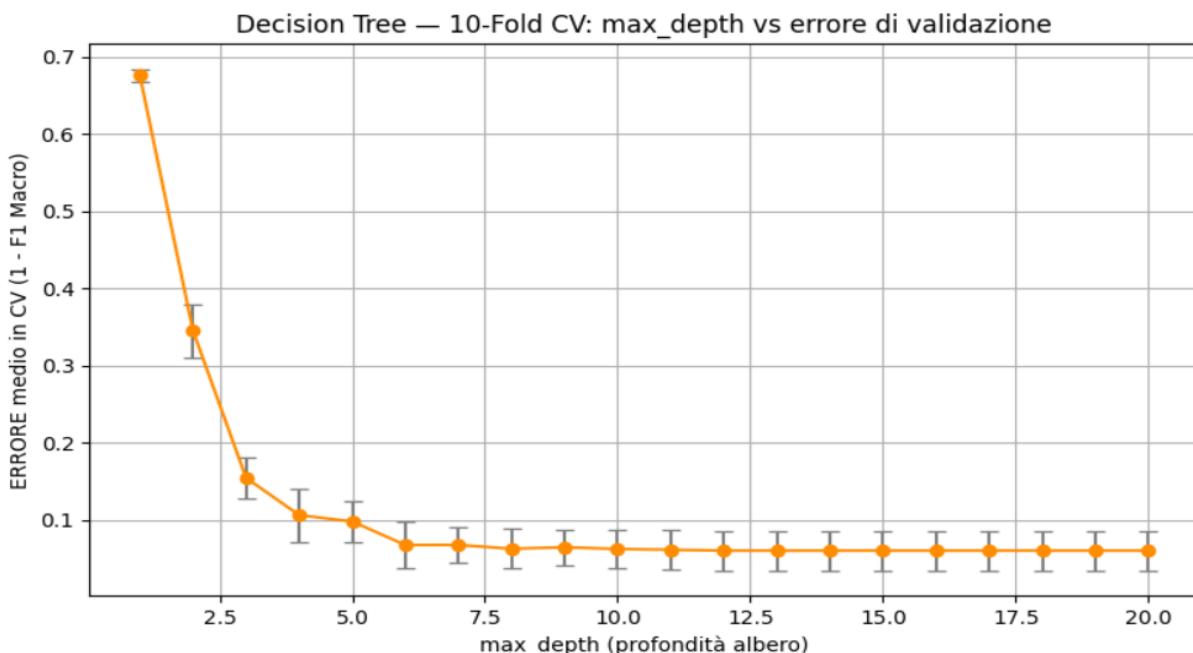
## Albero di decisione con Iperparametro max\_depth

Eseguiamo la medesima valutazione che abbiamo fatto in precedenza ma questa volta sul iperparametro **max\_depth** che controlla la profondità massima dell'albero di decisione.

depth	acc_mean	acc_std	prec_mean	prec_std	rec_mean	rec_std	f1_mean	f1_std	
0	1	0.4531	0.0122	0.2750	0.0141	0.4540	0.0141	0.3240	0.0084
1	2	0.6959	0.0305	0.7401	0.0488	0.6960	0.0314	0.6547	0.0345
2	3	0.8449	0.0261	0.8555	0.0307	0.8445	0.0263	0.8456	0.0272
3	4	0.8949	0.0335	0.8988	0.0340	0.8948	0.0336	0.8938	0.0342
4	5	0.9031	0.0271	0.9059	0.0271	0.9032	0.0275	0.9021	0.0271
5	6	0.9327	0.0300	0.9347	0.0297	0.9328	0.0300	0.9326	0.0301
6	7	0.9327	0.0224	0.9354	0.0216	0.9328	0.0224	0.9324	0.0223
7	8	0.9378	0.0260	0.9403	0.0248	0.9377	0.0260	0.9375	0.0258
8	9	0.9357	0.0233	0.9380	0.0222	0.9357	0.0232	0.9355	0.0230
9	10	0.9378	0.0252	0.9400	0.0243	0.9378	0.0251	0.9377	0.0250
10	11	0.9388	0.0254	0.9412	0.0245	0.9388	0.0253	0.9387	0.0252
11	12	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
12	13	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
13	14	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
14	15	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
15	16	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
16	17	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
17	18	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
18	19	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258
19	20	0.9398	0.0260	0.9420	0.0250	0.9398	0.0259	0.9398	0.0258

