Analisi Partite League of Legends

Componenti del gruppo

- Domenico Bianchini, [MAT.733249], d.bianchini1@studenti.uniba.it

Link GitHub: progetto

A.A. 2024-2025

Sommario

Capitolo 0 - Introduzione

Il presente progetto è stato sviluppato nell'ambito del corso di ICON 2024/2025, tenuto dal Prof. Nicola Fanizzi presso l'Università degli Studi di Bari Aldo Moro.

L'obiettivo principale del progetto è mettere in pratica le competenze acquisite durante il corso, applicandole all'analisi di un dataset relativo a un videogioco online.

League of Legends

League of Legends è un MOBA (Multiplayer Online Battle Arena) in cui due squadre da cinque giocatori si affrontano con lo scopo di distruggere la base avversaria. Ogni giocatore controlla un "campione", dotato di abilità uniche. La partita si sviluppa in diverse fasi caratterizzate da strategie, combattimenti e gestione delle risorse, come oro e esperienza ottenuti uccidendo minion, mostri neutrali e avversari, che permettono di potenziare il proprio campione e acquistare oggetti.

Condizioni di vittoria e sconfitta:

- **Vittoria:** la partita termina quando una delle due squadre distrugge la base del nemico. La squadra vincente è quindi quella che riesce per prima a distruggere la base del nemico combinando strategia, collaborazione e controllo delle risorse sulla mappa.
- **Sconfitta:** l'opposto della vittoria: la squadra perdente è quella che vede la propria base distrutta.

Definizione formale del gioco

Per il progetto, si è deciso di considerare un modello semplificato del gioco, facilmente trattabile e comprensibile anche da chi non conosce il gioco, inoltre considerare tutte le possibili features, aumenterebbe notevolmente il grado di difficoltà di progettazione e produzione, rendendo il progetto inutilmente complesso.

Il gioco semplificato è definito nel modo seguente:

Sia:

- **G** = $\{g_1, g_2, \ldots, g_n\}$ l'insieme dei giocatori.
- **P** = $\{p_1, p_2, \ldots, p_m\}$ l'insieme delle partite.
- **C** = $\{c_1, c_2, \ldots, c_k\}$ l'insieme dei campioni.

Per ogni partita **p** ∈ **P** giocata da un giocatore **g** ∈ **G**, valgono le seguenti condizioni:

1. Il giocatore utilizza **un solo campione c** ∈ **C** durante la partita.
2. L'**outcome** della partita è definito come:
    - Vittoria.
    - Sconfitta.
3. Le **azioni principali** durante la partita sono:
    - **Kill:** il giocatore **g** uccide un nemico.
    - **Death:** il giocatore **g** viene ucciso da un nemico.
    - **Assist:** il giocatore **g** aiuta un compagno di squadra a uccidere un nemico.

Struttura del progetto

- **Apprendimento supervisionato**
  Questa fase ha l'obiettivo di predire la vittoria o la sconfitta di una partita sulla base delle statistiche di un giocatore, cioè si cerca di capire **quanto un giocatore abbia influenzato positivamente o negativamente l'esito della partita**.
- **Apprendimento della struttura**
  In questa fase si analizzano **le relazioni tra le diverse statistiche** in modo da comprendere come queste dipendano l'una dall'altra per identificare quali siano più rilevanti nella predizione del risultato della partita.
- **Creazione della knowledge base**
  Infine, viene sviluppata una **knowledge base** con la quale si possa fare inferenza sulle partite di un giocatore fornendo una valutazione.

Capitolo 1 - Dataset e preprocessing

Raccolta dei dati

Il dataset che ho usato per questo progetto è stato preso da **Kaggle** e si chiama **League Of Legends EUW challenger game stats.** Questo dataset contiene i dati di gioco dei

300 migliori giocatori di League of Legends nella regione EUW (Europe West, cioè la regione occidentale dell'Europa) e include le statistiche di gioco e i dati sulle vittorie, raccolti tramite l'API di Riot.

Il dataset è composto da 5702 record e 15 features.

Feature	Descrizione	Dominio
ID	Contatore delle partite: identifica univocamente ogni record nel dataset	Numerico intero positivo
kills	Numero di uccisioni effettuate dal giocatore durante una partita	Numerico intero positivo
deaths	Numero di morti del giocatore durante una partita	Numerico intero positivo
assists	Numero di assist del giocatore durante una partita	Numerico intero positivo
killParticipation	Partecipazione alle uccisioni del giocatore, cioè quante uccisioni ha contribuito a creare	Numerico decimale
kda	Rapporto Kill/Death/Assists	Numerico decimale
goldPerMinute	Oro guadagnato dal giocatore per minuto	Numerico decimale
totalMinionsKilled	Numero totale di minion uccisi dal giocatore in una partita	Numerico intero positivo
gold	Oro totale guadagnato dal giocatore	Numerico intero positivo
totalDamageDealt	Danno totale inflitto dal giocatore	Numerico intero positivo
visionScore	Punteggio di visione del giocatore	Numerico intero positivo
visionScorePerMinute	Punteggio di visione per minuto del giocatore	Numerico decimale
skillshotsDodged	Numero di abilità nemiche evitate dal giocatore	Numerico intero positivo
skillshotsHit	Numero di abilità colpite del giocatore	Numerico intero positivo
win	Indica se il giocatore ha vinto la partita (True/False)	Booleano (True/False)

Preprocessing

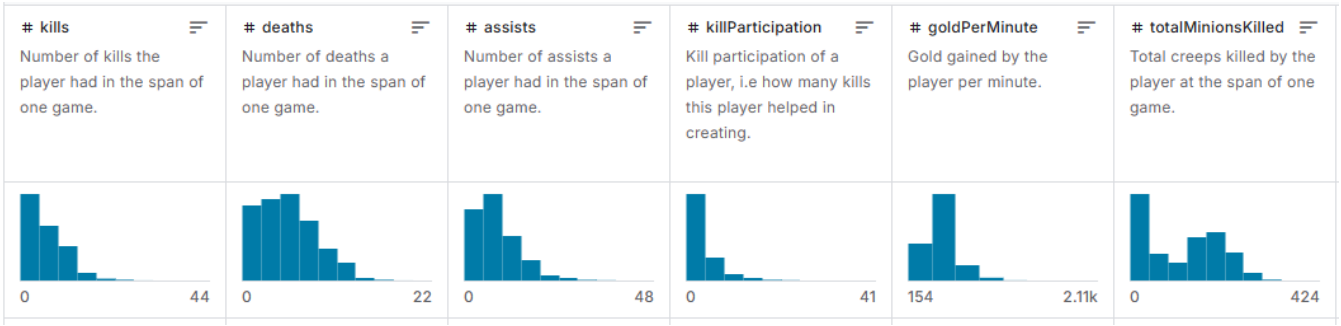
Il preprocessing dei dati è una fase molto importante ai fini di aumentare la qualità del modello, poiché essa è strettamente legata alla qualità dei dati.

