



UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Corso di Fondamenti di Intelligenza Artificiale

FlagFinder

Agente Razionale per la risoluzione del gioco del Campo Minato
<https://github.com/ljcia4/FlagFinder>

Professore:
Ch.mo Prof. Fabio Palomba

Studenti:
Chierchia Michele 0512120449
Riccio Felicia 0512120452
Scarallo Gloria 0512119236

ANNO ACCADEMICO 2025/2026

Indice

1	Introduzione	3
1.1	Sistema Attuale	3
1.2	Obiettivo	3
1.3	Analisi del problema	3
1.4	Tecnologie utilizzate	4
1.5	Specifiche PEAS	4
1.6	Caratteristiche dell'ambiente	5
2	Analisi dell'Algoritmo	6
2.1	Strategia Generale	6
2.2	Fase 1: Logica Deterministica (Single-Cell)	6
2.3	Fase 2: Logica Avanzata (Insiemistica)	6
2.3.1	Applicazione pratica: Il Pattern 1-2-1	7
2.4	Fase 3: Gestione dell'Incertezza (Guessing)	7
3	Dettagli Implementativi	8
3.1	Struttura della Classe AI	8
3.2	Configurazione dei Modelli	8
3.2.1	XGBoost (eXtreme Gradient Boosting)	8
3.2.2	MLP (Multi-Layer Perceptron)	8
3.3	Generazione Post-Factum	9
3.4	Ciclo di Decisione (Game Loop)	9
3.5	Raccolta dei Dati e Dataset	10
3.6	Feature Engineering	11
3.7	Validazione	12
4	Test e Risultati	13
4.1	Metodologia di Test	13
4.2	Configurazioni di Test	13
4.3	Risultati Ottenuti	13
4.4	Analisi dei Trade-offs tra i Due Algoritmi	14
4.4.1	Paradigma di Apprendimento: Batch vs Online	14
4.4.2	Il problema dell' <i>Explainability</i>	15
4.4.3	Architettura e Gestione delle Feature	15
4.4.4	Performance	16
5	Possibili Miglioramenti	17
5.1	Calcolo Probabilistico Globale	17
5.1.1	Formulazione Matematica	17
5.2	End-Game Solver	17
6	Soluzioni non implementate	19
6.1	Raccolta dei dati	19
6.2	Scelta delle feature	19
6.3	Classificatori	19

7 Conclusione	20
----------------------	-----------

1 Introduzione

1.1 Sistema Attuale

Il gioco del Campo Minato (Minesweeper) è un classico rompicapo logico per computer. L’obiettivo è ripulire un campo rettangolare pieno di mine nascoste, senza farne esplodere nessuna. Ad ogni turno il giocatore può decidere di scoprire una cella o posizionare una bandiera, per indicare la presenza di una mina: cliccando su una cella vuota, può apparire un numero (da 1 a 8), che indica quante mine sono presenti nelle caselle adiacenti (la griglia intorno 3×3).

Attualmente, la risoluzione del gioco è affidata interamente all’intuizione e alla capacità di calcolo dell’utente umano. Questo approccio presenta diversi limiti: la fatica cognitiva nel calcolare le probabilità su griglie di grandi dimensioni, la propensione all’errore umano dovuto a distrazioni e la difficoltà nel gestire situazioni di incertezza probabilistica senza strumenti di supporto.

1.2 Obiettivo

Lo scopo del progetto è sviluppare un modulo di Intelligenza Artificiale, nello specifico un **Agente Razionale**, in grado di interagire autonomamente con l’ambiente di gioco del Campo Minato . L’agente deve essere capace di percepire lo stato della griglia, applicare regole di inferenza logica per determinare le mosse sicure e gestire situazioni di incertezza minimizzando il rischio di sconfitta. L’obiettivo finale è massimizzare la percentuale di vittorie (*win rate*) e anche la velocità di risoluzione del campo minato in tempo reale.

1.3 Analisi del problema

Il problema della risoluzione del Campo Minato può essere formalizzato come un **Problema di Soddisfacimento di Vincoli (CSP)**. Ogni cella numerata impone un vincolo sul numero di mine presenti nelle celle adiacenti coperte.

- **Stati:** Ogni possibile combinazione in una griglia $N \times M$ di bandiere piazzate (fino ad un massimo di K bandiere) e di caselle scoperte.
- **Stato iniziale:** Una griglia $N \times M$ in cui tutte le celle sono coperte, e sono presenti K mine in celle casuali.
- **Azioni:** Cliccare una cella o posizionare una bandiera in una cella che indica la presenza di una potenziale mina.
- **Modello di transizione:** Restituisce la griglia con una o più celle scoperte, o in cui è stata posizionata una bandiera.
- **Test obiettivo:** Tutte le celle senza mine sono state scoperte, oppure una delle mine è stata scoperta.

Approccio Logico vs Probabilistico: Si è scelto di implementare un agente che privilegia la deduzione logica stretta (determinismo) e ricorre a euristiche probabilistiche (es. scelta sulla frontiera) solo quando strettamente necessario.