1 row ✓ 1 rows x 4 cols Static Output

	Accuracy	Precisione	Richiamo	F1-score
Media ± std su tutti i k	0.893 ± 0.026	0.889 ± 0.026	0.893 ± 0.026	0.885 ± 0.026



Possiamo osservare che, pur essendo difficile classificare i Pokémon esclusivamente **tramite soglie numeriche rigide**, il Decision Tree riesce comunque a costruire regole efficaci per molte classi, ottenendo metriche complessive elevate ( $F1\text{-macro} \approx 0.85$ ).

Le limitazioni emergono soprattutto nelle classi ibride o sovrapposte, dove algoritmi basati su distanza, come KNN, possono risultare più adatti a catturare la struttura a cluster dei dati.

Dall'analisi della curva di validazione e della tabella dei risultati si osserva che:

- Per valori bassi di profondità (1–3) l'errore in cross validation è elevato: l'albero è troppo semplice e riesce a generalizzare
- Aumentando la profondità, l'errore cala rapidamente fino a stabilizzarsi intorno a **max\_depth = 9**
- Oltre questo valore, l'errore non migliora più in maniera significativa e le metriche (Accuracy, Precision, Recall, F1-score) rimangono praticamente costanti.
- Scegliere una profondità maggiore (es. 15–20) non porta vantaggi ma aumenta il rischio di overfitting, cioè di adattare troppo il modello ai dati di training.

Per questo motivo è stato scelto **max\_depth = 9**, che rappresenta il **miglior compromesso** tra accuratezza e complessità del modello:

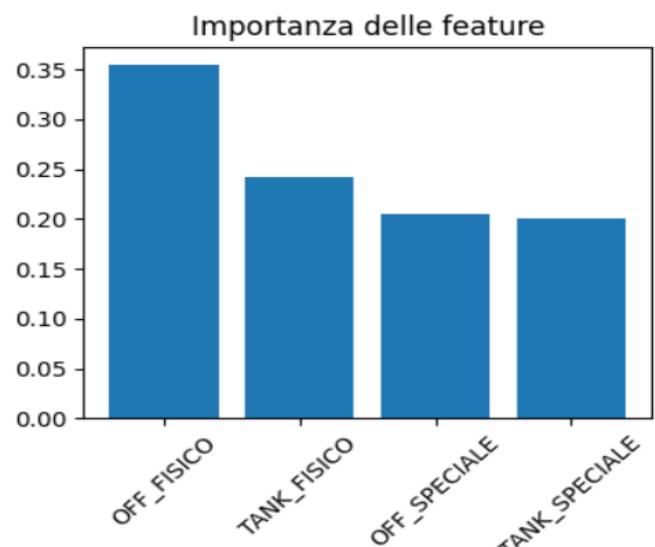
- prestazioni elevate (Accuracy  $\approx 0.93$ , F1  $\approx 0.93$ ),
- variabilità contenuta (deviazione standard bassa)

Analizziamo l'importanza attribuita dall'albero di decisione alle feature per capire come splittare:

L'albero di decisione ha assegnato la maggiore importanza alla feature **OFF\_FISICO**, indicando che la componente offensiva fisica è determinante nella distinzione tra i diversi archetipi.

In seconda posizione troviamo **TANK\_FISICO**, che riflette la capacità difensiva fisica, mentre **OFF\_SPECIALE** e **TANK\_SPECIALE** hanno un peso leggermente inferiore ma comunque significativo. Questo suggerisce che le differenze principali tra archetipi emergono soprattutto dall'attacco fisico, con un contributo rilevante anche dalla difesa fisica.