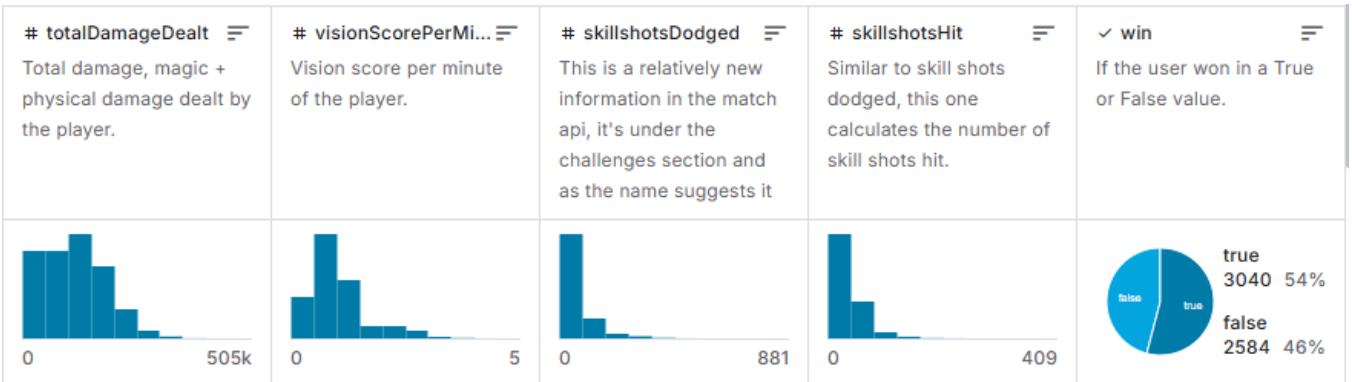
Per questo motivo ho effettuato le seguenti operazioni:

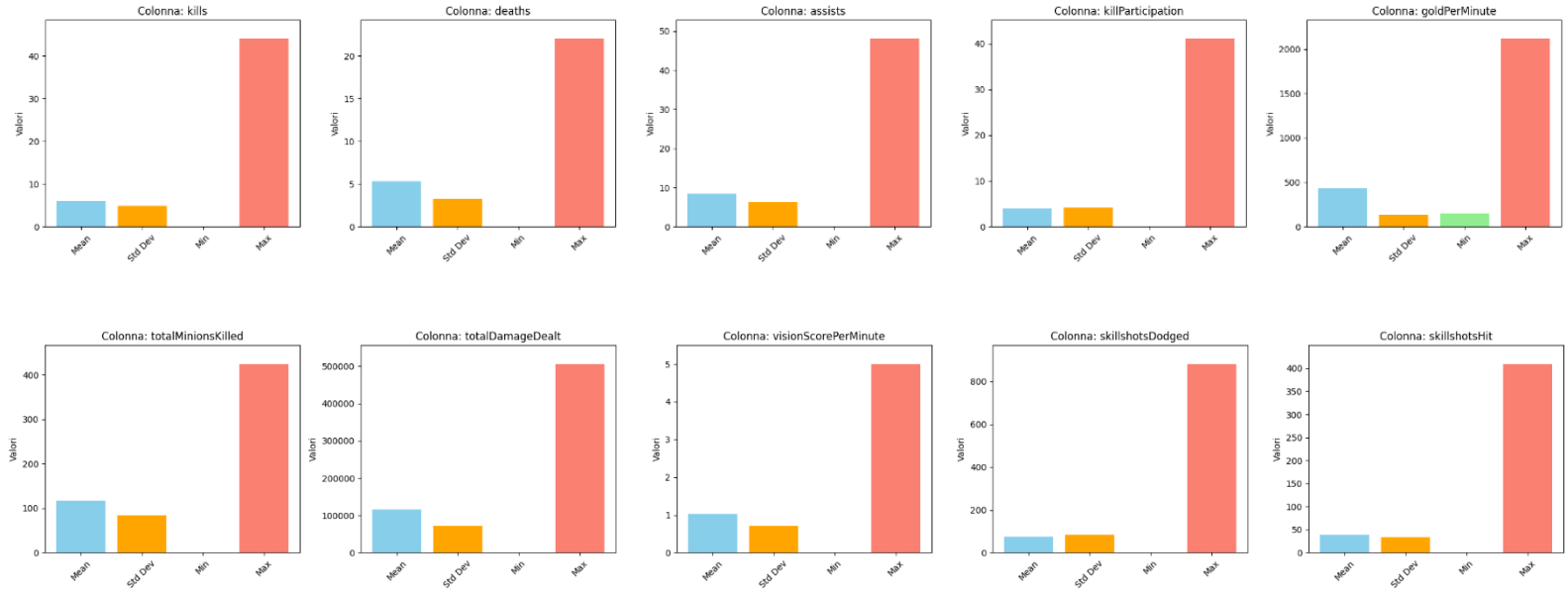
- **Rimozione di features che non danno informazioni utili ai fini della predizione del risultato della partita:**
    - **ID:** è un identificativo univoco per ogni partita quindi non è una feature che influisce sull'outcome.
- **Rimozione di features derivate:**
    - **KDA:** ridondante rispetto a **kills**, **deaths** e **assists**.
    - **Gold:** ridondante rispetto a **goldPerMinute**.
    - **visionScore:** ridondante rispetto a **visionScorePerMinute.**
- **Taglio degli outlier:** per eliminare i valori anomali nelle feature ho deciso di usare il metodo IQR, cioè "taglio le due code della distribuzione di una feature in base ai quantili". Questo permette di rimuovere quei valori troppo lontani dalla media che potrebbero influenzare negativamente i modelli.
    - **Motivazioni:** In una partita squilibrata, le statistiche di un giocatore possono risultare molto alte o molto basse. Questo non riflette necessariamente le capacità del giocatore, ma dipende dal contesto della partita. Ad esempio:
        - Uno o più giocatori si sono disconnessi.
        - Alcuni giocatori hanno comportamenti poco collaborativi che penalizzano la squadra.
- **Normalizzazione delle feature:** alcuni modelli, come le SVM, lavorano sulle distanze tra i punti dati e quindi sono molto sensibili alla scala delle feature. Normalizzare le feature permette di uniformare i valori e porta spesso a risultati migliori, rendendo il modello più stabile e preciso.