1.4 Tecnologie utilizzate

Le tecnologie utilizzate per lo sviluppo sono:

- **Python 3.x:** Linguaggio di programmazione principale.
- **Tkinter:** Libreria standard per la creazione dell’interfaccia grafica (GUI) e la simulazione dell’ambiente di gioco. Nel contesto del progetto l’abbiamo utilizzata per ricreare in locale il gioco originale (il *campo minato*).
- **Random:** Modulo utilizzato per la gestione della stocasticità (posizionamento mine e guessing).
- **Pandas:** Libreria fondamentale per la manipolazione e l’analisi dei dati strutturati. Nel progetto funge da ”ponte” tra i dati grezzi salvati durante le partite e gli algoritmi di intelligenza artificiale, e viene utilizzata per il caricamento del database e nella fase di *data preparation*.
- **Scikit-learn:** Libreria standard di Python per il Machine Learning classico, che funge da ”base” fondamentale per l’intero progetto. Contiene gli algoritmi e le strutture dati necessarie per fare previsioni, oltre che i metodi necessari per il calcolo delle metriche di qualità del modello. È stata utilizzata nelle fasi di *pre-processing* e *normalizzazione* delle feauture, nell’implementazione dell’ *MLPClassifier* e l’applicazione delle *metriche di valutazione* delle performance.
- **NumPy:** Libreria fondamentale per il calcolo scientifico in Python, che fornisce supporto per array multidimensionali e funzioni matematiche ad alte prestazioni. Nel progetto svolge il ruolo di ”motore matematico”, ed è stata utilizzata per la manipolazione delle liste di feature del gioco e l’esecuzione di operazioni sulle probabilità e sul dataset.
- **XGBoost (eXtreme Gradient Boosting):** Una libreria di Machine Learning basata su alberi decisionali ottimizzati (Gradient Boosting), scelta per la sua straordinaria efficienza e velocità sui dati strutturati (tabulari). Nel progetto è stata impiegata come ”cervello” alternativo alla rete neurale per l’addestramento offline.

1.5 Specifica PEAS

Un ambiente è un’istanza di un problema di cui gli agenti razionali rappresentano le soluzioni, in questo caso l’ambiente è il campo minato.

Un ambiente viene generalmente descritto tramite la formulazione PEAS, ovvero Performance, Environment, Actuators, Sensors.

- P.** Misura di prestazione adottata per valutare l’operato di un agente.
- E.** Descrizione degli elementi che formano l’ambiente.
- A.** Gli attuatori disponibili all’agente per intraprendere le azioni.
- S.** I sensori attraverso i quali riceve gli input percettivi.

Di seguito è riportata la specifica PEAS del progetto.

- **Performance (P):** Completamento del gioco senza far esplodere mine (Win Rate).
- **Environment (E):** Griglia $N \times M$ con K mine. Regole di adiacenza a 8 vicini.
- **Actuators (A):** Click sinistro (svelare), Click destro (bandiera).
- **Sensors (S):** Input visivo numerico (0-8) dalle celle svelate e delle bandiere piazzate.

1.6 Caratteristiche dell'ambiente

- **Parzialmente osservabile:** L'agente non conosce la posizione delle mine.
- **Stocastico:** La disposizione delle mine è casuale.
- **Sequenziale:** Le decisioni passate influenzano le informazioni future.
- **Statico:** L'ambiente non cambia spontaneamente.
- **Discreto:** Stati e azioni discreti.
- **Singolo-Agento:** L'agente opera da solo.

2 Analisi dell’Algoritmo

2.1 Strategia Generale

L’agente implementato segue una gerarchia di strategie per determinare la mossa successiva. L’approccio è *iterativo*: l’agente tenta di applicare la logica più semplice e computazionalmente leggera. Se questa non produce mosse, scala verso logiche più complesse, fino ad arrivare, come ultima risorsa, al tentativo probabilistico (“guessing”).

2.2 Fase 1: Logica Deterministica (Single-Cell)

Il primo livello di ragionamento analizza ogni singola cella svelata e i suoi vicini immediati. Vengono applicate due regole fondamentali di inferenza:

Regola della Certezza (Flagging): Se una cella ha valore N e il numero di celle adiacenti coperte (più quelle già bandierate) è esattamente uguale a N , allora tutte le celle coperte adiacenti sono necessariamente mine.

$$\text{Valore}_{cella} = \text{Count}_{vicini_coperti} + \text{Count}_{vicini_bandierati} \implies \forall c \in \text{Vicini}_{coperti}, c = \text{Mina}$$

Regola della Sicurezza (Revealing) Se una cella ha valore N e il numero di celle adiacenti già bandierate è uguale a N , allora il vincolo della cella è soddisfatto. Tutte le restanti celle coperte adiacenti sono sicure.

$$\text{Valore}_{cella} = \text{Count}_{vicini_bandierati} \implies \forall c \in \text{Vicini}_{coperti}, c = \text{Sicura}$$

2.3 Fase 2: Logica Avanzata (Insiemistica)

Quando la logica a singola cella non produce nuove mosse, l’agente applica regole di inferenza basate sulla teoria degli insiemi, confrontando i vicini di due celle adiacenti A e B . Se l’insieme dei vicini coperti di A è un sottoinsieme dei vicini coperti di B :

$$\text{Vicini}(A) \subseteq \text{Vicini}(B)$$

possiamo dedurre informazioni sulla differenza:

$$D = \text{Vicini}(B) \setminus \text{Vicini}(A)$$

A questo punto si applicano due regole fondamentali:

- **Regola della Zona Sicura:** Se il numero di mine rimanenti per A è uguale al numero di mine rimanenti per B , allora tutte le celle in D sono **sicure**.
- **Regola della Saturazione:** Se la differenza di mine richieste tra B e A è uguale alla cardinalità di D , allora tutte le celle in D sono **mine**.

Il principio matematico alla base di queste regole è che le celle condivise (l’intersezione $\text{Vicini}(A)$) agiscono da vincolo comune. Poiché tutte le mine che soddisfano A devono trovarsi nell’intersezione, esse contribuiscono automaticamente anche al conteggio di B .

