AI report

Marco Caruso – `Matricola: 65836`
Massimiliano Carello – `Matricola: 61059`
Giacomo Alberto Napolitano – `Matricola: 51578`

A.A. 2025-2026

1 Tesina 1

1.1 Introduzione-Spiegazione dei 3 scenari

L'obiettivo di questa tesina è stato quello di ottimizzare la manutenzione e il rifornimento di 20 stazioni di ricarica elettrica in una rete urbana interconnessa.

Si è modellata quindi la rete urbana in un grafo, nel quale:

- I nodi sono le venti stazioni, identificate con un numero univoco
- I pesi delle connessioni identificano la tipologia di collegamento. In particolare, il peso pari a 120 identifica un collegamento "standard", mentre il peso pari a 180 un collegamento "cross-area". Questo rappresenta il costo/distanza del collegamento.
- Sono stati aggiunti degli ulteriori pesi per poter tenere in considerazione anche il tempo impiegato e l'energia consumata. In particolare, si è scelto di modellare tempo ed energia proporzionalmente alla distanza, in base ai due casi presenti:

Collegamento	Coef. tempo	Coef. energia
Standard	1	0.8
Cross-area	0.5	1.4

Questo ha permesso di modellare i collegamenti cross-area come strade lunghe ma ad alto scorrimento e ad alto dispendio energetico, mentre i collegamenti "standard" come strade a lento scorrimento e a basso dispendio energetico.

Il problema si articola nella risoluzione di tre diversi scenari, in termini di obiettivo di minimizzazione e vincoli da soddisfare, il tutto descritto nella tabella seguente:

Scenario	Start	Goal	Vincoli
Emergenza	1	17	Max energia: 800 Max tempo: 400
Manutenzione	6	$5 \rightarrow 12 \rightarrow 8$	Ordine Distanza totale: 1800
Bilanciamento Energetico	11	{5,10,14,18}	Distanza totale: 1400

Tutti e tre gli scenari sono stati risolti utilizzando l'algoritmo A* con opportune modifiche in base allo scenario utilizzato.

1.2 Implementazione euristiche

L'algoritmo prevede di utilizzare una $g(n)$, rappresentante il costo sostenuto per arrivare al nodo n partendo dal nodo iniziale, e una funzione $h(n)$, che rappresenta l'euristica utilizzata per stimare il costo del percorso dal nodo `n` al `goal`.

Come euristica di base si è inizialmente pensato di implementare la distanza euclidea. Nonostante dei risultati iniziali promettenti è stato possibile verificare che, in un particolare caso, questa metrica non è ammissibile (vedi 1.5.2). Questo è dovuto al fatto che i costi degli archi non sono perfettamente coerenti con la geometria: nel caso riportato, i due nodi (6 e 14) sono geometricamente più vicini ma il percorso è più costoso rispetto all'alternativa (da 7 a 14). Dunque, utilizzare la distanza euclidea porterebbe a preferire il percorso più lungo invece di quello ottimo: l'euristica non è ammissibile.

Si è deciso quindi di trovare un'euristica alternativa ammissibile. Per ottenere sempre una stima a ribasso, si è partiti dalle seguenti osservazioni:

- si assume che tutti gli archi abbiano peso pari al peso minimo (nel caso specifico, tutti archi con peso pari a 120)
- si assume che il percorso scelto per andare da un nodo A ad un nodo B sia il percorso che attraversa il minor numero di archi

In questo modo, la stima della distanza del percorso da fare è sempre minore o uguale alla lunghezza del percorso minimo reale, rendendo così quest'euristica sempre ammissibile.

Per calcolare il numero minimo di archi attraversati in un qualsiasi percorso, è stato utilizzato un algoritmo di BFS[1]. Lo pseudocodice dell'euristica risulta essere:

[1] Si è scelto di calcolare dinamicamente il numero minimo di archi tramite BFS. Nonostante la BFS abbia un costo computazionale superiore alla semplice distanza euclidea, il suo utilizzo è ampiamente giustificato dalla dimensione contenuta del grafo (20 nodi). In un grafo di queste dimensioni, il tempo di esecuzione della BFS è trascurabile e non impatta sulle performance complessive di A*, garantendo al contempo l'ammissibilità necessaria per l'ottimalità dell'algoritmo. Una possibile ottimizzazione futura sarebbe quella di pre-calcolare tutte le distanze prima di eseguire l'algoritmo A*

```
shortest_path = BFS(current, goal
shortest_path_length = len(shortest_path)-1
h = 120 * shortest_path_length
```

Questa euristica viene utilizzata nei vari scenari:

- Negli scenari di manutenzione e bilanciamento viene utilizzata l'euristica così come descritta sopra, senza apportare nessun cambiamento, in quanto l'obiettivo di minimizzazione è la distanza.
- Nello scenario di emergenza, invece, si desidera minimizzare il tempo, dunque l'euristica in questo caso dovrà rappresentare il tempo stimato. Per questo motivo, si è scelto di utilizzare la distanza, così com'è stato descritto precedentemente, moltiplicata per il coefficiente di tempo minimo definito (in questo caso 0,5). L'euristica risulta essere:

```
h = 0.5 * 120 * shortest_path_length
```

1.3 Implementazione A*

Per risolvere tutti e tre gli scenari si è implementato dunque un algoritmo A*.

Per lo scenario di "Manutenzione" e di "Bilanciamento energetico", l'algoritmo segue la versione "classica" dove lo stato è definito soltanto dal nodo e come costo si è utilizzato il peso dell'arco. Questo perché in questi scenari l'obiettivo era quello di minimizzare la distanza del percorso, rappresenta dal peso dell'arco.

Nello scenario di "Emergenza", l'obiettivo è stato quello di trovare il percorso a tempo minimo rispettando i vincoli di energia e tempo massimo. Per questo motivo, l'algoritmo A* differisce da quello utilizzato per gli scenari precedenti. In particolare, una prima modifica riguarda la funzione di costo `f`, in quanto adesso misura il tempo e non più la distanza. Nello specifico:

- `g(n)` rappresenta il tempo necessario per arrivare dal nodo di partenza al nodo `n`
- `h(n)` è una stima ammissibile del tempo necessario per arrivare al goal partendo dal nodo `n`

Un'altra fondamentale differenza sta nello stato utilizzato dall'algoritmo: infatti, in questo scenario, lo stato è definito dalla coppia `(nodo, energia)` e non più dal solo nodo. Questo perché il percorso non dipende solo dal nodo visitato, ma anche dalla quantità di energia consumata per arrivarci.

L'ultima differenza risiede nell'introduzione di un criterio di dominanza: se una stazione è stata raggiunta con una quantità di energia minore o uguale, eventuali nuovi stati che arrivano a quella stazione con un'energia maggiore verranno scartati.

Ricapitolando, l'estrazione dalla coda ordinata per tempo garantisce la minimizzazione del costo temporale, mentre la dominanza permette di scartare stati meno promettenti in termini di energia, riducendo lo spazio di ricerca.

In tutti casi l'algoritmo prevede i seguenti step:

1. Inizializzazione dell'`open_set`, e dei dizionari di supporto
2. Ciclo principale che itera `open_set`:
   - estraendo il nodo con `f-score` più basso;
   - controllando se è il nodo finale, se si allora si ricostruisce e ritorna il percorso
   - altrimenti esamina i vicini, aggiornando, se necessario, i dizionari di supporto

Di default si esegue la versione "classica" dell'A*, ma si può settare alto un flag quando si istanzia la classe `AStarPathfinder` per selezionare l'esecuzione con stato esteso.

1.4 Logica risolutiva

1.4.1 Scenario "Manutenzione"

Poiché nello scenario "Manutenzione" l'ordine di visita è definito, si è scelto di chiamare l'algoritmo A* (con `flag = 0`) 3 volte su "subpath" ($6 \to 5$; $5 \to 12$; $12 \to 8$), ottenendo poi il percorso finale concatenando i percorsi restituiti dalle 3 iterazioni. Una volta risolto lo scenario, si genera un report in formato `.txt` contenente tutte le informazioni riguardanti la soluzione ottenuta.

Si controlla infine la validità della soluzione (verificando che il vincolo sulla distanza sia rispettato) una volta ottenuto il percorso finale. Poiché in questo scenario l'ordine di visita è fisso e l'euristica è ammissibile, il costo minimo del percorso totale equivale alla somma dei singoli sottopercorsi minimi.

1.4.2 Scenario "Bilanciamento energetico"

Poiché nello scenario "Bilanciamento energetico" l'ordine di visita non è definito, si è scelto di implementare un approccio *greedy*. In particolare, anche in questo caso si esegue l'algoritmo A* più volte, ma ogni volta seleziono il goal corrente dando priorità ai nodi più vicini e più critici.

Dunque, prima di eseguire l'algoritmo, si assegna ai goal ancora da raggiungere il seguente score:
`criticità(g)=1-livello_energia_normalizzato(g)`
`score(n,g)=distanza_euclidea(n,g)/criticità(g)`
Si seleziona il goal corrente come il goal che ha ottenuto uno score più basso.

Scelto il goal, si esegue l'algoritmo di A* (con `flag = 0`), che restituisce il percorso da `current` $\to$ `goal`. Si ripete questo processo (assegnazione score $\to$ selezione goal $\to$ A*) finché la lista dei goal non è vuota.

Una volta risolto lo scenario, si genera un report in formato `.txt` contenente tutte le informazioni riguardanti la soluzione ottenuta.

Si controlla infine la validità della soluzione(verificando che il vincolo sulla distanza sia rispettato) una volta ottenuto il percorso finale.

Si osserva, inoltre, che questo tipo di approccio ha dei limiti: data la natura *greedy* dell'algoritmo, questo in particolari casi potrebbe non trovare una soluzione ottima globalmente, com'è possibile vedere nell'esempio riportato in 1.5.3

1.4.3 Scenario "Emergenza"

In questo scenario si utilizza l'A* (con `flag = 1`), quindi con la versione stato esteso. In particolare, durante

l'esecuzione dell'algoritmo vengono controllati anche i vincoli temporali ed energetici, scartando a priori nodi che porterebbero a violarli.

Dunque, il percorso restituito, se esiste, soddisferà sicuramente i vincoli imposti dal problema.

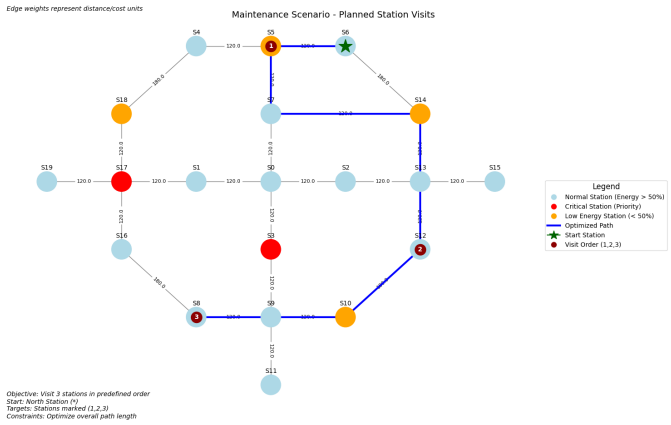
Una volta risolto lo scenario, si genera un report in formato `.txt` contenente tutte le informazioni riguardanti la soluzione ottenuta in caso di percorso trovato, oppure conterrà tutti i percorsi valutati ma scartati per violazione dei vincoli.