Realizzazione

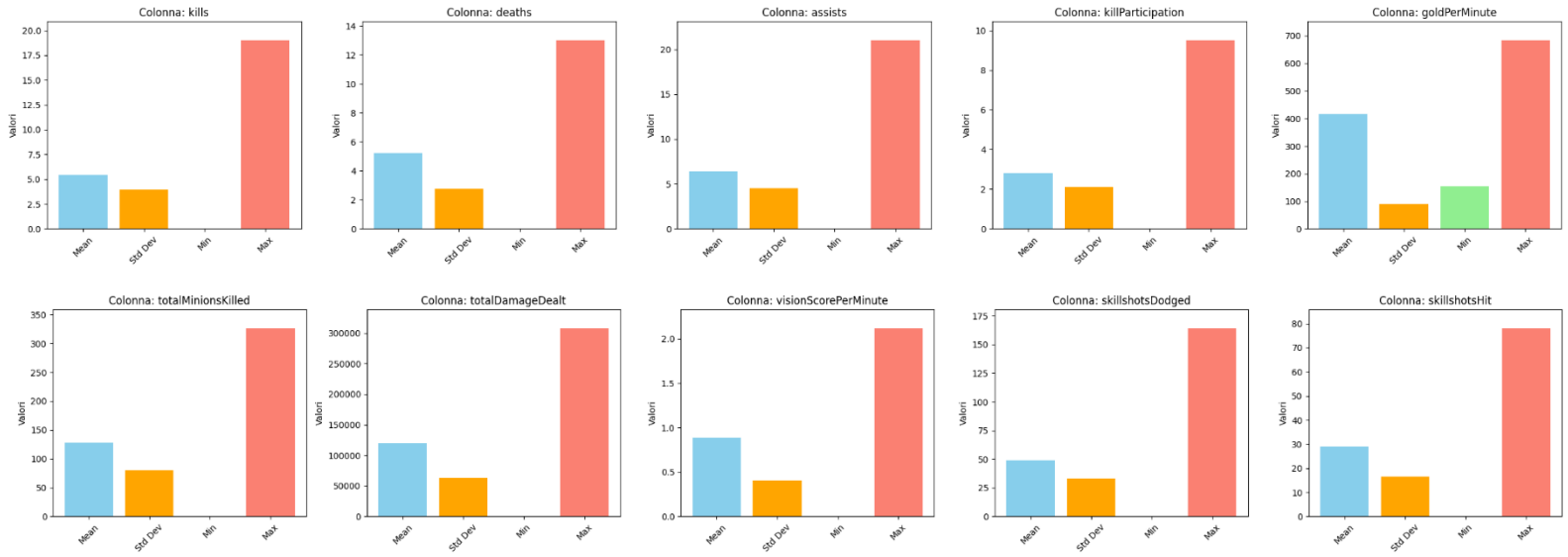
Come prima fase del preprocessing, ho esaminato le distribuzioni delle feature principali utilizzando grafici specifici per individuare eventuali valori anomali.





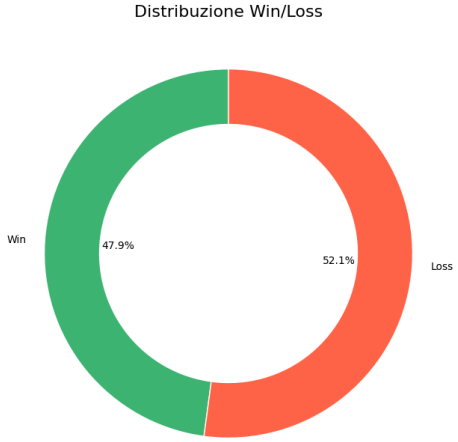


Analizzando le distribuzioni (media, deviazione standard, minimo e massimo) si può notare la presenza di valori estremi, per questo ho usato il metodo **IQR** per eliminare gli outliers di queste feature numeriche.




```
Valori NaN:
kills                  0
deaths                 0
assists                0
killParticipation      0
goldPerMinute          0
totalMinionsKilled     0
totalDamageDealt       0
visionScorePerMinute   0
skillshotsDodged       0
skillshotsHit          0
win                    0
dtype: int64

Valori duplicati: 0
```



Risultato

Come si può notare, le distribuzioni sono state bilanciate eliminando i valori irrilevanti o troppo distanti dalla media.

I grafici a barre mostrano che le medie delle statistiche sono più equilibrate rispetto alla situazione iniziale. In alcuni casi, la deviazione standard è leggermente aumentata, ma questo non rappresenta un problema in quanto riflette meglio la realtà dato che le prestazioni di un giocatore variano.

Non sono stati inseriti grafici post-normalizzazione perché i grafici pre-normalizzazione rappresentano valori più "naturali" e interpretabili dall'utente; la normalizzazione è stata tuttavia applicata prima dell'addestramento quando richiesta dai modelli.

Capitolo 2 - Apprendimento Supervisionato

L'apprendimento supervisionato è una tecnica di machine learning che si basa sul fornire al modello un insieme di dati etichettati, ovvero dati per i quali è già nota l' etichetta, e far apprendere al modello la relazione tra features e labels.

Procedura adottata

Nel nostro caso, ci troviamo di fronte a un problema di classificazione binaria in quanto vogliamo predire se una partita è vinta o persa in base alle statistiche di un giocatore. Per affrontarlo ho seguito questi passaggi principali:

- **Suddivisione dei dati**: il dataset è stato diviso in training e test set.
- **SMOTE (opzionale)**: ho confrontato i risultati con e senza SMOTE anche se la differenza tra le classi era molto piccola, quindi non dovrebbe influire molto, in quanto SMOTE si usa principalmente per dataset sbilanciati.
- **Scelta dei modelli**: ho utilizzato cinque modelli differenti per la classificazione: **Decision Tree** (**DT**), **Random Forest** (**RF**), **Logistic Regression** (**LR**), **Support Vector Machine** (**SVM**) e **Rete Neurale** (**ANN**).
- **Tuning degli iperparametri**: per ciascun modello ho usato la tecnica di **GridSearch** per trovare i migliori iperparametri.
- **Addestramento e test**: i modelli sono stati addestrati sul training set e testati sul test set.
- **Valutazione e confronto**: ho valutato le performance usando diverse metriche e confrontato i modelli tra loro.

Suddivisione dei dati

Ho utilizzato la **k-fold cross validation**, che consiste nel dividere il dataset in k parti, addestrare il modello su k-1 parti e testarlo sulla parte rimanente, ripetendo il processo k volte.

Per il tuning degli iperparametri ho inizialmente utilizzato una cross validation su un tuning set. Successivamente, per l'addestramento e la valutazione finale, ho eseguito una seconda cross validation sul test set.

Il dataset essendo relativamente piccolo, ho scelto **k = 10**, perché con k più basso i risultati erano meno stabili, in quanto aumentare k significa dare più importanza alla fase di training.

Scelta dei modelli

I modelli scelti sono stati:

- **Support Vector Machine** (**SVM**): modello che cerca l'iperpiano che separa al meglio le due classi.
- **Logistic Regression(LR)**: modello classico di classificazione binaria basato sulla funzione logistica.
- **Decision Tree (DT)**: modello che costruisce un albero di decisione in base alle features.
- **Random Forest (RF)**: Modello di classificazione basato su un insieme di alberi decisionali. Viene utilizzato perché gli alberi singoli sono molto sensibili ai dati e spesso rischiano di andare in overfitting.
- **Rete Neurale (ANN)**: modello complesso basato su un insieme di neuroni e layer.