- Nel primo caso ($\text{Mine}_A = \text{Mine}_B$), B non necessita di mine aggiuntive rispetto ad A , rendendo vuota (sicura) la zona esclusiva D .

- Nel secondo caso ($Mine_B - Mine_A = |D|$), le mine aggiuntive richieste da B devono trovarsi necessariamente nell'unica zona che A non vede, ovvero D . Poiché lo spazio disponibile in D è esattamente pari alle mine mancanti, ogni cella di D deve essere minata.

2.3.1 Applicazione pratica: Il Pattern 1-2-1

Un esempio classico di applicazione della seconda regola (Saturazione) è il pattern **1-2-1** lungo un bordo. Immaginiamo tre celle adiacenti con valori **1**, **2**, **1** affacciate su tre celle coperte x, y, z .

Analizzando la coppia **1** (cella A) e **2** (cella B):

1. I vicini di A (il valore 1) sono un sottoinsieme dei vicini di B (il valore 2).
2. La differenza insiemistica D contiene solo la cella z .
3. La differenza di mine richieste è $2 - 1 = 1$.
4. La cardinalità di D è 1.

Poiché la differenza di mine (1) eguaglia la cardinalità (1), per la regola descritta sopra, la cella z è sicuramente una mina. Applicando la stessa logica specularmente tra l'altro 1 e il 2 centrale, si deduce che anche la cella x è una mina. Di conseguenza, avendo il 2 centrale già le sue due mine (x e z), la cella centrale y è sicuramente sicura.

2.4 Fase 3: Gestione dell'Incertezza (Guessing)

In assenza di mosse logiche certe, l'agente deve indovinare. Per massimizzare la razionalità anche in questa fase stocastica, l'agente non sceglie a caso su tutta la griglia, ma seleziona una cella dalla **Frontiera**. La frontiera è definita come l'insieme di celle coperte adiacenti a celle svelate. Svelare una cella sulla frontiera ha un'alta probabilità di sbloccare nuove deduzioni logiche ("Information Gathering"), a differenza di una cella isolata.

3 Dettagli Implementativi

Il progetto è sviluppato in Python e si avvale della libreria `Tkinter` per la simulazione dell'ambiente. Il cuore del sistema è la classe `MinesweeperAI`.

3.1 Struttura della Classe AI

La classe mantiene un riferimento all'istanza del gioco (`game_instance`) per accedere allo stato della griglia (lettura sensori) e per eseguire le mosse (attuatori). Gestisce inoltre il caricamento e l'uso dei modelli ML tramite `joblib`.

3.2 Configurazione dei Modelli

Le scelte architetturali e i parametri dei modelli sono stati selezionati per bilanciare accuratezza e capacità di generalizzazione.

3.2.1 XGBoost (eXtreme Gradient Boosting)

XGBoost (eXtreme Gradient Boosting) è un algoritmo di machine learning supervisionato, basato su ensemble di alberi decisionali. Utilizza la tecnica del gradient boosting, costruendo modelli in sequenza dove ogni nuovo albero corregge gli errori dei precedenti. Utilizzato per l'addestramento offline sul dataset completo.

- **`n_estimators = 200`**: Numero di alberi decisionali sequenziali. Un numero elevato permette di correggere errori residui.
- **`learning_rate = 0.05`**: Un tasso di apprendimento conservativo per garantire una convergenza stabile senza oscillazioni.
- **`max_depth = 6`**: La profondità massima degli alberi è limitata a 6 per prevenire l'overfitting su pattern troppo specifici.
- **`subsample = 0.8` e `colsample_bytree = 0.8`**: Tecniche di regolarizzazione stocastica. Ogni albero utilizza solo l'80% dei dati e delle feature, aumentando la robustezza del modello.
- **`eval_metric = 'logloss'`**: La funzione di perdita ottimizzata è la Binary Cross-Entropy.

3.2.2 MLP (Multi-Layer Perceptron)

Il MLP (Multi-Layer Perceptron) è una rete neurale artificiale composta da più strati di neuroni interconnessi, utilizzata per risolvere problemi di classificazione e regressione complessi tramite l'apprendimento di pattern non lineari.

Utilizzato per l'apprendimento online grazie alla sua leggerezza e flessibilità.

- **Pipeline di Preprocessing:** I dati passano attraverso uno `StandardScaler`. Normalizzare le feature (media 0, varianza 1) è fondamentale per le reti neurali per evitare che feature con scale diverse influenzino i gradienti in modo sproporzionato.
- **Architettura:**

- *Input Layer*: **25** nodi (24 feature locali + 1 densità).
- *Hidden Layers*: Due strati densi rispettivamente di **64** e **32** neuroni.
- *Funzione di Attivazione*: **ReLU** (Rectified Linear Unit) per gli strati nascosti, scelta per l’efficienza computazionale.
- **Ottimizzatore**: **Adam** con learning rate adattivo (‘adaptive’), per gestire in modo efficiente la discesa del gradiente.
- **Warm Start**: Impostato a **True**, permette al modello di mantenere i pesi appresi tra una partita e l’altra, continuando l’addestramento (Incremental Learning).

3.3 Generazione Post-Factum

Per garantire che la prima mossa dell’agente sia sempre sicura e permetta l’inizio del ragionamento logico, è stata implementata una logica di generazione *post-factum* (o lazy generation). Le mine non vengono piazzate all’inizializzazione della classe, ma solo dopo il primo click dell’agente. Il primo click (tipicamente al centro della griglia, $(rows//2, cols//2)$) e i suoi vicini vengono esclusi dal set delle possibili posizioni delle mine, garantendo un’area di partenza libera.