4 rows		4 rows x 2 cols
	Feature	Importanza
0	OFF_FISICO	0.3541
1	TANK_FISICO	0.2414
2	OFF_SPECIALE	0.2045
3	TANK_SPECIALE	0.1999



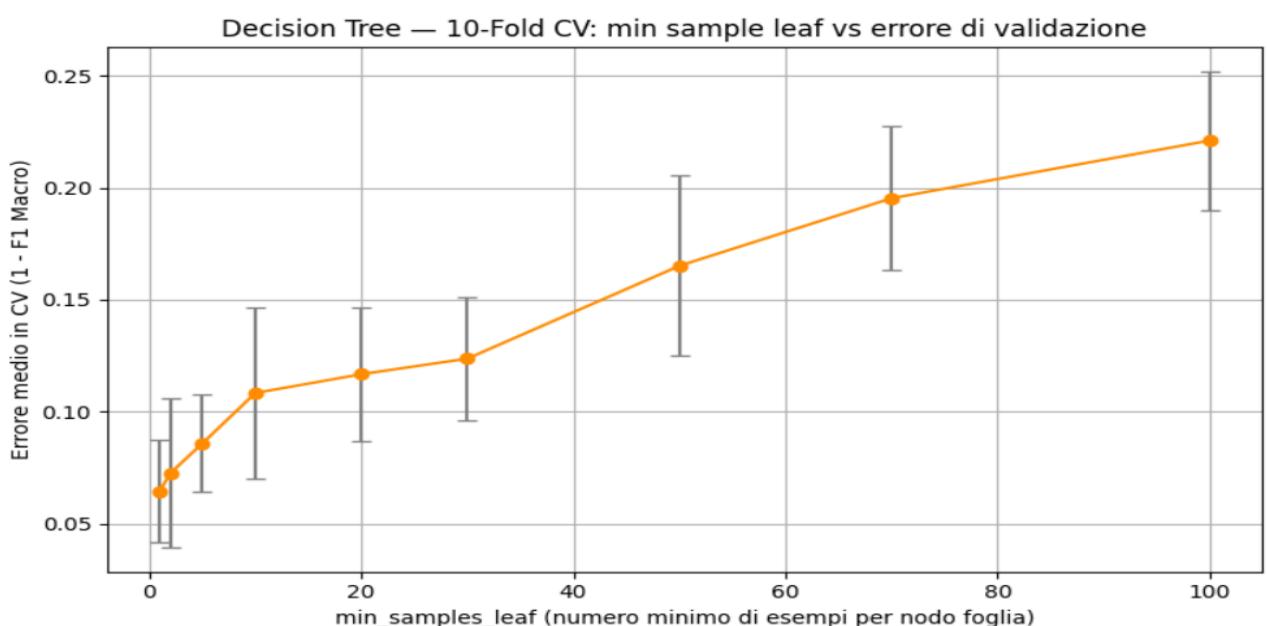
## Albero di decisione con iper-parametro min\_sample\_leaf

Valutiamo le medesime performance, viste per l'albero di decisione, variando l'iperparametro `min_sample_leaf` che impone il numero minimo di esempi che un nodo foglia può contenere. Se dopo uno split un nodo ha meno di `min_samples_leaf` esempi, quello split non viene fatto. In pratica: impedisce che l'albero cresca rami deboli o su dati troppo rari.

Come iperparametro della profondità massima scegliendo il valore ottimale che abbiamo scelto al passo precedente ovvero `max_depth = 9` e valutiamo le performance; mentre testiamo con la ten fold cross validation su questi valori `min_leaf_grid = [1, 2, 5, 10, 20, 30, 50, 70, 100]`

9 rows $\downarrow$ 9 rows $\times$ 9 cols										Static Output
	min_samples_leaf	acc_mean	acc_std	prec_mean	prec_std	rec_mean	rec_std	f1_mean		
0	1	0.9357	0.0233	0.9380	0.0222	0.9357	0.0232	0.9355		
1	2	0.9276	0.0334	0.9300	0.0327	0.9276	0.0333	0.9274		
2	5	0.9143	0.0224	0.9172	0.0210	0.9145	0.0225	0.9142		
3	10	0.8918	0.0388	0.8957	0.0372	0.8918	0.0388	0.8917		
4	20	0.8847	0.0289	0.8876	0.0310	0.8847	0.0294	0.8833		
5	30	0.8786	0.0256	0.8860	0.0254	0.8784	0.0260	0.8763		
6	50	0.8357	0.0394	0.8412	0.0410	0.8355	0.0393	0.8349		
7	70	0.8082	0.0319	0.8251	0.0364	0.8081	0.0323	0.8047		
8	100	0.7786	0.0289	0.7986	0.0290	0.7785	0.0298	0.7789		