Non è possibile stabilire a priori quale modello sarà il più efficace per il nostro problema, ma si possono comunque fare alcune osservazioni:

- **SVM e Logistic Regression** sono modelli sui quali ripongo maggiore fiducia dato che sono pensati per problemi di classificazione come il nostro e, in genere, richiedono un minor sforzo computazionale rispetto ad altri approcci.
- **Decision Tree (DT)** è molto sensibile ai dati e tende facilmente all'overfitting.
- **Random Forest (RF)**, essendo costituito da più alberi decisionali, serve a mitigare l'overfitting.

Gli alberi funzionano bene quando le feature sono facilmente separabili, ma nel nostro caso, con una classificazione binaria potrebbero non essere la scelta ottimale.

- **Rete Neurale (ANN)** è un modello complesso quindi solo il confronto pratico determinerà se si adatta bene al nostro problema.

Tuning degli iperparametri

Per trovare i migliori iperparametri per i modelli ho utilizzato la tecnica di **GridSearch**, che esplora tutte le possibilità tra i valori scelti per ogni iperparametro. Questo approccio è molto costoso in termini di calcolo, perché il numero totale di combinazioni si ottiene moltiplicando tra loro i valori disponibili per ciascun iperparametro, ma dato che il nostro dataset è relativamente piccolo ci possiamo permettere questo sforzo computazionale.

L'obiettivo principale era confrontare i modelli tra loro piuttosto che entrare nel dettaglio di ognuno includendo tutti gli iperparametri.

Addestramento dei modelli

L'addestramento è stato effettuato tramite **k-fold cross validation** con k=10.

- **DT**, **RF**, **LR** e **SVM** hanno richiesto tempi di calcolo contenuti.
- **ANN** inizialmente non riusciva a convergere con poche epoche ma in seguito aumentando il numero di epoche e fissando un limite superiore, il problema si è risolto. L'addestramento è stato leggermente più lungo rispetto agli altri modelli, come era prevedibile data la maggior complessità del modello.

Valutazione dei modelli

Le metriche utilizzate sono state:

- **Learning curve**: per verificare overfitting o underfitting.
    - **Precision**: percentuale di predizioni positive corrette.
    - **Recall**: percentuale di predizioni positive corrette rispetto al totale dei positivi reali.
    - **F1-Score**: media armonica tra precision e recall.
- Invece di utilizzare le curve di apprendimento, che mostrano l'andamento di una metrica specifica, ho preferito rappresentare l'**errore relativo alla metrica**. In questo modo, invece di ottenere un grafico che cresce con

l’aumentare del numero di elementi nel test set, l’andamento dell’errore forma curve simili a rami di iperbole.

Ho confrontato i risultati per ogni modello con e senza SMOTE per vedere se ci fossero differenze significative.

Confronto dei modelli

Per ciascun modello ho riportato:

- Breve descrizione
- Iperparametri
- Risultati senza SMOTE
- Risultati con SMOTE

E alla fine conclusioni personali e osservazioni finali.

Questo approccio permette di avere una visione completa delle prestazioni dei modelli sul nostro dataset, evidenziando punti di forza e limiti di ciascuno.

Confrontando la distribuzione delle classi:

- **Senza SMOTE** → win=0: 1887, win=1: 1733 (dataset leggermente sbilanciato).
- **Con SMOTE** → win=0: 1887, win=1: 1887 (dataset bilanciato).

Decision Tree

Un **Decision Tree** è un modello di classificazione (o regressione) strutturato come un albero binario: i nodi interni rappresentano test su feature, mentre le foglie contengono le previsioni finali.

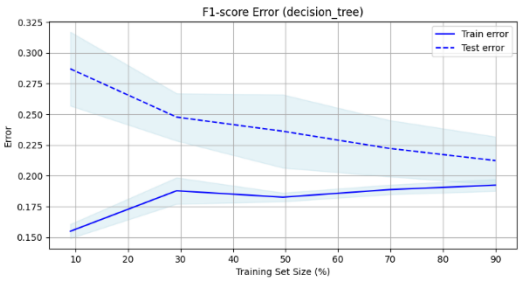
Iperparametri

- **criterion**: funzione per valutare la qualità delle suddivisioni.
    - **'gini'**: imposta l’impurità di Gini come criterio.
    - **'entropy'**: si basa sull’entropia dell’informazione.
    - **'log_loss'**: adotta la perdita logaritmica per problemi di classificazione.
- **max_depth**: profondità massima dell’albero.
    - **[5, 10, 15]**: impone un limite superiore.

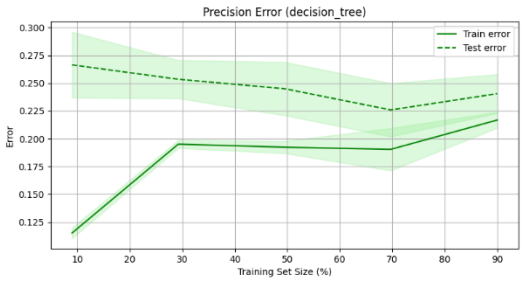
- **min_samples_split**: numero minimo di campioni richiesti per dividere un nodo.
    - **[2, 5, 10]**: valori maggiori riducono il rischio di overfitting, ma aumentano quello di underfitting.

Senza SMOTE

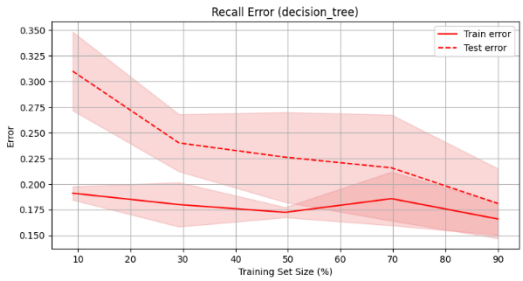
- Iperparametri ottimali: **criterion='entropy'**, **max_depth=5**, **min_samples_split=2**
- Accuratezza di validazione: **0.7890**



Training Size (%)	Train Error	Test Error
9.0%	0.155	0.287
29.2%	0.188	0.248
49.5%	0.183	0.236
69.7%	0.189	0.222
90.0%	0.192	0.212



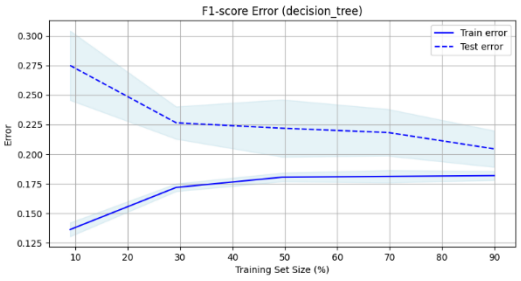
Training Size (%)	Train Error	Test Error
9.0%	0.115	0.266
29.2%	0.195	0.253
49.5%	0.192	0.245
69.7%	0.190	0.226
90.0%	0.217	0.241