1.5 Risultati

1.5.1 Risultati scenari regolari

Si mostrano di seguito i percorsi trovati nei 3 scenari, sia utilizzando l'algoritmo di A* descritto sopra e sia utilizzando l'algoritmo di Dijkstra. [2]

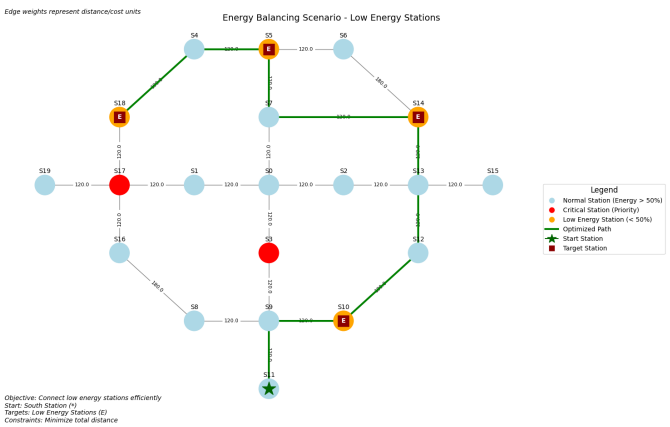
Scenario Manutenzione



Il grafico completo è disponibile al seguente indirizzo

	A*	Dijkstra
Costo	1020	1020
Energia	924	924
Tempo	930	930
Lunghezza Percorso	9	9
Nodi Esplorati	12	21

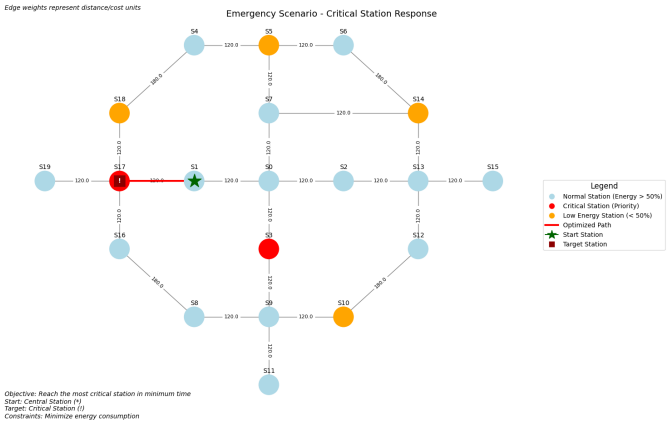
Scenario Bilanciamento energetico



Il grafico completo è disponibile al seguente indirizzo

	A*	Dijkstra
Costo	1020	1020
Energia	1176	1176
Tempo	1020	1020
Lunghezza Percorso	10	10
Nodi Esplorati	13	29

Scenario Emergenza



Il grafico completo è disponibile al seguente indirizzo

	A*	Dijkstra
Costo	120	120
Energia	96	96
Tempo	120	120
Lunghezza Percorso	2	2
Nodi Esplorati	2	3

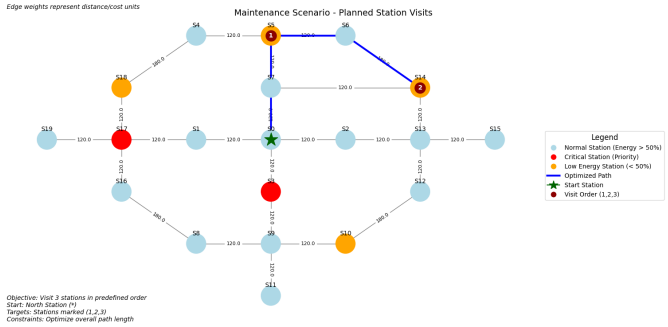
[2] Per maggiori dettagli sui risultati, per ogni simulazione mostrata di seguito è presente una visualizzazione grafica della soluzione ed un report testuale al seguente link: Link Github-Tesina 1

1.5.2 Risultati scenario ”Manutenzione” - caso limite

Si riporta ora un caso costruito ad hoc per mostrare l’inammissibilità dell’euristica basata sulla distanza euclidea. In particolare i dettagli dello scenario sono:

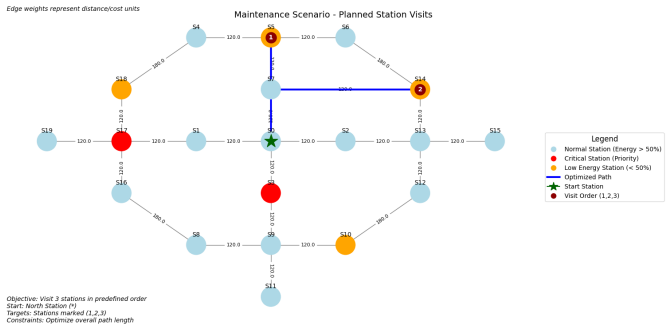
- **start**: 0
- **goals**: [5, 14]
- **vincolo**: ordine di visita

Utilizzando la distanza euclidea, il percorso trovato per questo scenario è il seguente:



Il grafico completo è disponibile al seguente indirizzo

Usando invece l’euristica spiegata in 1.2 si ottiene:



Il grafico completo è disponibile al seguente indirizzo

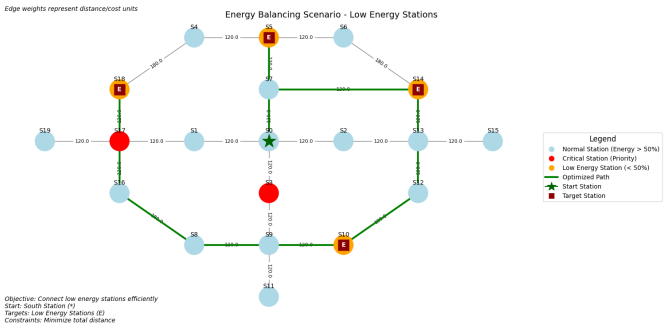
Com’è possibile notare dai due risultati, l’euristica utilizzata consente di trovare correttamente il percorso minimo, mentre la distanza euclidea sovrastima il costo minimo reale.

1.5.3 Risultati scenario ”Bilanciamento” - caso limite

In questa sezione viene sottolineato il limite dell’approccio greedy andando ad analizzare un particolare caso limite:

- **start**: 0
- **goals**: [5, 10, 14, 18]
- **livelli di energia**: i livelli di energia del nodo 5 e del nodo 18 sono stati invertiti
- **vincoli**: distanza < 1800

Il risultato ottenuto è il seguente:



Il grafico completo è disponibile al seguente indirizzo

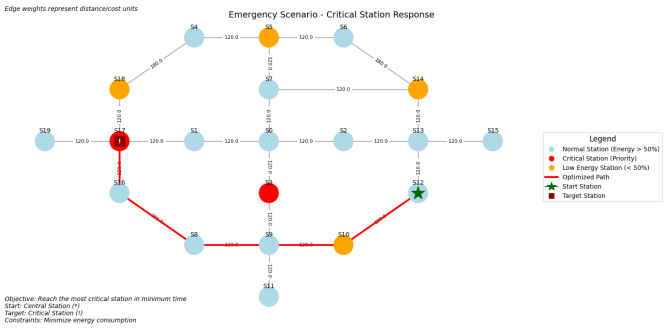
Si osserva come il percorso scelto sia sub-ottimo: attraverso una visione globale è possibile notare come il percorso ottimale si ottenga scegliendo come primo goal il nodo 18. Tuttavia, il nodo 5 è più vicino ed ha un livello di energia più basso e viene quindi scelto come primo goal dall’algoritmo.

1.5.4 Risultati scenario ”Emergenza” - caso limite

In questo particolare caso viene mostrato come l’algoritmo A*, utilizzando l’euristica descritta in 1.2, scelga un percorso più lungo ma più veloce, coerente con la richiesta.

- **start:** 12
- **goal**: 17
- **vincoli**: energia < 800 ; tempo < 500

Il risultato ottenuto è il seguente:



Il grafico completo è disponibile al seguente indirizzo

Si nota quindi come viene scelta la strada che prevede il passaggio per due archi cross-area (lunghi ma veloci) invece del passaggio per gli archi standard (più corti ma più lenti).

2 Tesina 2

2.1 Introduzione-GA

L'obiettivo di questa tesina è stato quello di implementare un algoritmo genetico (GA) e utilizzarlo per effettuare feature selection sul dataset DARWIN, per lo studio dell'Alzheimer.

Il dataset è composto da 450 features che descrivono le caratteristiche di scrittura a mano di un soggetto.

Volendo utilizzare un GA è stato necessario definire:

- **Individuo** (classe `Individual`): per questo problema un individuo è rappresentato da un vettore binario, di 450 elementi, dove ogni elemento è una feature del dataset e quindi rappresenta se quella feature è stata selezionata o meno.
- **Metodi di selezione**: sono stati implementati due metodi possibili di selezione:
  - **Tournament selection** (funzione `tournament_selection`): Seleziona k individui random e restituisce il migliore.
  - **Roulette wheel selection** (funzione `roulette_wheel_selection`: Seleziona l'individuo con probabilità proporzionale alla fitness
- **Metodo di crossover e metodo di mutazione**: come metodo di crossover è stato implementato un **single-point-crossover** (funzione `single_point_crossover`) mentre come metodo di mutazione un **bit-flip-mutation** (funzione `bit_flip_mutation`).
- **classe GA** (classe `GeneticAlgorithm`): quest'ultima implementare l'algoritmo vero e proprio, specificando i seguenti aspetti:
  - Funzione di fitness (metodo `evaluate_population`): si è scelto di utilizzare una fitness basata sulla correlation analysis, considerando quindi correlazione features-classe e features-features. Un buon subset è caratterizzato da alta correlazione con la classe e bassa ridondanza.
  - Ciclo GA (metodo `run`): implementazione di una run (di `max_generations` generazioni) dell'algoritmo che prevede:
    1. Valutazione della popolazione e logging delle risultati
    2. Controllo della convergenza
    3. Elitismo: conservazione dell'individuo più promettente
    4. Selezione dei genitori e generazione dell'offspring tramite crossover
    5. Applicazione della mutazione
    6. Sostituzione della popolazione

Volendo studiare sistematicamente l'effetto dei parametri sono stati definiti i seguenti scenari:

- Scenario 1-Dimensione popolazione (funzione `run_experiment_population_size`): si è analizzato l'algoritmo andando a utilizzare diverse dimensioni della popolazione. In particolare si sono testati i seguenti valori: $[20, 50, 100, 200, 500]$. Per questo scenario i parametri fissi sono:
  - Crossover rate 0.8
  - Mutation rate 0.1
- Scenario 2-Operatori genetici (funzione `run_experiment_genetic_operators`): si è analizzato l'algoritmo andando a variare crossover rates, mutation rates e metodi di selezione. In particolare:
  - Crossover rates: $[0.6, 0.7, 0.8, 0.9]$
  - Mutation rates: $[0.01, 0.05, 0.1, 0.15]$
  - Tournament selection con $k = 2, 3, 4$ e Roulette Wheel

  Per questo scenario il parametro fisso è la dimensione della popolazione pari a 100
- Scenario 3-Criteri di Stop (funzione `run_experiment_stopping_criteria`): si è analizzato l'algoritmo andando a variare:
  - Numero di generazioni fisse, tra $[50, 100, 200]$
  - Convergenza con soglie tra $[10, 20, 30]$ generazioni e tolleranze tra $[1e-4, 1e-5, 1e-6]$

Nelle prossime sezioni sono riportati i risultati ottenuti per ogni scenario (eseguito su 30 run) e le relative considerazioni[3].