1 row $\downarrow$ 1 rows $\times$ 4 cols					Static Output
	Accuracy	Precisione	Richiamo	F1-score	
Media $\pm$ std su tutti i k	0.873 $\pm$ 0.030	0.880 $\pm$ 0.031	0.873 $\pm$ 0.031	0.872 $\pm$ 0.031	



Con valori molto bassi di `min_samples_leaf` (es. 1–2) l’albero ottiene le prestazioni migliori (**Accuracy ≈ 0.93, F1 ≈ 0.93**), perché ha la massima flessibilità.

- All’aumentare del parametro, l’errore medio in cross validation cresce e tutte le metriche calano: l’albero diventa troppo vincolato e perde capacità predittiva (**il modello non riesce a generalizzare**).
- Le deviazioni standard restano contenute, segno che il modello è stabile, ma l’accuratezza scende progressivamente fino a  $\approx 0.78$  con `min_samples_leaf=100`.

In conclusione, questo modello non dimostra nessun miglioramento (come d’altronde è lecito aspettarsi). Questa variante è stata solo proposta per confrontare come possono cambiare le performance del modello variando valori di iper parametri differenti.

## Random Forest (modello Ensemble)

Pur non avendo avuto delle prestazioni eccelse con il decision tree, per via della difficoltà del compito (impostare soglie rigide di split) valutiamo ora il modello che ingloba al suo interno più alberi di decisione per effettuare la predizione e valutiamo se troviamo un miglioramento delle prestazioni.

Ci concentriamo sull'iperparametro che valuta il numero di alberi della foresta ovvero **n\_estimators** che testiamo nel seguente range [10, 25, 50, 75, 100, 150, 200]

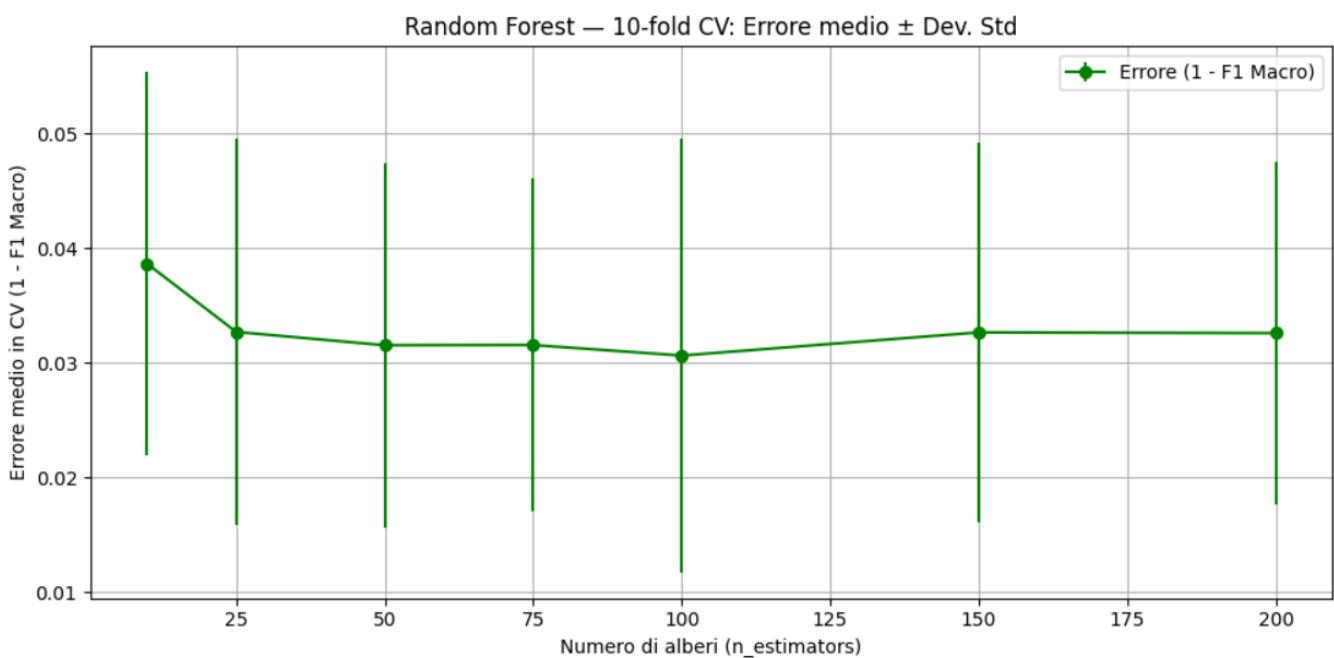
#	n_estimators	accuracy_mean	accuracy_std	precision_mean	precision_std	recall_mean	recall_std
0	10	0.9612	0.0170	0.9633	0.0159	0.9612	0.0169
1	25	0.9673	0.0170	0.9690	0.0160	0.9673	0.0169
2	50	0.9684	0.0161	0.9699	0.0152	0.9685	0.0160
3	75	0.9684	0.0148	0.9697	0.0139	0.9685	0.0147
4	100	0.9694	0.0188	0.9711	0.0173	0.9694	0.0189
5	150	0.9673	0.0163	0.9692	0.0148	0.9673	0.0165
6	200	0.9673	0.0150	0.9691	0.0137	0.9674	0.0150