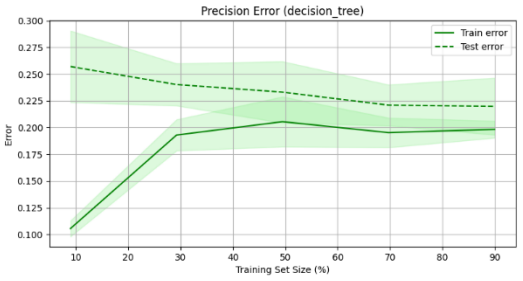
Training Size (%)	Train Error	Test Error
9.0%	0.191	0.310
29.2%	0.180	0.240
49.5%	0.173	0.226
69.7%	0.186	0.216
90.0%	0.166	0.181

Con SMOTE

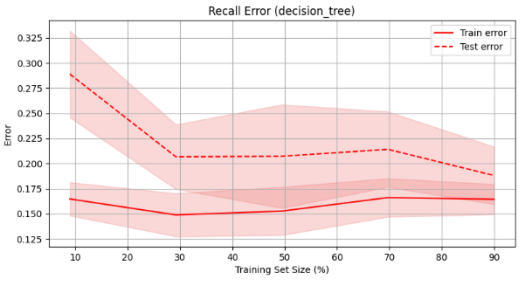
- Iperparametri ottimali: **criterion='gini'**, **max_depth=5**, **min_samples_split=10**
- Accuratezza di validazione: **0.7912**



Training Size (%)	Train Error	Test Error
9.0%	0.136	0.275
29.2%	0.172	0.227
49.5%	0.181	0.222
69.7%	0.181	0.218
90.0%	0.182	0.204



Training Size (%)	Train Error	Test Error
9.0%	0.106	0.257
29.2%	0.193	0.240
49.5%	0.206	0.233
69.7%	0.195	0.221
90.0%	0.198	0.220



Training Size (%)	Train Error	Test Error
9.0%	0.165	0.289
29.2%	0.149	0.207
49.5%	0.153	0.207
69.7%	0.166	0.214
90.0%	0.164	0.188

Random Forest

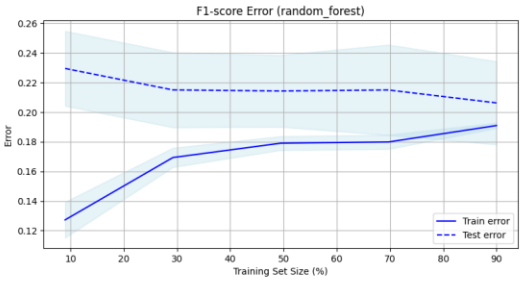
La Random Forest è un modello di classificazione basato su un insieme di alberi decisionali con lo scopo di migliorare la robustezza e ridurre l'overfitting rispetto a un singolo albero.

Iperparametri

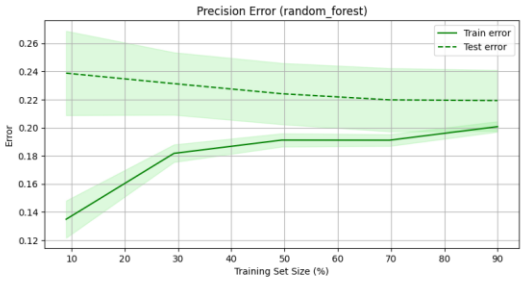
- **n_estimators**: numero di alberi nella foresta.
    - **[25, 50, 100].**
- **max_depth**: profondità massima di ciascun albero.
    - **[5, 10, 20]**: analogo a decision tree.
- **min_samples_split**: analogo al decision tree, determina il numero minimo di campioni necessari per dividere un nodo.

Senza SMOTE

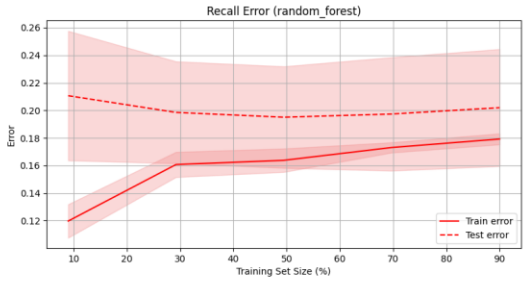
- Iperparametri ottimali: **max_depth=5**, **min_samples_split=10**, **n_estimators=50**
- Accuratezza di validazione: **0.8003**



Training Size (%)	Train Error	Test Error
9.0%	0.127	0.230
29.2%	0.169	0.215
49.5%	0.179	0.214
69.7%	0.180	0.215
90.0%	0.191	0.206



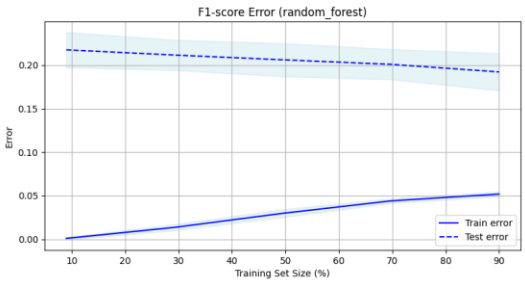
Training Size (%)	Train Error	Test Error
9.0%	0.135	0.239
29.2%	0.182	0.231
49.5%	0.191	0.224
69.7%	0.191	0.220
90.0%	0.201	0.219

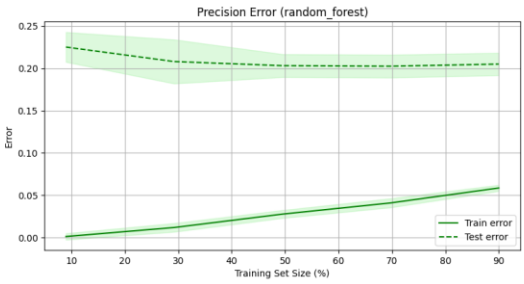


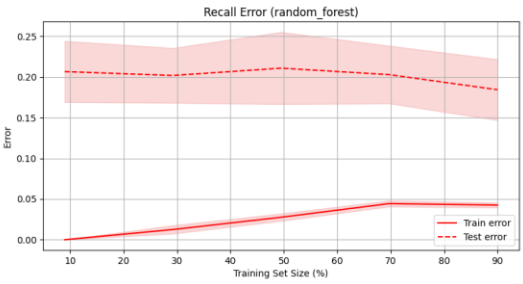
Training Size (%)	Train Error	Test Error
9.0%	0.120	0.211
29.2%	0.161	0.199
49.5%	0.164	0.195
69.7%	0.173	0.197
90.0%	0.179	0.202