3.4 Ciclo di Decisione (Game Loop)

Il metodo **step** è il nucleo dell’agente. A differenza di un approccio puramente sequenziale, l’agente dà precedenza assoluta alla certezza logica rispetto alla stima probabilistica. Il flusso di esecuzione segue un ordine rigoroso in cui ogni fase viene eseguita solo se la precedente non ha prodotto alcuna azione valida.

1. **Acquisizione Stato**: L’agente legge lo stato attuale di tutte le celle visibili, e verifica se la partita è conclusa esplosione di una mina (`game_over = True`) o se il numero di celle rivelate corrisponde al totale delle celle sicure (`victory = True`).
2. **Inferenza Logica**: Applica le regole base e avanzate (Fase 1 e 2). Non appena viene trovata ed eseguita una mossa sicura, la funzione esegue un `return True` immediato. Questo “Early Exit” è fondamentale: garantisce che l’agente rivaluti sempre lo stato aggiornato della griglia prima di procedere a ragionamenti più complessi.
3. **Esecuzione**: Se viene trovata una mossa sicura, viene eseguita immediatamente e il loop ricomincia.
4. **ML Fallback**: Se nessuna mossa sicura è trovata, viene invocato il metodo `make_guess_with_ml` per la predizione probabilistica, che:
 - (a) Isola la frontiera delle celle esplorabili.
 - (b) Interroga il modello predittivo (XGBoost o MLP) per ottenere le probabilità di sicurezza.
 - (c) Esegue la mossa con la probabilità di successo più alta (`argmax`).
5. **Apprendimento (Solo MLP)**: Al termine della partita, viene invocato `learn_online` che esegue `partial_fit` sui nuovi dati raccolti.

(a) **Aggiornamento dello Scaler:** Viene invocato `scaler.partial_fit(X)`. Questo permette allo `StandardScaler` di aggiornare le statistiche globali (media e varianza) dei dati in modo incrementale, adattando la normalizzazione man mano che l'agente esplora nuove configurazioni di gioco.

(b) **Aggiornamento della Rete Neurale (Backpropagation):** Dopo aver normalizzato i nuovi dati (X_{scaled}), viene chiamato:

```
mlp.partial_fit(X_scaled, y, classes = [0, 1])
```

Questo comando esegue un ciclo di *Stochastic Gradient Descent* solo sui nuovi dati. I pesi della rete vengono aggiustati leggermente per ridurre l'errore commesso nelle ultime mosse, consolidando l'esperienza appena acquisita senza dimenticare quella passata.

(c) **Persistenza (Salvataggio):** Infine, il modello aggiornato viene serializzato su disco tramite `joblib.dump`. Questo garantisce che l'agente mantenga la sua "intelligenza" anche chiudendo e riaprendo l'applicazione.

3.5 Raccolta dei Dati e Dataset

Per addestrare i modelli predittivi, è stato necessario costruire un dataset significativo. I dati vengono raccolti in due modalità:

- **Raccolta Offline (Storico):** Metodo di raccolta dati per la costruzione di un database utilizzato per l'addestramento dell'agente **XGBoost**, e il pre-training della rete **MLP**. Le partite simulate vengono salvate in un file CSV (`minesweeper_dataset.csv`). I dati vengono salvati tramite la funzione `_record_context`, esclusivamente quando le euristiche logiche non riescono a determinare una mossa sicura, costringendo l'agente a effettuare un tentativo probabilistico (*guessing*).

Data preparation: Le operazioni principali di data preparation del dataset eseguite sono:

- **Rimozione dei duplicati:** Viene invocata la funzione `df.drop_duplicates()`. Dato che molte configurazioni del campo minato sono ricorrenti (es. angoli o spazi vuoti), rimuovere i duplicati è essenziale per evitare che il modello vada in *overfitting* memorizzando pattern e configurazioni frequenti invece di generalizzare e apprendere la logica.
- **Splitting:** Separazione tra le variabili di input (la griglia e la densità) e la variabile target (la colonna '`'safe'`') tramite il comando `df.drop('safe', axis=1)`. Il dataset viene diviso in matrice delle feature (X) e vettore target (y). Successivamente, viene applicata la funzione `train_test_split` per separare i dati in:
 - * **Training Set (80%):** Usato per istruire l'algoritmo.
 - * **Test Set (20%):** Usato esclusivamente per la valutazione finale, garantendo che le metriche riflettano le prestazioni su dati mai visti prima.

- **Raccolta Online (Real-time):** Metodo di raccolta dati utilizzato per l'addestramento della rete neurale **MLP**. All'avvio, il sistema verifica l'esistenza di un "cervello" preesistente (`minesweeper_brain_online.pkl`).

- Se il file esiste, il modello viene caricato tramite `joblib`, permettendo all’agente di recuperare l’esperienza delle sessioni precedenti.
- Se il file non esiste, viene istanziato un nuovo modello. In questa fase, se è presente il dataset storico `minesweeper_dataset.csv`, viene eseguito un **pre-training completo** (`_full_pre_train`). Questo passaggio è fondamentale per evitare che l’agente inizi la sua esplorazione in modo totalmente casuale, fornendogli una conoscenza di base derivata dalle partite passate.

Durante il gioco, quando le euristiche logiche falliscono e l’agente esegue un *guess*, la coppia (feature, esito) viene salvata in un buffer temporaneo (`self.memory`). Al termine della partita (Game Over o Vittoria), viene invocato il metodo `learn_online`. A differenza dell’addestramento offline, questo metodo utilizza la funzione `partial_fit`, che aggiorna i pesi della rete neurale solo sui nuovi dati raccolti, senza resettare la conoscenza precedente. Questo approccio permette al modello di adattarsi continuamente a nuove situazioni senza dover rielaborare l’intero storico dei dati.