f1_mean	f1_std
0.9613	0.0168
0.9673	0.0169
0.9685	0.0159
0.9684	0.0146
0.9694	0.0190
0.9673	0.0165
0.9674	0.0150

== Media ± Deviazione Std delle metriche su tutti i n\_estimators ==

4 rows ▾ 4 rows × 2 cols

	media	dev_std
Accuracy	0.9671	0.0027
Precision (macro)	0.9688	0.0025
Recall (macro)	0.9671	0.0027
F1-score (macro)	0.9671	0.0027



Dall'analisi della curva di validazione emerge che:

- All'aumentare del numero di alberi l'errore medio in cross validation ( $1 - F1$  Macro) diminuisce sensibilmente fino a circa  $n_{estimators} = 50-75$ .
- Oltre questo valore, le prestazioni si stabilizzano: aggiungere altri alberi non porta miglioramenti significativi, mentre aumenta il costo computazionale.
- Con 75 alberi si ottiene un ottimo compromesso tra accuratezza, stabilità e tempi di addestramento.

Le metriche medie confermano l'elevata qualità del modello:

- **Accuracy ≈ 0.967,**
- **Precision (macro) ≈ 0.969,**
- **Recall (macro) ≈ 0.967,**
- **F1-score (macro) ≈ 0.967,**  
con deviazioni standard molto basse ( $\approx 0.002-0.003$ ), segno di un comportamento estremamente stabile.

Per questo motivo è stato scelto  **$n_{estimators} = 75$** , in quanto garantisce ottime performance e robustezza, senza complicare inutilmente il modello.

La qualità del modello appreso risulta molto elevata in tutte le metriche.

Si tratta di un notevole passo avanti rispetto all'albero di decisione, come evidenziato non solo dai valori ottenuti, ma anche dalla matrice di confusione calcolata sul **test set** (mai utilizzato in fase di addestramento), che conferma la capacità del modello di generalizzare correttamente anche in presenza di test set con classi sbilanciate.