Con SMOTE

- Iperparametri ottimali: **max_depth=10**, **min_samples_split=2**, **n_estimators=50**
- Accuratezza di validazione: **0.8074**







Training Size (%)	Train Error	Test Error
9.0%	0.001	0.218
29.2%	0.014	0.212
49.5%	0.030	0.206
69.7%	0.044	0.201
90.0%	0.052	0.192

Training Size (%)	Train Error	Test Error
9.0%	0.001	0.225
29.2%	0.012	0.208
49.5%	0.028	0.203
69.7%	0.041	0.203
90.0%	0.059	0.205

Training Size (%)	Train Error	Test Error
9.0%	0.000	0.207
29.2%	0.013	0.202
49.5%	0.028	0.211
69.7%	0.044	0.203
90.0%	0.043	0.184

Logistic Regression

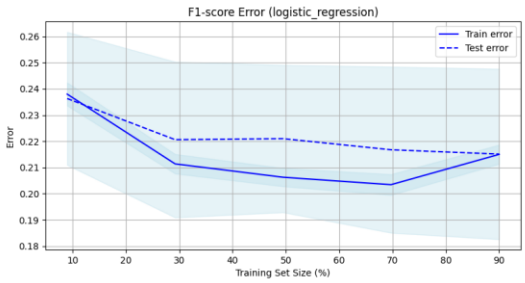
La Logistic Regression è un modello utilizzato principalmente per problemi di classificazione binaria. La sua funzione di base è la **sigmoide**, che mappa valori da uno spazio multidimensionale in un intervallo [0,1], permettendo di stimare la probabilità di appartenenza a ciascuna classe.

Iperparametri

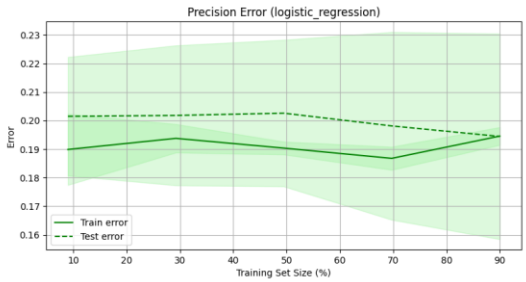
- **penalty**: specifica il tipo di regolarizzazione applicata al modello.
  - **'l2'.**
- **C**: coefficiente di penalizzazione.
  - **[0.01, 0.1, 1, 10].**
- **solver**: algoritmo per l'ottimizzazione.
  - **'liblinear'**: coordinate descent.
  - **'saga'**: basato sul gradiente stocastico, con variante che usa gradienti medi per velocizzare la convergenza.
- **max_iter**: numero massimo di iterazioni per raggiungere la convergenza.
  - **[1000].**

Senza SMOTE

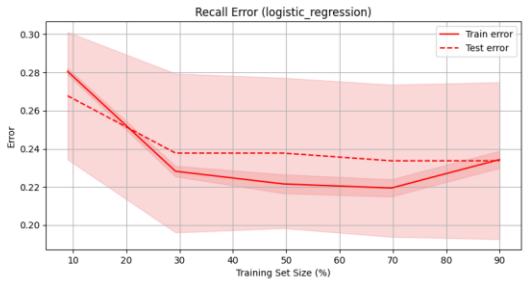
- Iperparametri ottimali: **C=1**, **max_iter=1000**, **penalty='l2'**, **solver='saga'**
- Accuratezza di validazione: **0.7989**



Training Size (%)	Train Error	Test Error
9.0%	0.238	0.236
29.2%	0.211	0.221
49.5%	0.206	0.221
69.7%	0.203	0.217
90.0%	0.215	0.215



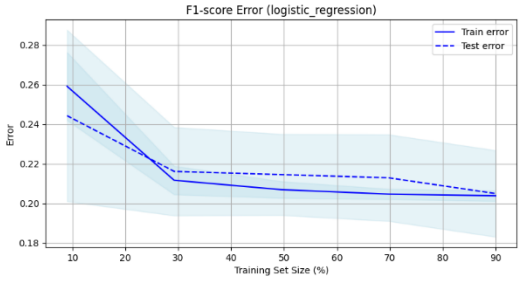
Training Size (%)	Train Error	Test Error
9.0%	0.190	0.201
29.2%	0.194	0.202
49.5%	0.190	0.203
69.7%	0.187	0.198
90.0%	0.195	0.194



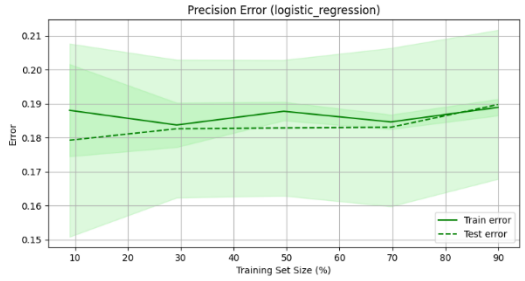
Training Size (%)	Train Error	Test Error
9.0%	0.281	0.268
29.2%	0.228	0.238
49.5%	0.222	0.238
69.7%	0.219	0.234
90.0%	0.234	0.234

Con SMOTE

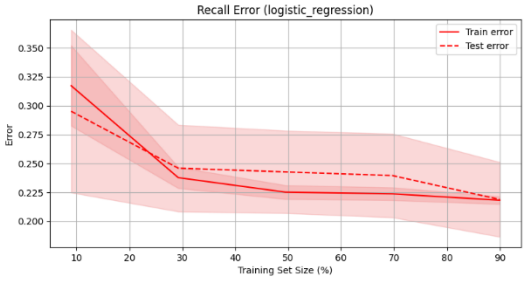
- Iperparametri ottimali: **C=0.1**, **max_iter=1000**, **penalty='l2'**, **solver='liblinear'**
- Accuratezza di validazione: **0.7986**



Training Size (%)	Train Error	Test Error
9.0%	0.259	0.244
29.2%	0.212	0.216
49.5%	0.207	0.215
69.7%	0.205	0.213
90.0%	0.204	0.205



Training Size (%)	Train Error	Test Error
9.0%	0.188	0.179
29.2%	0.184	0.183
49.5%	0.188	0.183
69.7%	0.185	0.183
90.0%	0.189	0.190



Training Size (%)	Train Error	Test Error
9.0%	0.317	0.295
29.2%	0.238	0.246
49.5%	0.225	0.243
69.7%	0.224	0.240
90.0%	0.218	0.219

Support Vector Machine

Le SVM sono un metodo di apprendimento supervisionato che mira a trovare l'iperpiano che separa le classi con il massimo margine.