3.6 Feature Engineering

Per ogni cella candidata sulla frontiera, viene estratto un vettore di feature strutturato:

- **Finestra Locale:** Una griglia 5×5 centrata sulla cella candidata. Le celle sono codificate numericamente: -2 per i bordi e -1 per le celle coperte. Per le celle già svelate, invece, viene salvato il *valore effettivo*, calcolato, tramite la funzione `_get_effective_value`, come la differenza tra il numero mostrato sulla cella e il numero di bandiere adiacenti già posizionate.

$$\text{Mine}_{\text{residue}} = \text{Mine}_{\text{adiacenti}} - \text{Bandiere}_{\text{rivelate}}$$

- **Densità Globale:** Una feature aggiuntiva (‘`global_density`’) calcolata come il rapporto tra le mine rimanenti stimate

$$\text{Mine}_{\text{residue}} = \text{Mine}_{\text{totali}} - \text{Bandiere}_{\text{piazzate}}$$

e il numero totale di celle coperte

$$\text{Celle}_{\text{nascoste}} = \text{Celle}_{\text{totali}} - \text{Celle}_{\text{rivelate}}$$

Questo aiuta il modello a stimare il rischio di base della partita corrente.

$$\text{Densità}_{\text{globale}} = \frac{\text{Mine}_{\text{residue}}}{\text{Celle}_{\text{nascoste}}}$$

- **Target:** La colonna finale `safe` rappresenta l’etichetta (label) per l’addestramento supervisionato. Indica se il tentativo casuale effettuato dal modello sulla cella (r, c) ha avuto successo o meno:

$$y = \begin{cases} 1 & \text{se la cella rivelata era SICURA,} \\ 0 & \text{se la cella rivelata era una MINA.} \end{cases}$$

3.7 Validazione

Dopo l'addestramento (`fit`), il modello (sia XGBoost che MLP) viene valutato sul Test Set. Vengono calcolate delle metriche di qualità del modello, che permettono di misurare il grado di affidabilità del modello. Sulla base dei valori della matrice di confusione, che mostra il numero di predizioni corrette ed errate fatte dal modello, possiamo poi calcolare diverse metriche.

	Celle predette come Safe	Celle predette come Mine
Reale: Safe	True Positive (TP)	False Negative (FN)
Reale: Mina	False Positive (FP)	True Negative (TN)

Tabella 1: Matrice di Confusione.

Le misure di qualità principali sono:

- **Accuracy:** Misura la percentuale totale di predizioni corrette (sia mine che celle sicure) rispetto al totale dei casi esaminati. Sebbene utile per una visione d'insieme, nel gioco del Campo Minato non è sufficiente da sola a garantire la sopravvivenza.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision (Safe):** Indica l'affidabilità del modello quando predice che una cella è "sicura" (classe 1). Un valore basso di precisione implicherebbe un alto rischio di cliccare su una mina, portando al Game Over istantaneo. L'obiettivo primario dell'IA è massimizzare questa metrica.

$$\text{Precision} = \frac{\text{Veri Positivi}}{\text{Veri Positivi} + \text{Falsi Positivi}}$$

- **Recall:** Misura la capacità del modello di individuare tutte le celle sicure presenti. Un valore basso di Recall implica un alto tasso di Falsi Negativi (*FN*), ovvero celle sicure che l'IA classifica erroneamente come pericolose. In questo scenario, l'agente, pur non cliccando sulle mine, si rifiuta di scoprire celle sicure, portando la partita a una situazione di *stallo (deadlock)* in cui non vengono generate nuove mosse per mancanza di coraggio. L'obiettivo primario dell'IA è massimizzare questa metrica.

$$\text{Recall} = \frac{\text{Veri Positivi}}{\text{Veri Positivi} + \text{Falsi Negativi}}$$

- **F1-Score:** Utile per verificare se il modello è bilanciato o se tende a ignorare troppo spesso le mine per paura di sbagliare.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

4 Test e Risultati

4.1 Metodologia di Test

Poiché l'ambiente è stocastico (la posizione delle mine cambia ad ogni partita), la valutazione dell'agente si basa su una media statistica. Sono state eseguite 1000 simulazioni per ciascuna delle configurazioni standard di difficoltà.

4.2 Configurazioni di Test

- **Principiante:** Griglia 9×9 , 10 Mine. Densità mine: 12.3%.
- **Intermedio:** Griglia 16×16 , 40 Mine. Densità mine: 15.6%.
- **Esperto:** Griglia 16×30 , 99 Mine. Densità mine: 20.6%.

4.3 Risultati Ottenuti

I risultati mostrano come la percentuale di successo (*Win Rate*) diminuisca all'aumentare della densità delle mine, a causa della maggiore probabilità di dover ricorrere al *guessing*.

Difficoltà	Partite	Vittorie	Win Rate
Principiante	1000	902	90,2%
Intermedio	1000	756	75,6%
Esperto	1000	175	17,5%

Tabella 2: Risultati sperimentali dell'agente base

Analisi degli Errori La maggior parte delle sconfitte nella modalità Esperto non è dovuta a errori logici dell'agente, ma a situazioni di **ambiguità irrisolvibile** (es. blocchi 50/50 finali) dove la logica non può determinare la posizione della mina e l'agente è costretto a tirare a indovinare.