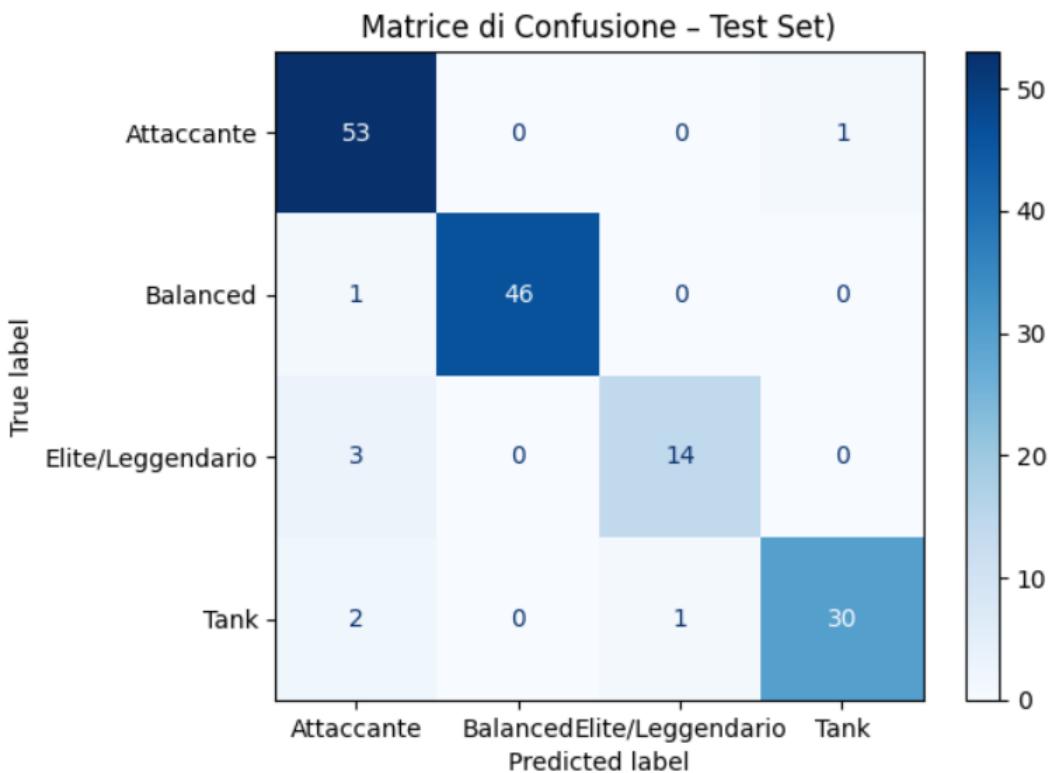
== Performance su test set ==

Accuracy: 0.9470

Precision\_macro: 0.9498

Recall\_macro: 0.9232

F1\_macro: 0.9350

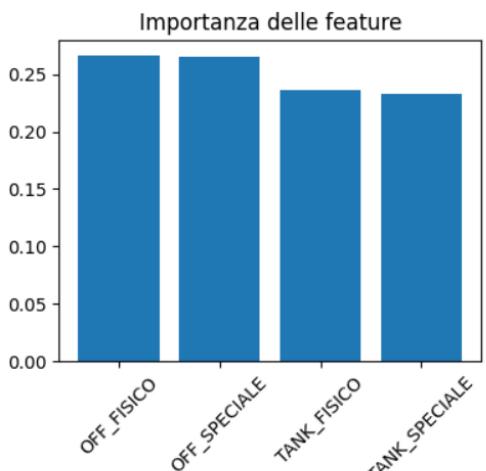


Seppur le metriche calcolate sul test set siano leggermente più basse, in realtà sono perfettamente nel range delle medie che abbiamo prodotto, segno che le metriche medie sono una evidente dimostrazione statistica della bontà del modello appreso.

È possibile anche esaminare le feature importanti assegnati dal randomForest e qui notiamo che il modello ha questa volta assegnato una importanza praticamente identica alle classi.

== Importanza delle feature (valori numerici) ==

4 rows		4 rows x 2 cols
	Feature	Importanza
0	OFF_FISICO	0.2659
1	OFF_SPECIALE	0.2650
2	TANK_FISICO	0.2362
3	TANK_SPECIALE	0.2329



## Valutazione delle performance degli algoritmi che abbiamo visto

Effettuiamo un confronto solo sulla base delle medie prodotte dalle 10-fold-cross validation per fornire una evidenza statistica della bontà del modello