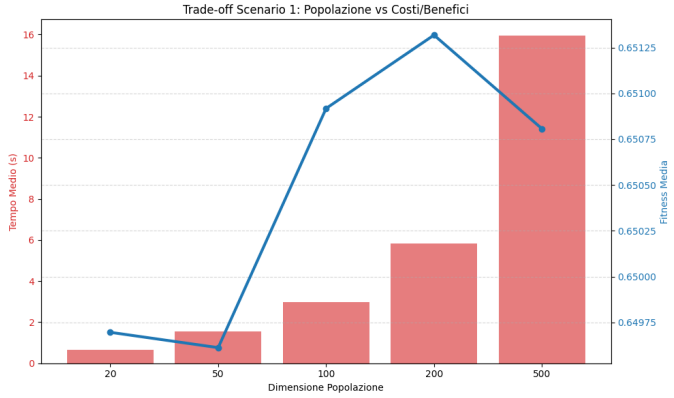
2.2 Scenario 1

Come descritto in precedenza l'obiettivo di questo scenario è quello di valutare le performance dell'algoritmo variando la dimensione della popolazione.

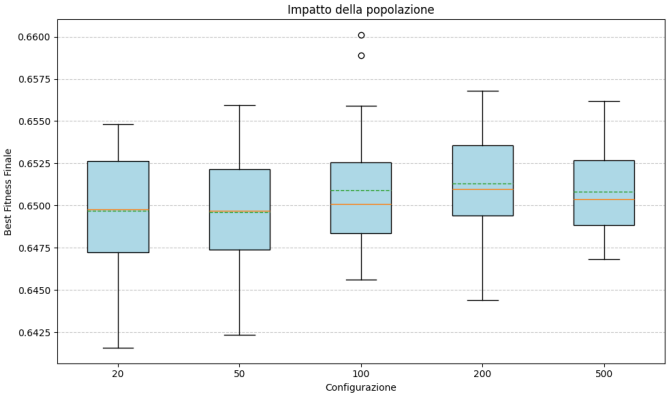
Dalla teoria ci si aspetta che una configurazione con popolazione ridotta si possa "bloccare" in un minimo locale, mentre configurazioni con popolazioni maggiori riescano ad esplorare di più lo spazio delle soluzioni facendo crescere di più la fitness. Ciononostante, popolazioni troppo grandi potrebbero richiedere risorse eccessive rispetto ai reali benefici in termini di fitness.

A supporto di questa considerazione teorica è stato generato il seguente plot che mostra il costo temporale e il guadagno in fitness variando la dimensione della popolazione (funzione `plot_population_tradeoff`):

[3] Nella repo github è possibile visionare i grafici mostrati di seguito, e di accedere ai file `.pkl` con i dati "grezzi" ottenuti per ogni scenario



Il grafico completo è disponibile al seguente indirizzo



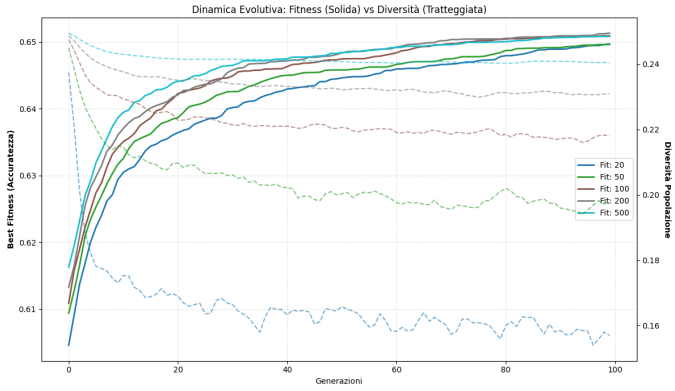
Il grafico completo è disponibile al seguente indirizzo

Come si può osservare aumentando la popolazione crescono anche i costi computazionali:

- `Pop=20`: $\sim$ 0.6s per run
- `Pop=50`: $\sim$ 1.6s (2.7x)
- `Pop=100`: $\sim$ 3.0s (5x)
- `Pop=200`: $\sim$ 5.8s (9.7x)
- `Pop=500`: $\sim$ 15.9s (26.5x)

La lieve riduzione della fitness osservata per `pop=500` può essere spiegata considerando che i risultati sono mediati su diversi metodi di selezione (tournament con diversi valori di `k` e roulette wheel). In particolare, metodi a bassa pressione selettiva come la roulette wheel, o tournament con `k` ridotto, risultano meno efficaci nel favorire rapidamente gli individui migliori quando la popolazione è molto grande. Questo porta a una maggiore probabilità di riproduzione di individui mediocri e ad una convergenza più lenta, rendendo il guadagno in fitness marginale rispetto all'aumento del costo computazionale..

Come ulteriori analisi, si riportano il grafico "convergenza-diversità" e "box plot distribuzioni fitness": (funzione `plot_convergence_and_diversity` e `plot_fitness_boxplots`)



Il grafico completo è disponibile al seguente indirizzo

Dal grafico della convergenza si nota come la configurazione con `pop=200,500` hanno una partenza più "esplosiva" (dovuta alla maggiore capacità di esplorazione), ma durante la fase intermedia tutte le configurazioni convergono andandosi poi ad assestare su 2 massimi diversi:

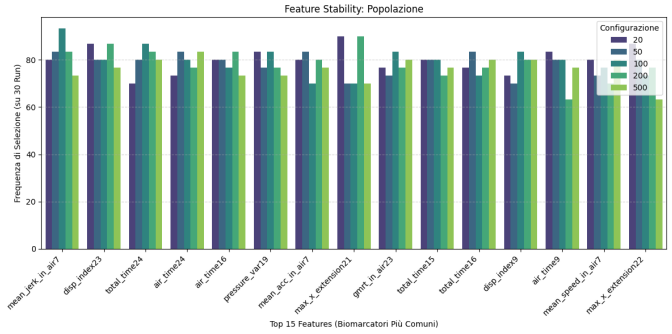
- `Pop=20,50` converge a $\sim$ 0.649
- `Pop=100,200,500` converge a $\sim$ 0.656

Queste considerazioni si riflettono anche sull'andamento delle curve di diversità. Infatti, più è grande la popolazione e più si ha una maggiore diversità durante la run, prevenendo una convergenza prematura dell'algoritmo.

Per quanto riguarda il boxplot, si osserva che nessuna configurazione è significativamente superiore alle altre, coerentemente con le curve di convergenza. Ovviamente la configurazione con pop=20 è più instabile, con maggiore variabilità poiché dipende fortemente dal seed iniziale: con pochi individui, una popolazione iniziale sfortunata ha meno possibilità di recuperare.

In conclusione, si identifica come punto ottimale di lavoro una dimensione di popolazione pari a $100 - 200$ in quanto rappresenta il migliore compromesso tra qualità (in termini di fitness) ed efficienza (in termini di esecuzione temporale). Si conclude però che la dimensione della popolazione non è il fattore critico (in termini di fitness) per questo problema, in quanto tutte e cinque le configurazioni si assestano a un valore di fitness circa pari a 0.65.

In aggiunta si riportano le frequenze di selezione delle 15 features più selezionate durante tutti gli esperimenti (funzione `plot_feature_frequency_comparison`):



Il grafico completo è disponibile al seguente indirizzo

2.3 Scenario 2

Come descritto in precedenza l'obiettivo di questo scenario è quello di valutare le performance dell'algoritmo variando gli operatori genetici dell'algoritmo.

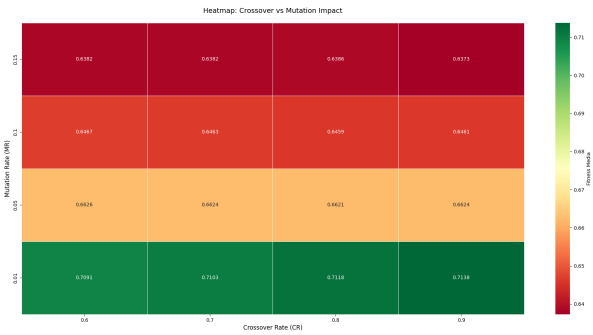
Ci si aspetta che configurazioni con crossover rate più alto e mutation rate basso abbiano delle performance migliori. Questo perché un crossover rate alto permette di generare popolazioni più diverse, mentre un mutation rate non elevato permette di "uscire" da minimi locali esplorando nuove zone nello spazio, senza però divergere a un random walk (mutation rate troppo alto).

Per quanto riguarda i metodi di selezione, ci si aspetta che le configurazioni migliori siano caratterizzate dal metodo di selezione "tournament" poiché imprime una maggiore pressione selettiva.

A supporto di questa considerazione teoriche si riportano i seguenti grafici:

- Heatmap "Crossover rate vs Mutation rate", che mostra come varia la fitness media cambiando i rate degli operatori genetici (funzione `plot_heatmap_operators`)

- Boxplot dei risultati ottenuti variando i metodi di selezione (funzione `plot_fitness_boxplots`)

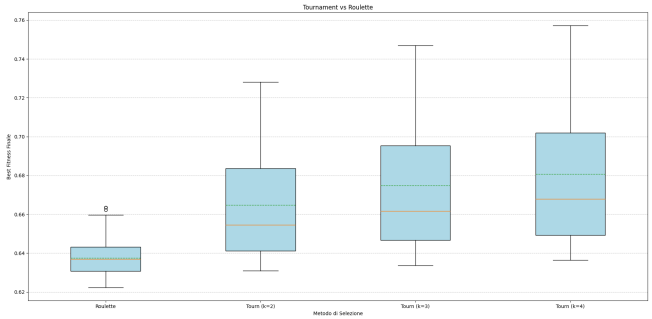


Il grafico completo è disponibile al seguente indirizzo

Dalla heatmap si deduce che il parametro fondamentale per questa applicazione è il mutation rate. Questo perché si osserva che fissato un MR, il crossover rate ha poco effetto, cioè ha una influenza marginale sulla fitness, tendenzialmente positivo.

Viceversa, se il mutation rate cresce, la fitness diminuisce in modo monotono, questo perché troppa mutazione rischia di distruggere buone combinazioni di features già trovate.

La configurazione migliore è quella con `MR=0.01` e `CR>=0.8`.