Difficoltà	Partite	Vittorie	Win Rate
Principiante	1000	979	97,9%
Intermedio	1000	892	89,2%
Esperto	1000	457	45,7%

Tabella 3: Risultati sperimentali dell'agente XGBoost

Analisi degli Errori Nonostante nelle varie modalità di gioco (specialmente Esperto) ci sia un netto miglioramento delle percentuali di vittoria, l'algoritmo si basa sulle probabilità estrapolate dai pattern scoperti tramite i dati di addestramento, ma ci sarà sempre un

certo grado di **imprevedibilità** che non ha basi statistiche, a causa della casualità del gioco causata dal posizionamento delle mine.

Difficoltà	Partite	Vittorie	Win Rate
Principiante	1000	968	96,8%
Intermedio	1000	897	89,7%
Esperto	1000	450	45%

Tabella 4: Risultati sperimentali della rete neurale MLP

Analisi degli Errori L'algoritmo ottiene risultati molto simili al XGBoost, in quanto viene limitato dagli stessi problemi dell'albero di classificazione, legati alla **casualità**.

4.4 Analisi dei Trade-offs tra i Due Algoritmi

L'analisi comparativa tra l'agente basato su **XGBoost** e l'agente basato su **MLPClassifier** evidenzia differenze strutturali significative che influenzano le prestazioni, l'adattabilità e i costi computazionali.

4.4.1 Paradigma di Apprendimento: Batch vs Online

La distinzione fondamentale risiede nella modalità di *aggiornamento* dei pesi e di *training* del modello:

- **Agente ML (XGBoost) - Apprendimento Statico:** Utilizza un approccio *Offline Batch Learning*. Il modello viene addestrato su un dataset storico completo (`minesweeper_dataset.csv`) prima dell'esecuzione, che viene generato tramite le mosse di `guessing` eseguite su 1000 partite per ogni difficoltà dal `solver.py`.

– Vantaggi:

- * Stabilità immediata: Dato che XGBoost eccelle su dati tabulari, l'agente offre prestazioni massime fin dalla prima partita, assumendo che il dataset di training sia rappresentativo.
- * Controllo del dataset: Avendo il dataset fisico, è possibile gestire i dati su cui il modello apprende, eseguendo operazioni di *data preparation* (data cleaning, feature scaling, feature selection, data balancing).

– Svantaggi:

- * Rigidità: Il modello è "congelato" (*frozen*). Non può correggere i propri bias o adattarsi a nuove strategie durante l'esecuzione senza un riaddestramento manuale completo.

- **Agente MLP - Apprendimento Incrementale:** Implementa un approccio *Online Learning* sfruttando il metodo `partial_fit` di Scikit-Learn.

– Vantaggi:

- * Adattabilità: L'agente apprende dai propri errori in tempo reale, aggiornando i pesi dei nodi al termine di ogni partita. Questo permette al sistema di evolvere autonomamente senza dataset esterni iniziali.
- **Svantaggi:**
 - * Problema del *Cold Start*: L'agente inizia con pesi casuali, comportandosi in modo puramente stocastico nelle prime fasi, e per questo richiede una fase di pre-training. Richiede inoltre un numero significativo di iterazioni, e di conseguenza ha bisogno di "giocare" un certo numero di partite, per convergere a prestazioni accettabili.
 - * Sensibilità agli outlier: L'agente apprende, alla fine di ogni partita, dei pattern sulla base delle mosse eseguite, ma se si trova di fronte a degli *outlier*, i pesi assegnati ai suoi neuroni possono cambiare anche drasticamente, portando ad un comportamento instabile.

4.4.2 Il problema dell'*Explainability*

Uno dei difetti più grandi della rete **MLP** è l'*opacità*. Infatti, nonostante sia un modello molto veloce e con risultati simili al suo corrispettivo basato su albero di classificazione, esso manca di *Explainability*.

L'*Explainability* rappresenta la capacità di spiegare il comportamento di un modello e dedurre il ragionamento logico dietro ogni possibile risultato prodotto dall'algoritmo.

La rete MLP ha un'Explainability nulla, infatti, se volessimo capire quali pattern la rete ha dedotto oppure andare a ritroso per comprendere perché ha scelto una determinata mossa, non potremmo, poiché l'unico dato che otteniamo sono i pesi delle connessioni tra i vari neuroni che la compongono. Questo lo rende il modello una "*black-box*", di cui è possibile osservare solo il comportamento esterno, "*cosa*" ha prodotto, ma non è possibile comprendere "*come*" è arrivato ai risultati, e quindi non adatto a questo scopo.

L'albero XGBoost invece si presta molto bene a questo tipo di attività: per quanto l'albero possa essere complesso e diramato resta comunque *leggibile*, quindi se volessimo studiare il gioco e i suoi pattern esso è decisamente più adatto.

4.4.3 Architettura e Gestione delle Feature

Entrambi gli algoritmi elaborano una griglia locale 5×5 e una feature globale di densità, ma differiscono nell'efficacia di elaborazione:

- **Gestione Dati Tabulari:** **XGBoost** (basato su alberi decisionali) gestisce naturalmente le relazioni non lineari e le discontinuità nei dati grezzi. L'**MLP**, essendo una rete neurale *Fully Connected*, richiede la normalizzazione dei dati in ingresso; il codice implementa una **Pipeline** con **StandardScaler** per mitigare questo problema, mantenendo lo stato dello scalatore durante il learning online.
- **Complessità del Modello:** Lo script di training XGBoost imposta vincoli conservativi (`max_depth=6, subsample=0.8`) per prevenire l'overfitting. L'**MLP** utilizza un'architettura leggera (64, 32), necessaria per mantenere bassi i tempi di latenza durante il training in-game, ma potenzialmente meno capace di catturare pattern spaziali complessi rispetto a un approccio convoluzionale (CNN).