Modello	Accuracy ( $\pm$ std)	Precisione ( $\pm$ std)	Richiamo ( $\pm$ std)	F1-score ( $\pm$ std)
KNN	$0.969 \pm 0.014$	$0.970 \pm 0.014$	$0.969 \pm 0.014$	$0.969 \pm 0.014$
Decision Tree (min_samples_leaf)	$0.873 \pm 0.030$	$0.880 \pm 0.031$	$0.873 \pm 0.031$	$0.872 \pm 0.031$
Decision Tree (max_depth)	$0.893 \pm 0.026$	$0.889 \pm 0.026$	$0.893 \pm 0.026$	$0.885 \pm 0.026$
Random Forest	$0.967 \pm 0.003$	$0.969 \pm 0.003$	$0.967 \pm 0.003$	$0.967 \pm 0.003$

Conclusione finale: Dall'analisi comparativa emerge che la Random Forest e il KNN sono i modelli con le prestazioni migliori e più stabili, con metriche intorno al 97% e deviazioni standard molto basse. Il Decision Tree con max\_depth ottimizzato migliora rispetto alla versione con min\_samples\_leaf, ma rimane inferiore agli altri modelli. In generale, i modelli ensemble come la Random Forest offrono il miglior compromesso tra accuratezza e robustezza, mentre il KNN si conferma un modello semplice ma altamente efficace in questo contesto con questo dataset processato in tal modo.

## Regressione logistica (multi classe)

Il modello della regressione logistica (multiclasse) è stato addestrato su un dataset con feature numeriche proveniente nella sezione dove abbiamo utilizzato il clustering. L'idea è di utilizzare le distanze dal valore delle feature numeriche rispetto al valore medio nel centroide del cluster e allenare un modello di regressione logistica sulla base di questo dataset. Sorprendentemente le prestazioni di questo modello si sono rivelate ottime a dimostrare la bontà delle feature di input.

Abbiamo, anche in questo caso, utilizzato l'iperparametro  $C$  (nel range [0.01, 0.1, 1, 10, 100]) per valutare in media le prestazioni del modello con la 10 fold cross validation. Tuttavia, essendo un dataset diverso dal precedente non è possibile confrontare le prestazioni di questo modello con i modelli precedenti.

	C	accuracy_mean	accuracy_std	precision_mean	precision_std	recall_mean	f1_mean	f1_std
0	0.01	0.8800	0.0446	0.9196	0.0312	0.8444	0.8689	0.0546
1	0.10	0.9550	0.0248	0.9649	0.0208	0.9356	0.9461	0.0332
2	1.00	0.9767	0.0186	0.9805	0.0157	0.9601	0.9672	0.0288
3	10.00	0.9817	0.0157	0.9844	0.0145	0.9718	0.9758	0.0271
4	100.00	0.9900	0.0082	0.9897	0.0099	0.9880	0.9884	0.0105

recall_std	f1_mean	f1_std
0.0609	0.8689	0.0546
0.0379	0.9461	0.0332
0.0340	0.9672	0.0288
0.0316	0.9758	0.0271
0.0116	0.9884	0.0105