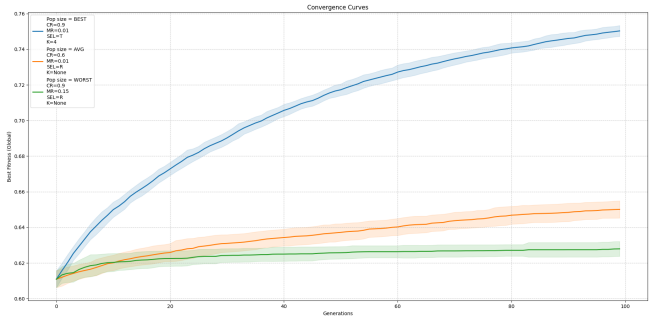


Il grafico completo è disponibile al seguente indirizzo

Dall'analisi dei boxplot si comprende come la roulette è un metodo stabile ma mediocre, poiché ha mediana bassa e box molto stretto (e quindi bassa varianza). Questo significa che è un metodo più conservativo, che esplora di meno soprattutto quando le fitness non sono molto differenziate.

Tournament selection applica invece una maggiore pressione selettiva, garantendo una maggiore qualità al crescere di k, in termini di massimi, ma varianza più alta. Quindi un tournament selection con $k = 3$ sembra un ottimo compromesso, in quanto ha quasi le stesse performance di $k = 4$, ma pressione selettiva migliore e quindi meno rischio di convergenza prematura.

Per evidenziare in modo chiaro l'impatto combinato degli operatori genetici si riporta il seguente grafico che rappresenta le curve di convergenza della configurazione migliore, media e peggiore trovata (funzione `plot_convergence_curves`):



Il grafico completo è disponibile al seguente indirizzo

Si osserva quindi:

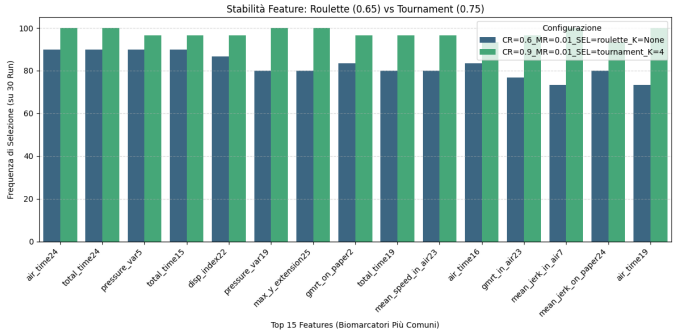
- La configurazione BEST (`CR = 0.9, MR = 0.01, selezione a torneo con k = 4`) mostra una crescita rapida e monotona della fitness globale, con una convergenza progressiva verso valori significativamente più elevati rispetto alle altre configurazioni. L'andamento regolare e le bande di variabilità contenute mostrano un processo evolutivo stabile, in grado di sfruttare in modo efficiente le soluzioni promettenti trovate già dalle prime generazioni

- La configurazione AVG (`roulette, MR = 0.01`) presenta invece una crescita più lenta e un plateau anticipato. Pur mantenendo una certa stabilità, il metodo di selezione meno aggressivo limita la capacità del GA di migliorare ulteriormente la qualità delle soluzioni, confermando quanto osservato nei boxplot relativi ai metodi di selezione.

- La configurazione WORST (`roulette e MR elevato`) evidenzia una convergenza precoce verso valori di fitness inferiori, con miglioramenti marginali nel corso delle generazioni. Questo comportamento suggerisce che un'eccessiva mutazione compromette la conservazione delle strutture genetiche utili, rendendo inefficace il processo evolutivo anche in presenza di un crossover elevato.

In conclusione, i risultati mostrano come il Mutation Rate rappresenti il parametro più critico in questo contesto, insieme alla scelta di un metodo di selezione sufficientemente selettivo. Quindi una configurazione caratterizzata da Mutation Rate basso (~ 0.01), Tournament Selection con pressione moderata ($k = 3$–4) e Crossover Rate elevato (≥ 0.8) emerge come una scelta robusta ed efficace.

In aggiunta si riportano le frequenze di selezione delle 15 features più selezionate durante gli esperimenti più promettenti che utilizzano tournament e roulette (funzione `plot_feature_frequency_comparison`):



Il grafico completo è disponibile al seguente indirizzo

2.4 Scenario 3

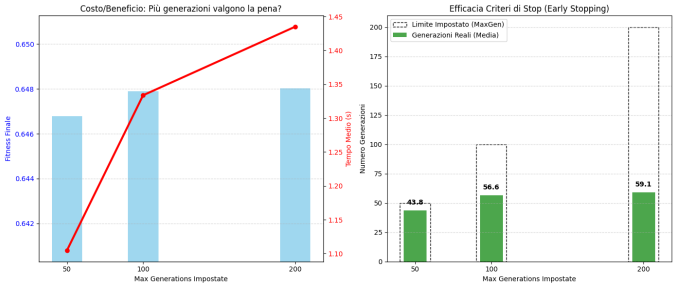
Quest'ultimo scenario ha l'obiettivo di analizzare l'efficacia dei criteri di stop.

L'algoritmo prevede due parametri principali:

- Numero massimo di generazioni per run (`max_gen`). Questo sicuramente ha un impatto temporale ma permette di far migliorare la fitness.

- Soglia di convergenza: cioè il numero minimo di generazioni successive nelle quali la fitness rimane costante (considerando una determinata soglia di tolleranza). Questo parametro permette di fermare l'esecuzione dell'algoritmo anche prima di aver raggiunto il massimo numero di generazioni previsto, ottimizzando così il costo computazionale.

Per mostrare l'andamento della fitness e il costo temporale dell'algoritmo si è generato il seguente grafico (funzione `plot_efficiency_and_stopping`):

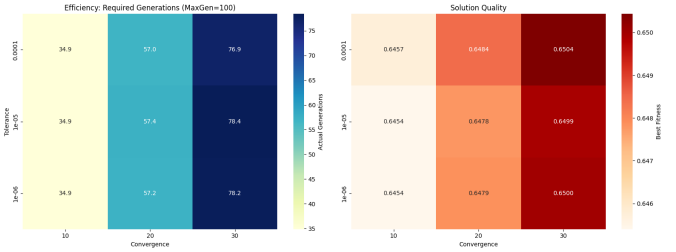


Il grafico completo è disponibile al seguente indirizzo

Come si può osservare dal plot di sinistra, aumentare il numero di massimo di generazioni permette di migliorare la fitness, ma superata la soglia di `max_gen=100`, si ha un aumento temporale non trascurabile mentre la fitness non migliora in maniera significativa.

Il plot di destra invece mostra l'efficacia dei criteri di stop anticipato e si osserva come mediamente ci si fermi intorno alla 60-esima generazione, anche se si aumenta il numero massimo di generazioni.

Per studiare l'effetto della soglia di convergenza e di tolleranza si riporta la seguente heatmap (funzione `plot_stopping_heatmap`:



Il grafico completo è disponibile al seguente indirizzo

La heatmap di sinistra mostra l'effetto dei criteri di stop rispetto al numero di generazioni effettivamente eseguite. In particolare si osserva che:

- il valore di tolleranza non influenza in maniera significativa l'andamento dell'algoritmo

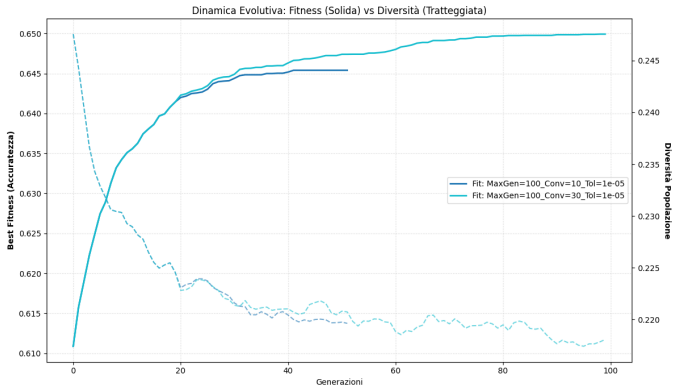
- il parametro che davvero permette un early stop è la soglia di convergenza.

La heatmap di destra, invece, mostra come viene influenzata la fitness. Anche in questo caso la tolleranza ha un effetto molto marginale, mentre la soglia di convergenza più alta permette di raggiungere fitness migliori, ma di poco. Infatti, passando da una soglia di 10 a una di 30 si guadagna circa +0.005 di fitness (da 0.645 a 0.650), ma si raddoppia il tempo di calcolo (da 35 a 77 generazioni).

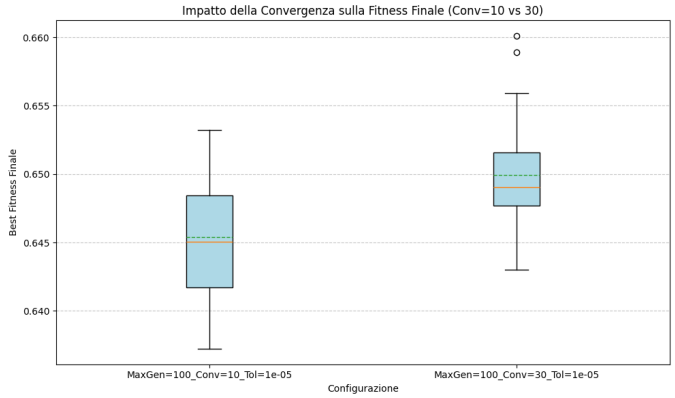
Dai risultati ottenuti, si mostrano gli andamenti delle due configurazioni più rappresentative, cioè il caso estremo dove ci si ferma "subito" (dopo 10 gen uguali), e il caso estremo dove si attende di più (servono 30 gen uguali per fermarsi) (funzione `plot_convergence_and_diversity` e `plot_fitness_boxplots`)

- `max_gen=100, Conv=10, Tol=1e-05`

- `max_gen=100, Conv=30, Tol=1e-05`



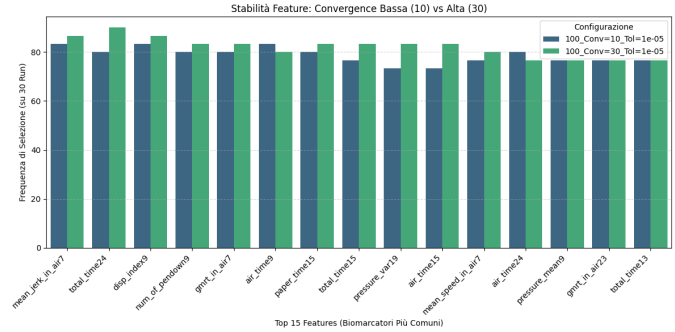
Il grafico completo è disponibile al seguente indirizzo



Il grafico completo è disponibile al seguente indirizzo

La configurazione `Convergence=20` rappresenta il migliore compromesso. Con una media di 57 generazioni reali, permette all'algoritmo di superare i minimi locali senza sprecare risorse computazionali per guadagni infinitesimali come avviene con soglia 30.