4.4.4 Performance

I trade-offs visti pocansi sono solo una linea guida generale per i due modelli. Infatti, nel nostro caso con quegli esatti iperparametri, il modello **XGBoost** raggiunge dei risultati simili e a volte superiori alla rete MLP. Ciò si può dedurre confrontando i *win rate* raggiunti in media dai due algoritmi per ogni difficoltà del gioco:

- **Difficoltà Principiante:** MLP: 96,8% vs XGBoost: 97,8%.
- **Difficoltà Intermedio:** MLP: 89,7% vs XGBoost: 89,2%.
- **Difficoltà Esperto:** MLP: 45% vs XGBoost: 45,7%.

Tuttavia **XGBoost** è meno performante rispetto a **MLP**, e impiega circa il doppio del tempo per eseguire le partite in ogni difficoltà del gioco. Confrontando le prestazioni dei due algoritmi nell'eseguire 1000 partite per ogni difficoltà del gioco, osserviamo:

- **Difficoltà Principiante:** MLP: 1.2 secondi vs XGBoost: 2.7 secondi
- **Difficoltà Intermedio:** MLP: 7.1 secondi vs XGBoost: 14.4 secondi
- **Difficoltà Esperto:** MLP: 37.3 secondi vs XGBoost: 1 minuto e 24 secondi.

Nonostante ciò, XGBoost riesce comunque a prendere scelte in tempo reale, riuscendo a prendere decisioni in pochi millisecondi, superando la velocità di un essere umano.

Infine presentiamo una tabella che riassume le differenze rilevanti dei due algoritmi, evidenziate in questa sezione:

Caratteristica	Agente XGBoost (Offline)	Agente MLP (Online)
Paradigma	Statico (Batch Learning)	Dinamico (Incremental Learning)
Training	Una tantum (pesante)	Continuo (leggero per step)
Overhead Runtime	Nullo (Solo inferenza)	Presente (Backpropagation)
Memoria	Fissa (Modello pre-addestrato)	Variabile (Buffer di memoria)
Adattabilità	Nulla (Modello congelato)	Alta (Impara mentre gioca)
Explainability	Alta	Praticamente nulla
Rischio Principale	Overfitting sul dataset	Instabilità
Win rate	Alto	Alto, ma leggermente peggiore
Performance	Buone	Ottime (il doppio più veloce)

Tabella 5: Confronto tabellare diretto tra le due implementazioni.

5 Possibili Miglioramenti

Nonostante i buoni risultati, l'agente può essere ulteriormente migliorato integrando tecniche più avanzate per la gestione dell'incertezza.

5.1 Calcolo Probabilistico Globale

Attualmente, nella fase di guessing, l'agente si affida a una stima probabilistica fornita dal modello di Machine Learning (o a una scelta casuale sulla frontiera nel caso base). Un miglioramento consisterebbe nel calcolare la probabilità esatta di ogni cella coperta risolvendo il sistema di equazioni lineari dato dai vincoli di bordo (*Model Checking*). Questo permetterebbe di scegliere la cella con la probabilità di mina matematicamente più bassa (es. 15% vs 33%), tuttavia sebbene questo potrebbe migliorare il winrate è molto dispendioso computazionalmente $O(2^n)$.

5.1.1 Formulazione Matematica

Il problema può essere modellato come un sistema di equazioni lineari su un campo binario. Sia $x_i \in \{0, 1\}$ una variabile che rappresenta lo stato della cella i -esima coperta (1 se mina, 0 se sicura). Per ogni cella rivelata C_j con valore V_j , deve valere il vincolo:

$$\sum_{i \in \text{Vicini}(C_j)} x_i = V_j - \text{Bandiere}(\text{Vicini}(C_j))$$

Questa formula indica che, data una cella i -esima, la somma delle mine rimanenti tra i vicini coperti è uguale al numero della cella meno le bandiere già piazzate nelle celle vicine nell'intorno 3×3 .

Per calcolare la probabilità esatta che una cella x_k sia una mina, è necessario enumerare tutte le configurazioni valide (S) che soddisfano simultaneamente tutti i vincoli della frontiera e contare in quante di queste x_k vale 1:

$$P(x_k = \text{Mina}) = \frac{\sum_{s \in S} s_k}{|S|}$$

Sebbene questo metodo garantisca il massimo *winrate* teorico possibile, esso è computazionalmente intrattabile per griglie di grandi dimensioni. Nel caso peggiore, quando le celle formano un'unica componente连通的 lungo una frontiera complessa, il numero di possibili assegnazioni scala esponenzialmente rispetto al numero di variabili:

$$\text{Complessità Temporale} \approx O(2^n)$$

Dove n è il numero di celle coperte nella frontiera.

Mentre un approccio basato su Machine Learning fornisce una stima istantanea $O(1)$ dopo l'addestramento, il calcolo esatto potrebbe richiedere secondi o addirittura minuti per singola mossa in fasi avanzate di gioco, rendendolo inadatto per un'esperienza utente fluida o per l'addestramento rapido su migliaia di partite.