Miglior valore di C (max F1 macro): 100.0

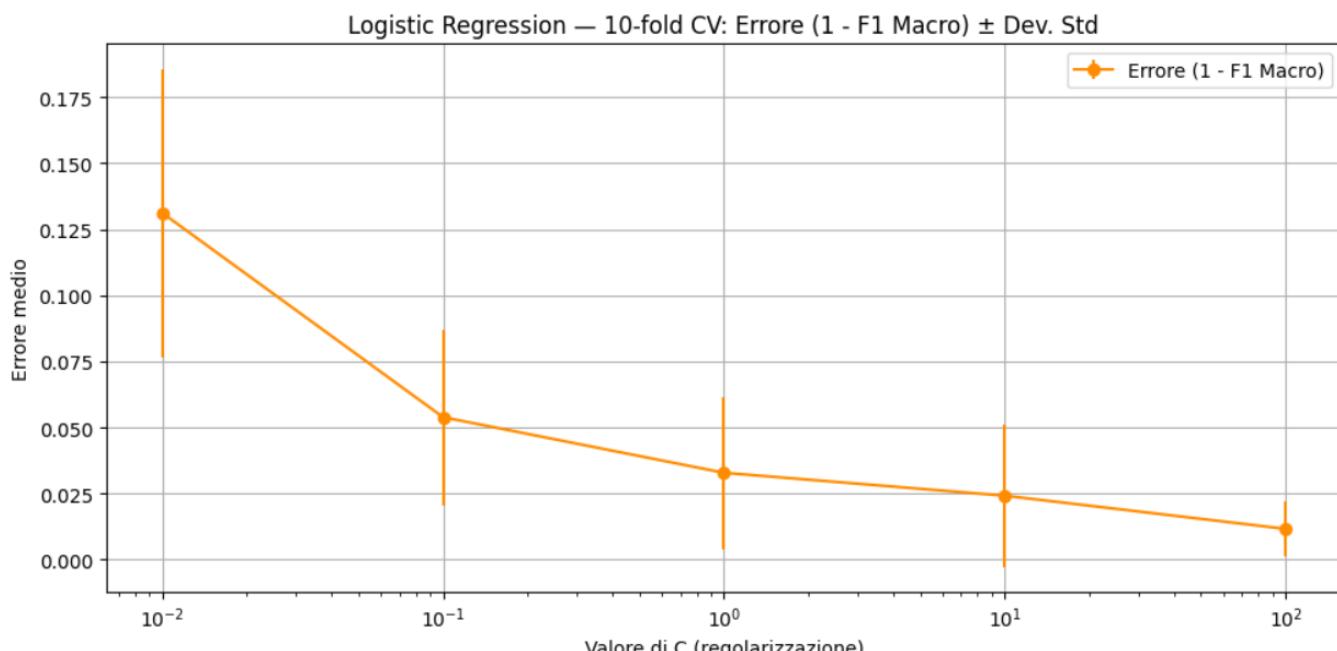
Media ± Std delle metriche su tutti i C:

4 rows ▾ 4 rows × 2 cols

	mean	std
Accuracy	0.9567	0.0448
Precision (macro)	0.9678	0.0285
Recall (macro)	0.9400	0.0567
F1-score (macro)	0.9493	0.0475

- **Accuracy = 0.957 ± 0.045** → il modello classifica correttamente circa il 96% dei casi.
- **Precision (macro) = 0.969 ± 0.029** → quando predice una classe, la quasi totalità delle predizioni sono corrette.
- **Recall (macro) = 0.940 ± 0.057** → riesce a individuare la grande maggioranza degli esempi reali di ogni classe, anche se leggermente meno rispetto alla precision.
- **F1-score (macro) = 0.949 ± 0.048** → ottimo equilibrio tra precision e recall, con una deviazione standard contenuta

Le performance sono **molti elevate e stabili**. La regressione logistica con C=100 (spiego dopo perché ho scelto 100 come iper parametro) si dimostra un classificatore solido, che supera le prestazioni viste con Decision Tree e si avvicina a quelle della Random Forest, mantenendo però il vantaggio di essere più semplice e interpretabile.



nb

68:13 (35 chars, 1 line break) LF UTF-8 ⌂ 4 spaces Pvlt

Guardando i tuoi risultati del grafico:

- l'errore medio ( $1 - F1$  Macro) diminuisce in modo costante all'aumentare del parametro **C**, stabilizzandosi su valori molto bassi per  **$C \geq 10$** .
- La tabella indica che con  **$C = 100$**  si ottengono le migliori prestazioni, con metriche elevate e bilanciate:
  - **Accuracy ≈ 0.957**
  - **Precision (macro) ≈ 0.969**
  - **Recall (macro) ≈ 0.940**
  - **F1-score (macro) ≈ 0.949**

La scelta di  **$C = 100$**  è quindi giustificata perché:

- massimizza l'F1-macro,
- riduce al minimo l'errore medio,
- mantiene la deviazione standard contenuta, segno di buona stabilità,
- rappresenta un compromesso ideale tra complessità del modello e capacità predittiva.

## Conclusione

Il progetto ha dimostrato come l'integrazione di **clustering non supervisionato, modelli supervisionati e arricchimento semantico** consenta di individuare archetipi coerenti di Pokémon e classificarli con elevata precisione.

Tra i modelli testati, la **Random Forest** si è confermata la soluzione più performante e stabile in questo dominio, mentre l'arricchimento di **PokémonKG** ha potenziato le capacità di ragionamento sul dominio rendendo possibili nuove query SPARQL prima non possibili.