Infine, anche in questo caso è stata fatta un'analisi sulla frequenza di selezione delle features, riportata nel seguente grafico (funzione `plot_feature_frequency_comparison`):



Il grafico completo è disponibile al seguente indirizzo

2.5 Conclusioni

Sulla base dei tre scenari, la configurazione raccomandata per applicazioni pratiche è:

Parameter	Value
Population Size	100–200
Crossover Rate	0.8–0.9
Mutation Rate	0.01
Selection Method	Tournament ($k = 3$)
Max Generations	100
Convergence Threshold	20
Tolerance	1×10^{-5}

Questa configurazione garantisce:

- Tempi di esecuzione accettabili, in quanto la popolazione non è troppo elevata e l'utilizzo di criteri di early stopping permettono di "risparmiare" sulle generazioni finali, che non avrebbero portato a un miglioramento significativo della fitness.
- Prestazioni elevate, in quanto si utilizza un criterio di selezione fortemente selettivo, andando a bilanciare in maniera efficace esplorazione di nuove soluzioni e sfruttamento di soluzioni buone già trovate.

Eseguiti tutti gli esperimenti sono stati individui i "top 20 biomarcatori", cioè le 20 features più selezionate dalla configurazione migliore.

In particolare la configurazione è caratterizzata da:

- `CR=0.9`
- `MR=0.01`
- Metodo di selezione: tournament con $k = 4$
- Numero di run: 30

Le 20 feature più selezionate sono state:

Rank	Feature	Freq	%
1	mean_jerk_in_air7	30	100.0
2	air_time15	30	100.0
3	disp_index17	30	100.0
4	air_time19	30	100.0
5	pressure_var19	30	100.0
6	air_time24	30	100.0
7	total_time24	30	100.0
8	max_y_extension25	30	100.0
9	gmrt_on_paper2	29	96.7
10	pressure_var5	29	96.7
11	air_time6	29	96.7
12	mean_acc_in_air7	29	96.7
13	mean_speed_in_air7	29	96.7
14	air_time13	29	96.7
15	total_time15	29	96.7
16	num_of_pendown19	29	96.7
17	total_time19	29	96.7
18	max_x_extension21	29	96.7
19	disp_index22	29	96.7
20	gmrt_in_air23	29	96.7

mentre il numero medio di feature selezionate è pari a 133, rappresentando una significativa riduzione dalle 450 feature iniziali.

La persistenza di queste feature suggerisce la loro efficacia e affidabilità per distinguere i pazienti sani da quelli affetti da Alzheimer.

3 Tesina 3

L'obiettivo di questa tesina è stato l'implementazione di una rete neurale al fine di studiare l'impatto dei suoi parametri sulle performance di classificazione nella diagnosi del cancro al seno.

Per questo motivo si è utilizzato il dataset *Wisconsin Breast Cancer* caratterizzato da:

- Features estratte da immagini di nuclei cellulari
- Classificazione binaria (maligno/benigno)
- 30 features, numeriche
- $\sim$ 570 istanze

Al fine di studiare in maniera sistematica l'impatto dei parametri della rete, sono stati identificati i seguenti tre scenari:

- Scenario 1- Architettura e Attivazione, nel quale sono stati analizzati le seguenti configurazioni:
  - Configurazioni layer:
    - Singolo layer: [(50,), (100,), (200,)]
    - Due layer: [(50,25), (100,50), (200,100)]
    - Tre layer: [(100,50,25), (200,100,50)]
  - Funzioni di attivazione:
    - 'identity'
    - 'logistic'
    - 'tanh'
    - 'relu'
- Scenario 2-Learning Rate e Ottimizzazione:
  - Learning rates iniziali [0.0001, 0.001, 0.01, 0.1]
  - Learning rate policies:
    - 'constant'
    - 'invscaling'
    - 'adaptive'
  - Solvers:
    - 'adam'
    - 'sgd'
    - 'lbfgs'
  - Batch sizes: [32, 64, 128, 256]
- Scenario 3-Regolarizzazione:
  - Valori alpha: [0.0001, 0.001, 0.01, 0.1]
  - Early stopping
  - Validation split: [0.1, 0.2, 0.3]
  - N_iter_no_change: [5, 10, 20]

Per ogni scenario, i parametri non specificati sono stati impostati con i valori di default.

Di seguito, viene mostrato come è stato strutturato il codice, i risultati e le analisi di ogni scenario e conclusioni tratte.

3.1 Codice implementato

Si è scelto di implementare il codice di base in maniera modulare in modo tale che potesse essere utilizzato facilmente per tutti gli scenari.

Per questo motivo, il codice prodotto può essere diviso nelle seguenti macro-categorie:

- Gestione dataset (funzione `load_dataset`)
- Implementazione di una classe che gestisce l'intero ciclo di un singolo esperimento (classe `MLPAlgo`).
  In particolare, poiché era richiesto memorizzare le informazioni di accuracy e loss per epoca, è stato necessario organizzare la run di un esperimento nei seguenti livelli:
  1. Livello più esterno, si cicla sulle 30 run
  2. Livello intermedio: si applica la cross-validation con 5 fold
  3. Livello basso: loop manuale sulle epoche.

  Per accedere alle singole epoche, ma utilizzando sempre *Sklearn*, l'MLP istanziato ha come parametri fissi:
  - `max_iter = 1`
  - `warm_start = True`
  - `early_stopping = False`

  In questo modo, ogni volta che si chiama la funzione `.fit` si esegue una sola iterazione, controllando se esistono già dei pesi ottimizzati in una chiamata precedente.
  Inoltre, l'early stopping è stato gestito manualmente, in quanto questo era richiesto solo per i primi due scenari.

  La funzione `run_experiment` restituisce un dizionario di questo tipo:
  - `params`: contiene i parametri originali dello scenario (ad esempio architettura dei layer, funzione di attivazione, learning rate, ecc.).
  - `data`: è una lista di lunghezza pari al numero di run (in questo caso 30). Ogni elemento della lista corrisponde ai risultati di una singola run ed è a sua volta un dizionario con le seguenti chiavi:
    - `accuracies`: lista delle accuracy finali per ciascun fold della cross-validation.
    - `y_true`: lista dei vettori contenenti le etichette reali del validation set per ciascun fold.
    - `y_pred`: lista dei vettori contenenti le predizioni finali per ciascun fold.
    - `y_proba`: lista dei vettori contenenti le probabilità predette per la classe positiva per ciascun fold.

    * `losses`: lista di liste, contenente per ciascun fold la loss registrata ad ogni epoca.
    * `train_scores_per_epoch`: lista di liste, contenente per ciascun fold l'accuracy sul training set ad ogni epoca.
    * `val_scores`: lista di liste, contenente per ciascun fold l'accuracy sul validation set ad ogni epoca.
    * `fit_times`: lista dei tempi di addestramento totali per ciascun fold.

  In sintesi, la struttura permette di analizzare l'andamento dell'addestramento sia a livello di epoca (curve di apprendimento), sia a livello di fold e run (stabilità e generalizzazione del modello).

- Una classe `Logger` per la generazione di un report testuale alla fine di ogni scenario
- Solver per ogni scenario (`scenario_architettura_att`, `scenario_learning_rate_ott`, `scenario_regolarizzazione`): per ogni scenario vengono specificate le variabili di interesse (ad esempio numero di neuroni e funzione di attivazione, learning rate e policy, o parametri di regolarizzazione) e viene chiamata la classe `MLPAlgo` per eseguire 30 run indipendenti per ciascuna configurazione.
  La funzione restituisce un dizionario di alto livello (`results_scenario`) con la seguente struttura:
  - le chiavi sono stringhe descrittive della configurazione (ad esempio `"Arch:(100,50)_Act:relu"`);
  - i valori sono i dizionari restituiti da `run_experiment`, contenenti sia i parametri della configurazione (`params`) sia i risultati delle 30 run (`data`), con tutte le metriche per fold ed epoca.

  In questo modo, `results_scenario` permette di confrontare facilmente tutte le configurazioni testate e di analizzare l'andamento delle performance del modello sia a livello di run che di cross-validation e di singola epoca.
- Visualizzazione: sono stati implementati diverse funzioni per la visualizzazione di: curve di apprendimento, matrici di confusione, curve ROC, boxplot e barchar.
- Estrazione dati: funzioni ausiliare necessarie per estrarre soltanto alcune configurazioni di interesse da tutti i risultati ottenuti.
- Main: prevede due modalità:
  - Una modalità "esperimento", nella quale vengono eseguiti tutti gli esperimenti per ogni scenario, i cui risultati vengono salvati in dei file `.pkl`.
    Inoltre viene generato un report testuale per ogni scenario.
  - Una modalità "visualizzazione": partendo dai risultati ottenuti, memorizzati nei file `.pkl`, vengono estratte configurazioni di interesse e visualizzati su grafici opportuni[4].

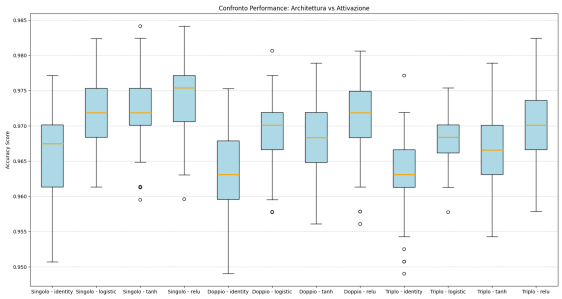
3.2 Scenario 1

L'obiettivo del primo scenario è stato quello di analizzare l'impatto dell'architettura della rete in termini di:

- profondità
- numero di neuroni
- funzione di attivazione

3.2.1 Analisi architettura

Si è condotta una prima analisi macroscopica, con l'obiettivo di capire la profondità delle rete ideale per questo problema. Per questo motivo, i risultati ottenuti dallo scenario 1 sono stati raggruppati per profondità (Singolo, Doppio e Triplo layer) e distinti per funzione di attivazione, generando il seguente boxplot[5]:



Il grafico completo è disponibile al seguente indirizzo

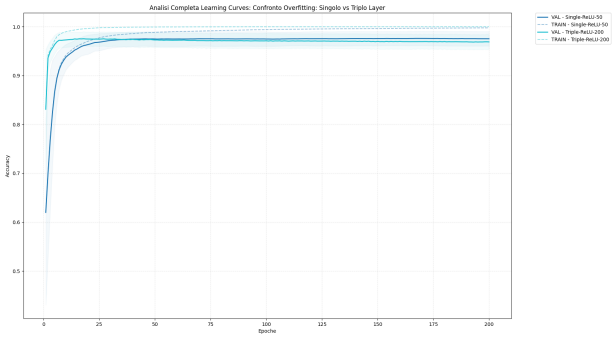
Si è dedotto quindi che:

- La funzione di attivazione `ReLU` è superiore per tutte le configurazioni.
- La funzione `identity` sia sempre la peggiore: questo è giustificato dal fatto che è una funzione lineare e quindi non riesce a catturare le correlazioni complesse del dataset.
- L'architettura a singolo layer nonostante sia quella "più semplice" ha ottenuto le performance migliori.