5.2 End-Game Solver

Nelle fasi finali della partita, è possibile contare le mine rimanenti per dedurre posizioni per esclusione. L'attuale implementazione usa il conteggio delle mine solo localmente

(Advanced Logic), ma un "End-Game Solver", che verrebbe sfruttato nelle fasi finali della partita, che consideri il numero totale di mine rimaste potrebbe risolvere alcune ambiguità finali. Questo approccio si basa sull'enumerazione delle configurazioni valide (*Constraint Satisfaction Problem - CSP*), e prevede i seguenti passaggi:

1. **Isolamento delle Componenti:** Si identificano tutte le celle ancora coperte. Nelle fasi finali, queste formano spesso piccoli gruppi disgiunti o un'unica componente连通的.
2. **Generazione delle Permutazioni:** Si generano tutte le possibili disposizioni di mine nelle celle coperte che soddisfano i vincoli numerici locali (i numeri sulle celle svelate adiacenti).
3. **Filtro Globale:** Tra le configurazioni valide localmente, si mantengono **solo** quelle in cui il numero totale di mine piazzate corrisponde esattamente a $M_{residui}$.
4. **Deduzione Finale:**
 - Se una cella è vuota in *tutte* le configurazioni rimaste, allora è *sicura* al 100%.
 - Se una cella contiene una mina in *tutte* le configurazioni rimaste allora è una *mina* al 100%.
 - Se lo stato varia, si calcola la probabilità esatta: $P(c_i) = \frac{\text{Configurazioni con mina in } c_i}{\text{Configurazioni Totali}}$.

Sebbene computazionalmente oneroso per l'inizio partita ($O(2^n)$), questo metodo diventa istantaneo ed estremamente potente quando n (celle rimaste) è piccolo ($n < 20$), garantendo la vittoria in situazioni dove l'approccio probabilistico potrebbe fallire.

6 Soluzioni non implementate

Questa sezione descrive tutte quelle soluzioni parziale o fallimentari che alla fine non sono state inserite nell'implementazione finale, illustrando le idee di partenza e le motivazioni che ci hanno spinti a scartare tali opzioni.

6.1 Raccolta dei dati

Nella soluzione finale l'algoritmo salva nel dataset solamente i casi in cui l'agente tira a caso (Guessing). Tuttavia all'inizio avevamo scelto di raccogliere tutte le mosse dell'agente. Questo si è rivelato un problema che portava con sè alcune criticità:

- **Metriche falsate:** Dato che la maggior parte delle mosse vengono eseguite seguendo la logica il dataset abbondava di soluzioni banali o comunque intuibili, e questo comprometteva le nostre metriche sul modello (es. 0.99 accuracy) facendolo sembrare molto intelligente ad un occhio meno attento.
- **Complessità superflua:** Con questa soluzione il modello era inutilmente più complesso, doveva imparare mosse e regole per cui non veniva interpellato, rischiando di approssimare troppo quelle di nostro interesse.

6.2 Scelta delle feature

Prima di arrivare alla rappresentazione definitiva dei dati ce ne sono state alcune intermedie che mancavano di feature importanti.

1. **Modello 3x3:** Inizialmente memorizzavamo nel dataset solo l'intorno stretto della cella (le 8 celle intorno ad essa). Questo causava predizioni *errate* la maggior parte delle volte con una accuracy di 0.30 poiché il modello non aveva abbastanza dati per predirre la mossa giusta da fare. Abbiamo deciso quindi di allargare il suo campo visivo.
2. **Modello 5x5:** Questo è stata una rappresentazione decisamente più performante. Veniva raccolto un intorno sufficiente per le predizioni (24 celle), che tuttavia rendeva il nostro modello ancora parzialmente miope.
3. **Densità di mine:** Dopo vari cicli di training ci siamo resi conto che forse con un'ultima feature il modello sarebbe potuto migliorare: aggiungendo la densità di mine nelle celle rimanenti abbiamo dato la possibilità al classificatore di capire meglio quando poteva *"azzardare"* e quando no, arrivando così alla soluzione finale.

6.3 Classificatori

Prima di passare al classificatore XGBoost avevamo utilizzato un classico **Random Forest** che alla fine non abbiamo implementato: questo perchè, a parità di dati, e nonostante vari fine-tuning degli iperparametri, il Random Forest risultava meno preciso, per raggiungere la precisione del modello XGBoost doveva essere molto più pesante (Overfitting). Guardando i dati sembra che un unico albero complesso sia più efficace per questo obiettivo rispetto a più alberi semplici.

7 Conclusione

Il progetto ha dimostrato con successo l'applicazione di un **Agente Razionale** in un ambiente parzialmente osservabile e stocastico come il Campo Minato.

L'implementazione ha evidenziato come un approccio ibrido, che combina rigorosa deduzione logica (CSP) e euristiche di esplorazione (Frontier Guessing), possa risolvere la maggior parte delle configurazioni di gioco in tempi ridotti.

I limiti riscontrati sono intrinseci alla natura del gioco stesso: esistono configurazioni in cui l'informazione disponibile è insufficiente per una deduzione certa. Tuttavia, l'agente sviluppato rappresenta un ottimo esempio di sistema esperto capace di superare le prestazioni di un giocatore umano medio, eliminando errori di distrazione e applicando sistematicamente regole complesse.

Riferimenti bibliografici

- [1] Prof. Fabio Palomba. 2025-2026, *Contenuto del Corso di Fondamenti di Intelligenza Artificiale*. <https://docenti.unisa.it/027888/home>, Accessed: 2025-2026.
- [2] Richard Kaye, *Minesweeper is NP-complete*, Mathematical Intelligencer, 2000.
- [3] Python Documentation, *Tkinter — Python interface to Tcl/Tk*, <https://docs.python.org/3/library/tkinter.html>
- [4] MLP neural network, *Multilayer Perceptrons in Machine Learning: A Comprehensive Guide*, <https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning>
- [5] Scikit-learn documentation, *Scikit-learn Machine Learning in Python*, <https://scikit-learn.org/stable/>
- [6] XGBoost documentation, *XGBoost Python Documentation*, <https://xgboost.readthedocs.io/en/stable/python/index.html>
- [7] Pandas Documentation, *Pandas Python Documentation*, <https://pandas.pydata.org/docs/>