Iperparametri

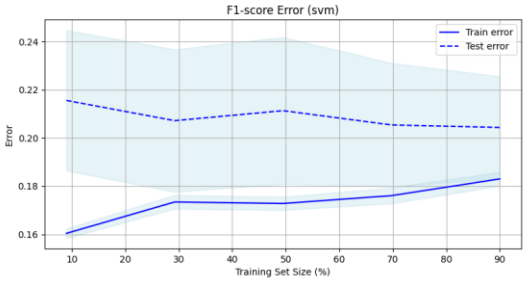
- **C**: parametro di regolarizzazione.
    - **[0.1,0.5, 1, 2]**: controlla il bilanciamento tra massimizzazione del margine e minimizzazione dell'errore.
- **kernel**: funzione kernel utilizzata per mappare i dati in uno spazio ad alta dimensione.

- **'linear'**: kernel lineare, nessuna trasformazione.
- **'rbf'**: kernel a base radiale, adatto a dati non lineari.
- **'poly'**: kernel polinomiale.
- **'sigmoid'**: funzione sigmoide come kernel.

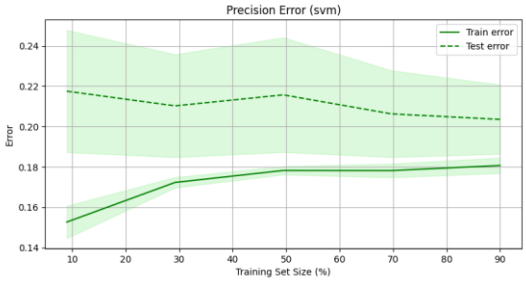
- **gamma**: coefficiente del kernel RBF, polinomiale o sigmoide.
  - **'scale'**: calcolato come l'inverso della somma delle varianze delle feature.
  - **'auto'**: calcolato come l'inverso del numero di feature.

Senza SMOTE

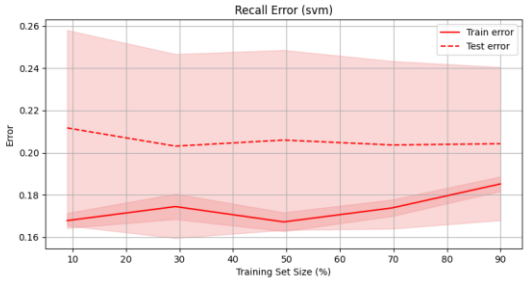
- Iperparametri ottimali: **C=1**, **gamma='scale'**, **kernel='rbf'**
- Accuratezza di validazione: **0.8047**



Training Size (%)	Train Error	Test Error
9.0%	0.160	0.216
29.2%	0.173	0.207
49.5%	0.173	0.211
69.7%	0.176	0.205
90.0%	0.183	0.204



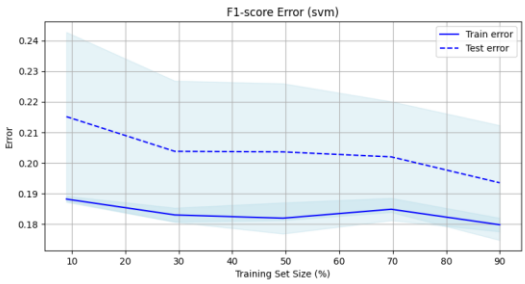
Training Size (%)	Train Error	Test Error
9.0%	0.153	0.218
29.2%	0.172	0.210
49.5%	0.178	0.216
69.7%	0.178	0.206
90.0%	0.181	0.204



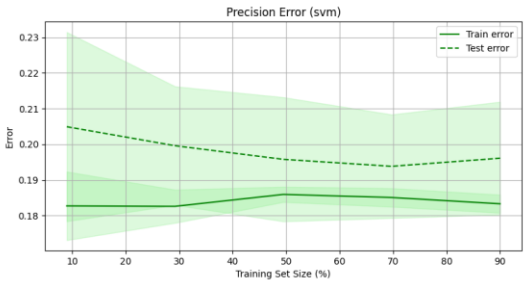
Training Size (%)	Train Error	Test Error
9.0%	0.168	0.212
29.2%	0.174	0.203
49.5%	0.167	0.206
69.7%	0.174	0.204
90.0%	0.185	0.204

Con SMOTE

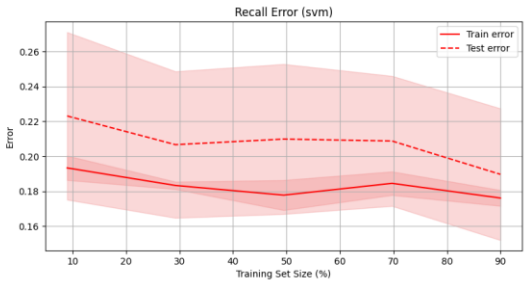
- Iperparametri ottimali: **C=0.5**, **gamma='scale'**, **kernel='rbf'**
- Accuratezza di validazione: **0.8058**



Training Size (%)	Train Error	Test Error
9.0%	0.188	0.215
29.2%	0.183	0.204
49.5%	0.182	0.204
69.7%	0.185	0.202
90.0%	0.180	0.194



Training Size (%)	Train Error	Test Error
9.0%	0.183	0.205
29.2%	0.183	0.200
49.5%	0.186	0.196
69.7%	0.185	0.194
90.0%	0.183	0.196



Training Size (%)	Train Error	Test Error
9.0%	0.193	0.223
29.2%	0.183	0.207
49.5%	0.178	0.210
69.7%	0.185	0.209
90.0%	0.176	0.190

Artificial Neural Network

Le ANN sono modelli di apprendimento ispirati al funzionamento del cervello umano, costituiti da neuroni artificiali organizzati in strati:

- **Strato di input**: riceve i dati delle features.
- **Strati nascosti**: ogni neurone elabora i segnali ricevuti, li combina tramite una funzione matematica detta funzione di attivazione e invia il risultato allo strato successivo.
- **Strato di output**: restituisce il risultato finale che può essere una classificazione o un valore numerico.

Funzionamento:
Ogni connessione tra neuroni ha un peso che determina l'importanza del segnale. Il modello impara modificando questi pesi durante l'allenamento per migliorare le predizioni. Il processo di aggiornamento dei pesi si chiama **backpropagation** e utilizza algoritmi di ottimizzazione come **gradient descent** o **Adam**.

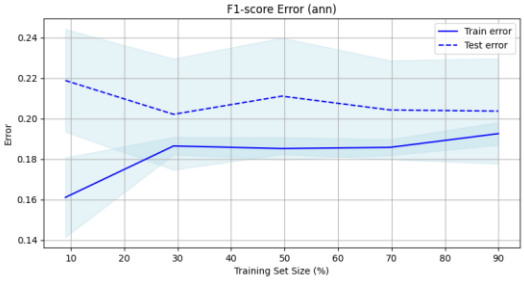
Iperparametri

- **hidden_layer_sizes**: configurazione dei layer nascosti.
    - **[(20,), (40,), (20, 10), (40,20)]**: specifica il numero di nodi per ciascun layer.
- **activation**: funzione di attivazione dei layer nascosti.
    - **'logistic'**: funzione sigmoide.
    - **'relu'**: Rectified Linear Unit, molto comune.
- **solver**: algoritmo di ottimizzazione.
    - **'sgd'**: discesa del gradiente stocastico.
    - **'adam'**: algoritmo avanzato, efficace su molti dataset.
- **alpha**: parametro di regolarizzazione L2.
    - **[0.0001, 0.05]**: valori più alti riducono l'overfitting.
- **learning_rate**: strategia di aggiornamento del tasso di apprendimento.
    - **'constant'**: tasso di apprendimento fisso.
    - **'adaptive'**: riduce il tasso se le prestazioni non migliorano.
- **max_iter**: numero massimo di iterazioni per l'allenamento.