Per analizzare in maniera più precisa l'effetto della profondità è stato generato la seguente curva di apprendimento, metto a confronto, singolo e triplo layer:

[4] Nella repo github è possibile visionare i grafici mostrati di seguito. Putroppo non è stato possibile caricare anche in questo caso i file `.pkl` per via della loro dimensione

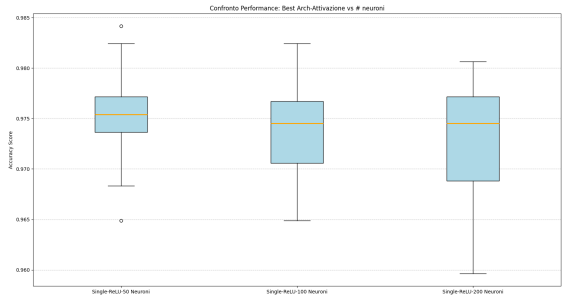
[5] Si noti che l'architettura a tre layer presenta un numero inferiore di configurazioni testate (2 architetture invece di 3) rispetto alle configurazioni a singolo e doppio layer. Questo sbilanciamento è stato mantenuto per aderire alle specifiche del progetto e non inficia significativamente la validità del confronto generale sulla profondità della rete.



Il grafico completo è disponibile al seguente indirizzo

Si comprende come in questo caso, una rete triplo layer rischia di "memorizzare" le specifiche istanze del training set (overfitting) a discapito delle capacità di generalizzazione su dati nuovi. Infatti, nel grafico precedente si può notare come la distanza tra la curva di training e quella di validation (nel caso triplo layer) tenda a crescere, fornendo un segnale evidente di inizio overfitting. Questo suggerisce che su un dataset di queste dimensioni ($\sim$ 570 istanze), aumentare eccessivamente la profondità può portare a instabilità nell'apprendimento o a un inizio di overfitting

Dopo aver stabilito che `singolo layer+ReLU` è la combinazione migliore, è stata svolta un'analisi microscopia sull'impatto del numero di neuroni per questa configurazione, generando il seguente boxplot:



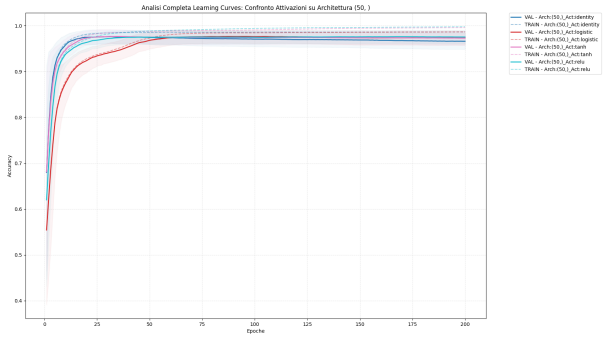
Il grafico completo è disponibile al seguente indirizzo

Si osserva come i boxplot per 50, 100, 200 neuroni sono quasi allineati sulla stessa mediana (circa 0.97) indicando, quindi, che per il dataset in esame una rete con funzione di attivazione ReLU a singolo layer con 50 neuroni è già sufficiente per separare bene le due classi. Aumentare il numero di neuroni non migliora le performance in maniera significativa ma aumenta solo il numero di pesi da calcolare.

Dunque, già da questa prima analisi si potrebbe affermare che la configurazione più efficiente, in ambito medico, prevede di usare `ReLU-Single layer (50)`.

3.2.2 Analisi convergenza-stabilità

Data la configurazione più promettente (`Single layer (50)`) si è voluto studiare l'effetto in termini di convergenza e stabilità delle 4 funzioni di attivazione, utilizzando il seguente grafico:



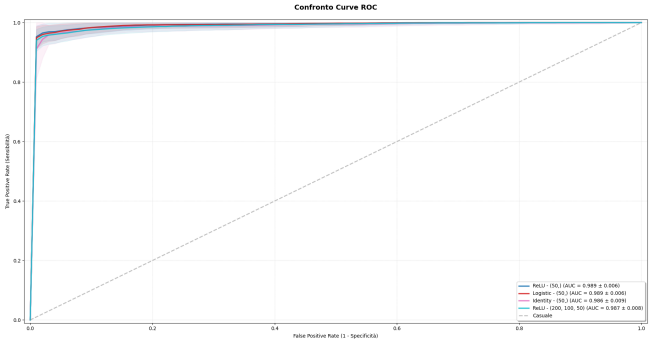
Il grafico completo è disponibile al seguente indirizzo

Per quanto riguarda la velocità di convergenza le funzioni `ReLU` e `Tanh` sono quelle più efficienti, in quanto già intorno alla 30-esima si è raggiunto un plateau di accuratezza.

La `Logistic`, invece, è quella più lenta, in quanto impiega circa 75 epoche per raggiungere la stessa accuratezza che le altre ottengono in un terzo del tempo. Questo è corretto, in quanto le funzioni sigmoidee tendono a saturare più facilmente, rallentando l'aggiornamento dei pesi.

Per quanto concerne capacità di generalizzazione tutte configurazioni analizzate non soffrono di overfitting. Similmente, si osserva che tutte le configurazioni dopo la fase iniziale sono caratterizzate da una varianza minima, suggerendo che l'accuratezza è robusta rispetto ai diversi fold della cross-validation.

Per effettuare un'ulteriore analisi sull'architettura e sulla funzione di attivazione, è stata generata la seguente curva ROC:



Il grafico completo è disponibile al seguente indirizzo

Il grafico prevede il confronto tra:

- La configurazione migliore in assoluto: `ReLU - (50,)`
- La "peggiore" funzione di attivazione: `Identity - (50,)`
- La configurazione "più lenta": `Logistic - (50,)`
- La configurazione più complessa: `Relu - (200, 100, 50)`

Analizzando questa curva si è ottenuta la conferma che le funzioni di attivazione non lineari sono più opportune per questo dataset, e che aggiungere layer non migliora le prestazioni ma anzi, rischia di degradarle.

3.2.3 Analisi temporale

L'ultima analisi effettuata riguarda l'impatto dell'architettura sul costo temporale. Si è generato

il seguente plot dove si mostra:

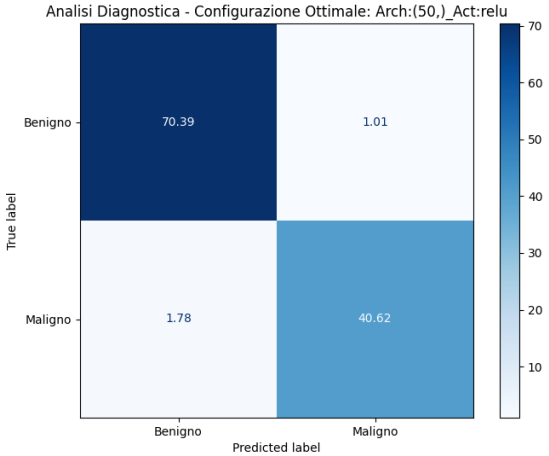
- L'accuratezza media per architettura (mediata su tutte le configurazioni) tramite il barplot
- L'andamento temporale medio per run

Oltre alla conferma dell'inutilità di una rete troppo profonda (in termini di accuracy), si osserva anche come il tempo medio per run cresca ripidamente.

In definitiva quest'analisi conferma che la configurazione **single layer (50,)-ReLU** non è solo la più precisa ma anche la più efficiente.

3.2.4 Capacità diagnostica

Compreso quale sia la configurazione migliore per questo scenario è stata generata la seguente matrice di confusione:



Il grafico completo è disponibile al seguente indirizzo

La configurazione in esame ha un'accuratezza media di 0.9755 (la più alta) e una deviazione standard molto bassa (0.0040), il che significa che è stabile e quindi adatta per un'applicazione medicale. Dalla matrice inoltre si osserva:

- Falsi Negativi (FN) pari a 1.78. Questo è il dato più critico per questa applicazione. La rete utilizzata però ha un FN molto vicino allo zero e quindi rappresenta un ottimo strumento di screening.

3.2.5 Conclusioni-Scenario 1

Dalle analisi effettuate si deduce quindi che la configurazione ottimale è **single layer (50,)-ReLU** in quanto:

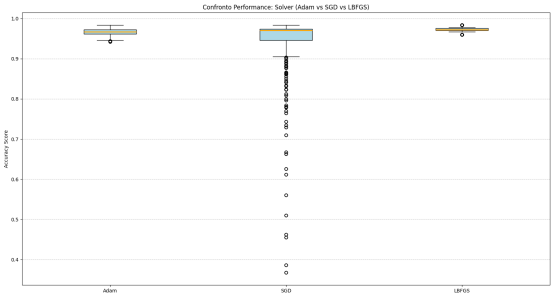
- Massimizza l'AUC (0.989).
- Minimizza i FN
- Evita l'overfitting tipico delle reti più profonde su dataset di queste dimensioni.
- È computazionalmente efficiente, raggiungendo la soluzione in meno tempo.

3.3 Scenario 2

Il seguente scenario mira ad analizzare l'impatto dei parametri legati al learning rate e di ottimizzazione sulla rete.

3.3.1 Impatto del solver

Seguendo lo stesso approccio di analisi usato per lo Scenario 1, si è iniziato lo studio con un'analisi macroscopica, questa volta sull'effetto dei solver, generando quindi il seguente boxplot:

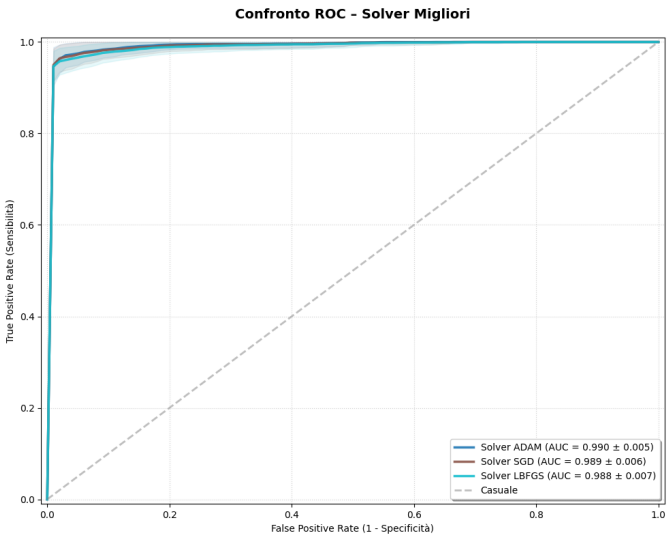


Il grafico completo è disponibile al seguente indirizzo

Si nota che i solver `LBFGS` e `Adam` siano quelli più stabili e precisi mentre `SGD` ha un'alta variabilità. Questo risultato è supportato dalle seguenti considerazioni teoriche:

- LBFGS: Essendo un metodo quasi-Newtoniano (del secondo ordine), tende a convergere molto velocemente e con precisione su dataset piccoli, mostrando una varianza minima[6].
- SGD: Estremamente sensibile al learning rate e al rumore dei gradienti, quindi ci aspettiamo una "scatola" molto allungata (alta variabilità).
- Adam: utilizzando momenti adattivi, rappresenta il giusto compresso tra performance e stabilità.

Per conclure l'analisi sui solver si è generata la seguente curva ROC per ogni solver con la propria configurazione migliore:



Il grafico completo è disponibile al seguente indirizzo

In sintesi, tutti e tre i solver ottengono risultati eccellenti, quasi vicini alla "classificazione perfetta". Le differenze

[6] Questo solver non utilizza `batch_size` ma lavora sull'intero dataset, ottenendo una stabilità quasi perfetta in quanto a differenza degli altri solver non subisce l'oscillazione dovuta alla stocasticità dei batch

tra loro sono statisticamente minime, come indicato dai margini di errore sovrapposti.

3.3.2 Analisi del solver SGD

In prima battuta, sono state estrapolate le configurazioni peggiori ottenute con il solver `SGD`, con l'obiettivo di individuare dei pattern comuni. In particolare si riportano quelle con accuracy < 0.6:

LR	Policy	Batch	Accuracy
0.0001	constant	256	0.367
0.0001	invscaling	256	0.367
0.0001	adaptive	256	0.367
0.0001	constant	256	0.387
0.0001	invscaling	256	0.387
0.0001	adaptive	256	0.387
0.0001	constant	256	0.455
0.0001	invscaling	256	0.455
0.0001	adaptive	256	0.455
0.0001	constant	256	0.462

Notiamo quindi che in tutte le configurazioni è stato utilizzato un learning rate iniziale pari a 0.0001 e dimensione del batch pari a 256, ottenendo così una convergenza troppo lenta. In particolare:

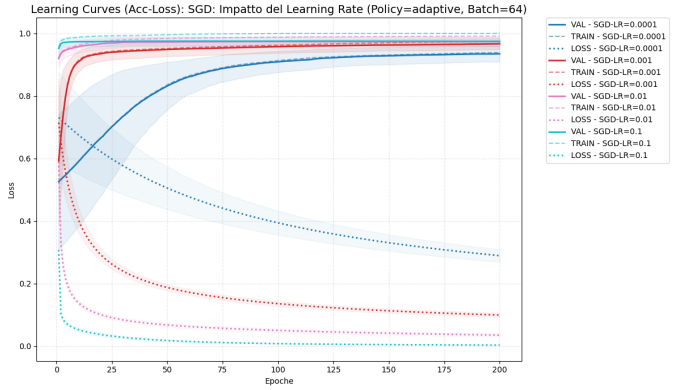
- Un learning rate così basso implica che la rete fa piccolissimi passi nella direzione del gradiente.
- Batch size così grande implica che con un dataset con 569 istanze, e 5 fold abbiamo circa 455 istanze per il training. Utilizzando però un batch di 256 si effettuano soltanto due update per epoca e quindi si aggiornano i pesi soltanto due volte per epoca

In definitiva, utilizzando questa combinazione il modello non ha modo di imparare.

Per approfondire l'instabilità del solver `SGD`, si sono isolate le curve di apprendimento (Accuracy sul training, sul validation, e loss function) per la seguente configurazione:

- Solver: SGD
- Policy: adaptive
- Batch: 64
- Learning rate: [0.0001, 0.001, 0.01, 0.1]

ottenendo il seguente grafico:



Il grafico completo è disponibile al seguente indirizzo

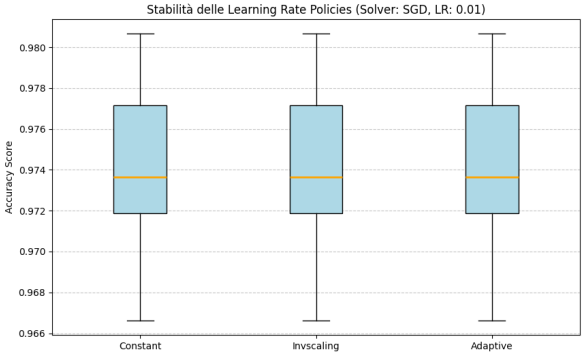
Per quanto riguarda la velocità di convergenza si nota come:

- `LR=0.1` è la configurazione "più aggressiva" in quanto la loss crolla quasi istantaneamente a zero e l'accuracy raggiunge il plateau in meno di 10 epoche.
- `LR=0.0001`: è troppo basso per questo numero di epoche. Infatti la curva è molto lenta, e a 200 epoche non ha ancora raggiunto le prestazioni delle altre configurazioni. Si nota inoltre che è caratterizzato da un'area di varianza molto ampia, segno che l'apprendimento è instabile o molto dipendente dal seed/fold.

Riguardo l'overfitting il gap tra la curva di accuracy nel validation test e training test è minimo per qualunque configurazione, suggerendo che che il modello generalizza bene e non sta soffrendo di overfitting marcato.

Infine, si osserva come learning rate intermedi rappresentano un ottimo compromesso in quanto sono molto stabili e raggiungono prestazioni finali elevate (molto simili a `LR=0.1`) ma con un andamento più "sano".

Per concludere l'analisi sull'instabilità del solver `SGD` si è voluto isolare l'effetto della learning rate policy, analizzando il seguente grafico:

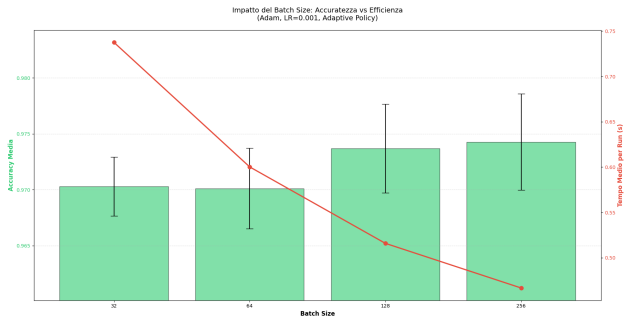


Il grafico completo è disponibile al seguente indirizzo

Se ne deduce che, poiché il dataset è composto da soltanto 570 istanze e le classi sono "facilmente" separabili dalla rete, il modello raggiunge il minimo globale così velocemente che la policy (che dovrebbe servire a correggere il tiro in situazioni difficili) non ha tempo di entrare in funzione e per questo motivo i tre boxplot sono identici.

3.3.3 Impatto temporale del batch size

Come fatto nello scenario 1, anche in questo si è voluto fare uno studio sull'andamento temporale ma questa volta in funzione del `batch_size`. Per non "sporcare" i dati con l'instabilità cronica di SGD, si utilizza Adam come solver di riferimento (anche perché LBFGS non usa il batch size), fissando come learning rate 0.001:



Il grafico completo è disponibile al seguente indirizzo

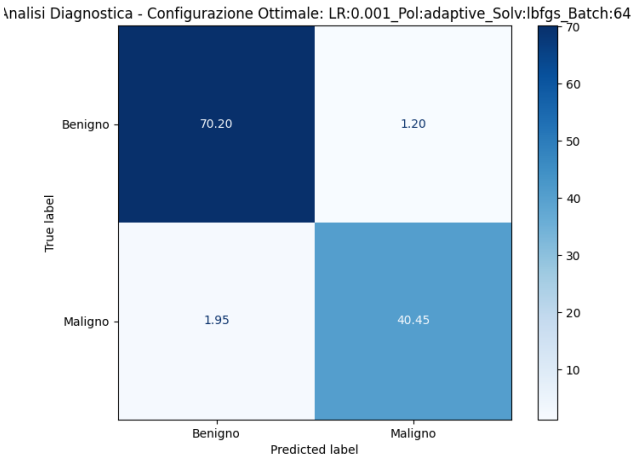
Quest'analisi è significativa in quanto, utilizzando questo dataset, il batch size sposta drasticamente il numero di aggiornamenti dei pesi per epoca:

- Batch 16 :≈ 28 aggiornamenti per epoca (molto rumore, ma molta "esplorazione").
- Batch 256 : 2 aggiornamenti per epoca (molto stabile, ma rischio di rimanere bloccati se il learning rate è basso)

Infatti, si osserva che aumentare la dimensione comporta una diminuzione del tempo medio, in quanto si fanno meno operazioni di aggiornamento. Inoltre, si osserva anche un un apparente aumento dell'accuracy.
Ciononostante per un'applicazione reale i valori 64 e 128 rappresentano i compressi migliori. Sebbene Batch 256 sia il "vincitore" in termini di accuracy, in alcuni casi potrebbe mostrare una leggerissima flessione nell'accuratezza (o una varianza maggiore) perché, con solo 2 aggiornamenti per epoca, il modello ha meno opportunità di correggersi durante l'addestramento.

3.3.4 Capacità diagnostica

Anche in questo si mostra la matrice di confusione della configurazione più adatta per questo scenario:



Il grafico completo è disponibile al seguente indirizzo

3.3.5 Conclusioni-Scenario 2

Concludendo l'analisi per questo scenario, si riassumono i parametri ottimali trovati:

- Solver: LBFGS/Adam
- Learing rate iniziale: 0.001
- Batch Size = 64/128
- Policy: Adaptive

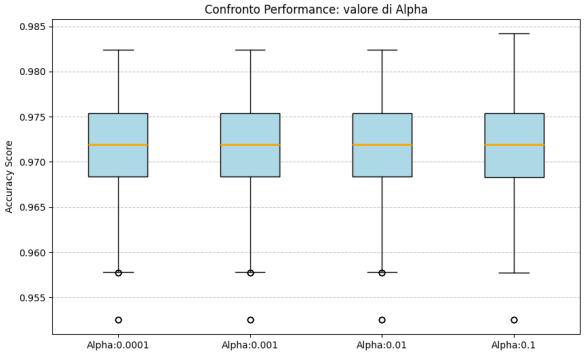
Per un'applicazione medicale, si consiglia LBFGS che garantisce una convergenza precisa e senza le oscillazioni tipiche dei metodi stocastici su piccoli campioni. Se si dovesse optare per un approccio neurale standard, ADAM con Batch Size 64 rappresenta la scelta più robusta, come dimostrato dalla tua analisi del "ginocchio" dei tempi e dalla stabilità dell'accuratezza.

3.4 Scenario 3

L'ultimo scenario di analisi si sofferma sull'impatto dei parametri di regolazione e l'efficacia dell'early stopping.

3.4.1 Analisi del parametro alpha

Anche in questo caso si è partiti da un'analisi macroscopica riguardante il parametro `alpha`, per capire l'effetto della forza della regolazione sulle performance. In particolare è stato generato il seguente boxplot, raggruppando tutti gli esperimenti per valore di `alpha`:



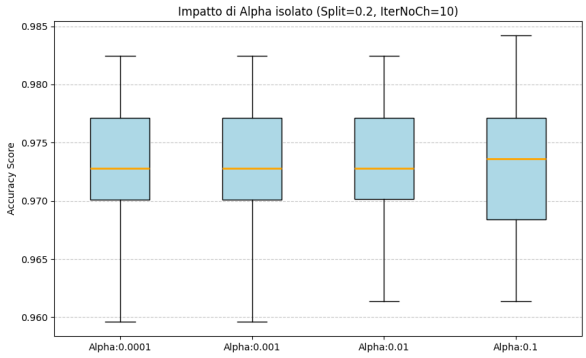
Il grafico completo è disponibile al seguente indirizzo

Da quest'analisi emerge che il parametro `alpha` non presenta un impatto significativo sulle performance del modello, e con tutti e quattro i valori testati si ottengono distribuzioni pressoché identiche (mediana ~ 0.972, range $0.958 - 0.982$).

Poiché non è emerso un chiaro vincitore, l'analisi microscopica (effettuata utilizzando il resto dei parametri fissi) è stata condotta su tutti e quattro i valori di alpha. In particolare si sono scelti i seguenti parametri fissi:

- Validation split pari a 0.2
- Numero di iterazioni senza cambiamenti pari a 10

ottenendo il seguente plot:

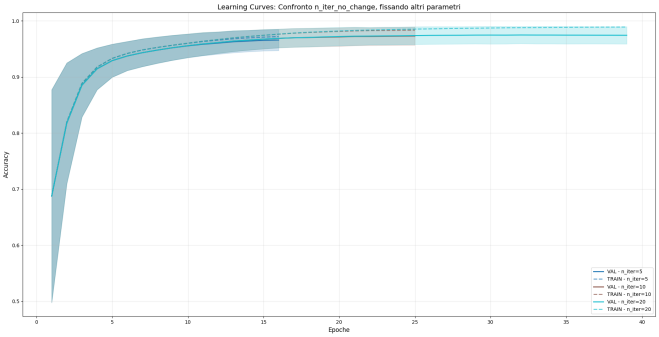


Il grafico completo è disponibile al seguente indirizzo

Si osserva quindi che anche eliminando eventuali influenze dovute alla variazione dei parametri `validation split` e `N_iter_no_change` il valore di `alpha` resta irrilevante.

3.4.2 Analisi sull'early stopping

Per analizzare l'impatto del parametro `n_iter_no_change` sulla convergenza e sul rischio di overfitting, sono state confrontate le curve di apprendimento (training vs validation) per i tre valori testati (5, 10, 20), fissando `alpha=0.001` e `validation_split=0.2` ottenendo il seguente grafico:



Il grafico completo è disponibile al seguente indirizzo

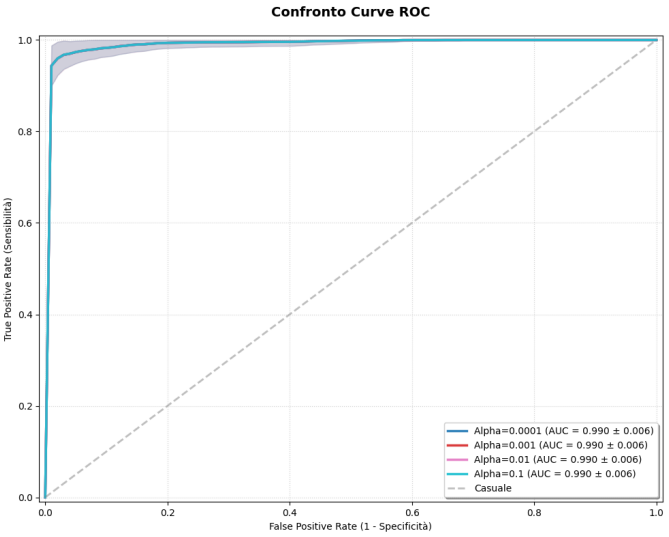
Si osserva che tutte e tre le configurazioni raggiungono lo 0.9 di accuracy entro le prime 10 epoche. L'andamento delle tre curve è molto simile e caratterizzato da una fase di apprendimento "veloce" iniziale. Ovviamente, utilizzando valori diversi per il parametro `n_iter_no_change`, l'algoritmo si fermerà dopo un numero di epoche diverse:

- Con `n_iter_no_change`= 5 intorno alla 16-esima epoca
- Con `n_iter_no_change`= 10 alla 25-esima epoca
- Con `n_iter_no_change`= 20 intorno alla 38-esima epoca

Per quanto riguarda l'overfitting, le curve train (tratteggiate) e validation (continue) sono praticamente sovrapposte e il modello generalizza bene, mentre per la stabilità si ha una banda di confidenza ampia durante le prime epoche, che si restringe e stabilizza dopo la 15-esima epoca.

Se ne deduce che il parametro `n_iter_no_change` pari a 5-10 rappresenta il miglior compromesso tra efficienza computazionale e performance, evitando iterazioni superflue senza sacrificare l'accuratezza.

In conclusione, a supporto delle analisi già fatte si riporta anche le tre curve ROC, variando il parametro `alpha`:

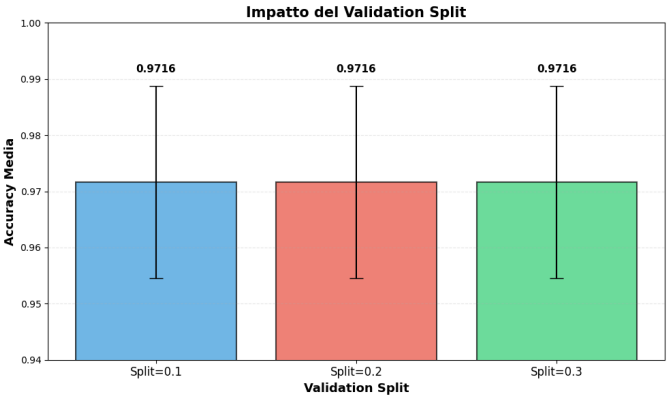


Il grafico completo è disponibile al seguente indirizzo

3.4.3 Analisi del validation split

L'ultima analisi riguarda l'impatto del parametro `validation_split`, in quanto rappresenta il trade-off tra quantità di dati utilizzati per il training effettivo e per la validation: più dati si utilizzeranno per la validazione e meno dati verranno utilizzati per il training. Ciò implica che, utilizzando un `validation_split` basso, si hanno più dati per il training e quindi potenzialmente è possibile ottenere un'accuracy leggermente più alta. Con un `validation_split` più basso si tenderà ad avere un'accuracy leggermente più bassa.

Si riporta di seguito il barplot ottenuto mediando tutti i risultati, raggruppando per il parametro in esame:



Il grafico completo è disponibile al seguente indirizzo

Dall'analisi si rileva una notevole robustezza del modello rispetto alla proporzione di dati destinati alla validazione. Le tre configurazioni testate (10%, 20%, 30%) producono risultati identici. Questo indica che:

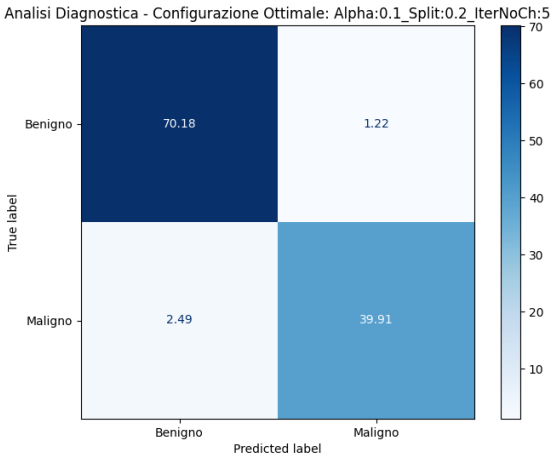
- Il dataset contiene dati sufficienti per il training anche con split più grandi (30%)
- Il trade-off tra quantità di dati per training e affidabilità della validation non è critico per questo prob-

lema

Questo risultato, combinato con la bassa sensibilità ad `alpha`, conferma che il dataset Wisconsin Breast Cancer non soffre di overfitting e che il modello generalizza efficacemente anche con configurazioni di regolarizzazione diverse.

3.4.4 Capacità diagnostica

Per concludere l'analisi dello scenario si riporta la matrice di confusione ottenuta per la configurazione più promettente:



Il grafico completo è disponibile al seguente indirizzo

3.4.5 Conclusioni-Scenario 3

Si conclude riassumendo:

- I parametri `alpha, validation_split` non hanno un impatto rilevante sulle prestazioni
- La presenza del meccanismo di early stopping è essenziale, in quanto ha permesso di ottenere risultati molto simili a quelli ottenuti per gli scenari 1-2 (nei quali l'early stopping non era presente) ma riducendo drasticamente il numero effettivo di epoche dell'algoritmo.
  In particolare, il parametro `n_iter_no_change` posto pari a 5-10 è sufficiente per rendere l'algoritmo efficiente senza andare a inficiare sulle prestazioni.

3.5 Conclusioni

Unendo i risultati e le considerazioni fatte sui vari scenari presi in esame, si può concludere che:

Parametro	Valore	Motivazione
Architettura	1 layer, 50 neuroni	Alta AUC, meno overfitting
Attivazione	ReLU	Veloce, stabile
Solver	LBFGS / Adam	Precisione vs robustezza
Learning rate	0.001	Convergenza stabile
Batch size	64–128	Efficienza/accuratezza
LR policy	Adaptive	Stabilità finale
Alpha (L2)	0.1	Riduce varianza
Early stopping	$n\_iter = 5 - 10$	Migliore generalizzazione
Validation split	0.2	Training/validation bilanciati

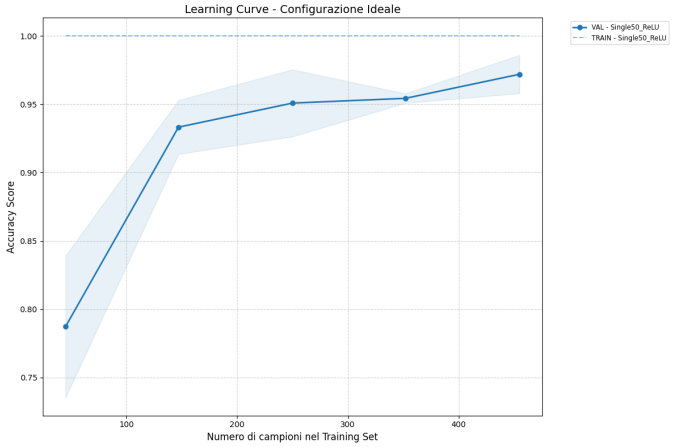
Ottenendo così:

- un modello clinicamente sicuro: pochi falsi negativi, minima varianza, AUC elevata.
- computazionalmente efficiente

3.5.1 Analisi dimensione dataset

Per concludere l'analisi, stabilita la migliore configurazione per la rete, si è condotto un esperimento con l'obiettivo di andare a studiare l'impatto della dimensione del dataset sull'accuracy, generando il seguente grafico (ottenuto con:

```
"Single50_ReLU": {
"params": {
"hidden_layer_sizes": (50,),
"activation": "relu",
"solver": "lbfgs",
"alpha": 0.1,
"learning_rate_init": 0.001, "max_iter": 200,
"early_stopping": True, "n_iter_no_change": 5,
"validation_fraction": 0.2, "random_state": 42
}
} )
```



Il grafico completo è disponibile al seguente indirizzo

Da questo grafico si deduce che il modello migliora all'aumentare della dimensione del dataset. In particolare otteniamo una accuracy sul training costante intorno al 1.0, ma una accuracy sul validation in continuo aumento, indicando quindi che il modello sta imparando schemi reali e non overfittando. Inoltre, quest'ultima non si appiattisce negli ultimi punti, ma la pendenza è ancora positiva: aggiungendo ancora più dati (se fosse possibile) le prestazioni potrebbero migliorare ulteriormente.

In conclusione, il grafico mostra che con circa 300-350 campioni, il modello raggiunge un plateau di accuratezza stabile (sopra il 96-97%). In un contesto clinico, questo suggerisce che la dimensione del dataset attuale (569 istanze) è adeguata per addestrare un MLP affidabile, poiché l'aggiunta di ulteriori dati produce benefici marginali decrescenti.