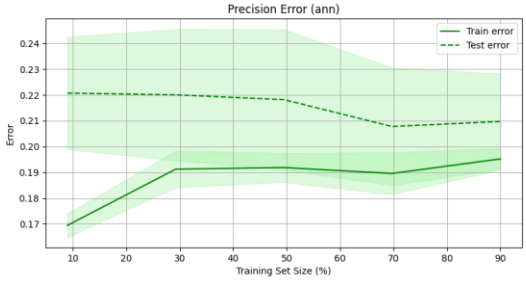
- **[3000]**: più iterazioni permettono di affinare il modello, ma aumentano il costo computazionale (provando più valori di max_iter ho notato che il modello converge mediamente a 3000 iterazioni).

Senza SMOTE

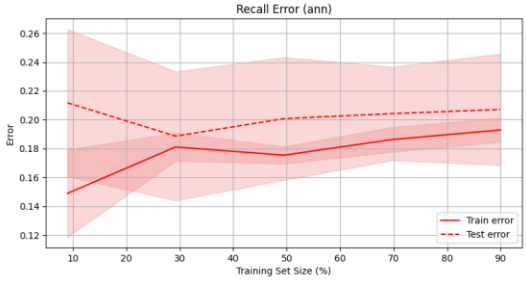
- Iperparametri ottimali: **activation='relu'**, **alpha=0.0001**, **hidden_layer_sizes=(40,)**, **learning_rate='constant'**, **max_iter=3000**, **solver='adam'**
- Accuratezza di validazione: **0.8072**



Training Size (%)	Train Error	Test Error
9.0%	0.161	0.219
29.2%	0.186	0.202
49.5%	0.185	0.211
69.7%	0.186	0.204
90.0%	0.193	0.204



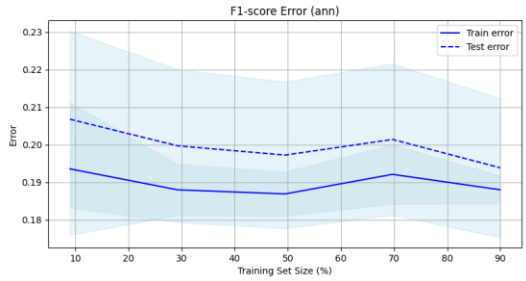
Training Size (%)	Train Error	Test Error
9.0%	0.169	0.221
29.2%	0.191	0.220
49.5%	0.192	0.218
69.7%	0.190	0.208
90.0%	0.195	0.210



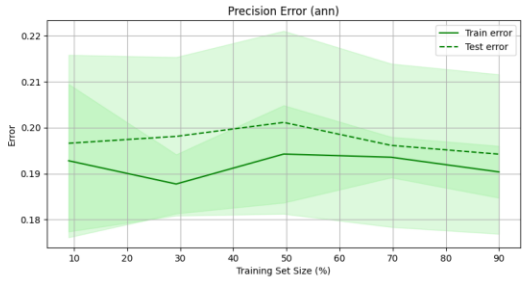
Training Size (%)	Train Error	Test Error
9.0%	0.149	0.212
29.2%	0.181	0.189
49.5%	0.175	0.201
69.7%	0.186	0.204
90.0%	0.193	0.207

Con SMOTE

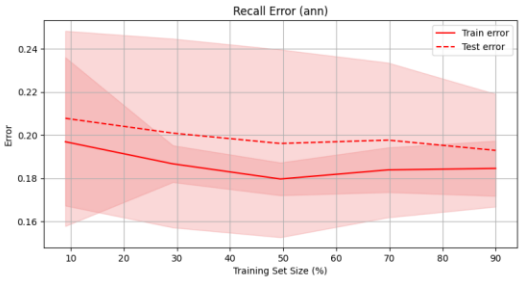
- Iperparametri ottimali: **activation='relu'**, **alpha=0.0001**, **hidden_layer_sizes=(20,)**, **learning_rate='constant'**, **max_iter=3000**, **solver='adam'**
- Accuratezza di validazione: **0.8074**



Training Size (%)	Train Error	Test Error
9.0%	0.194	0.207
29.2%	0.188	0.200
49.5%	0.187	0.197
69.7%	0.192	0.201
90.0%	0.188	0.194



Training Size (%)	Train Error	Test Error
9.0%	0.193	0.197
29.2%	0.188	0.198
49.5%	0.194	0.201
69.7%	0.194	0.196
90.0%	0.190	0.194



Training Size (%)	Train Error	Test Error
9.0%	0.197	0.208
29.2%	0.187	0.201
49.5%	0.180	0.196
69.7%	0.184	0.198
90.0%	0.185	0.193

Conclusioni

- **Prestazioni generali:** i modelli raggiungono accuracies intorno allo **0.79–0.81**, con l'ANN che ottiene il valore più alto (circa 0.807). Questo significa che, con le feature attuali e il preprocessing applicato, i modelli predicono correttamente l'esito di circa 8 partite su 10 che è un risultato solido per dati reali e rumorosi come quelli delle partite di League of Legends.
- **Confronto modelli:**
    - **ANN** e **SVM** sono i più performanti in termini di accuratezza, l'ANN risulta leggermente migliore ma richiede tempi di addestramento più lunghi.
    - **Logistic Regression** si comporta in modo stabile e vicino alle SVM, dimostrando che parte delle relazioni tra feature e outcome sono effettivamente lineari.
    - **Decision Tree** mostra la performance più bassa e evidenze di overfitting; **Random Forest** migliora rispetto al singolo albero ma può comunque soffrire se non opportunamente regolato.
- **Effetto di SMOTE:** la generazione di esempi sintetici ha bilanciato le classi (1887 vs 1887) e ha portato miglioramenti marginali su alcuni modelli (soprattutto **Random Forest** e **SVM**). Tuttavia l'impatto complessivo è stato contenuto dato che la distribuzione originale non era così sbilanciata da rendere SMOTE indispensabile.

Capitolo 3 - Apprendimento della Struttura

I principi della teoria delle probabilità vengono utilizzati nel ragionamento probabilistico per trarre conclusioni e fare inferenze sui dati.

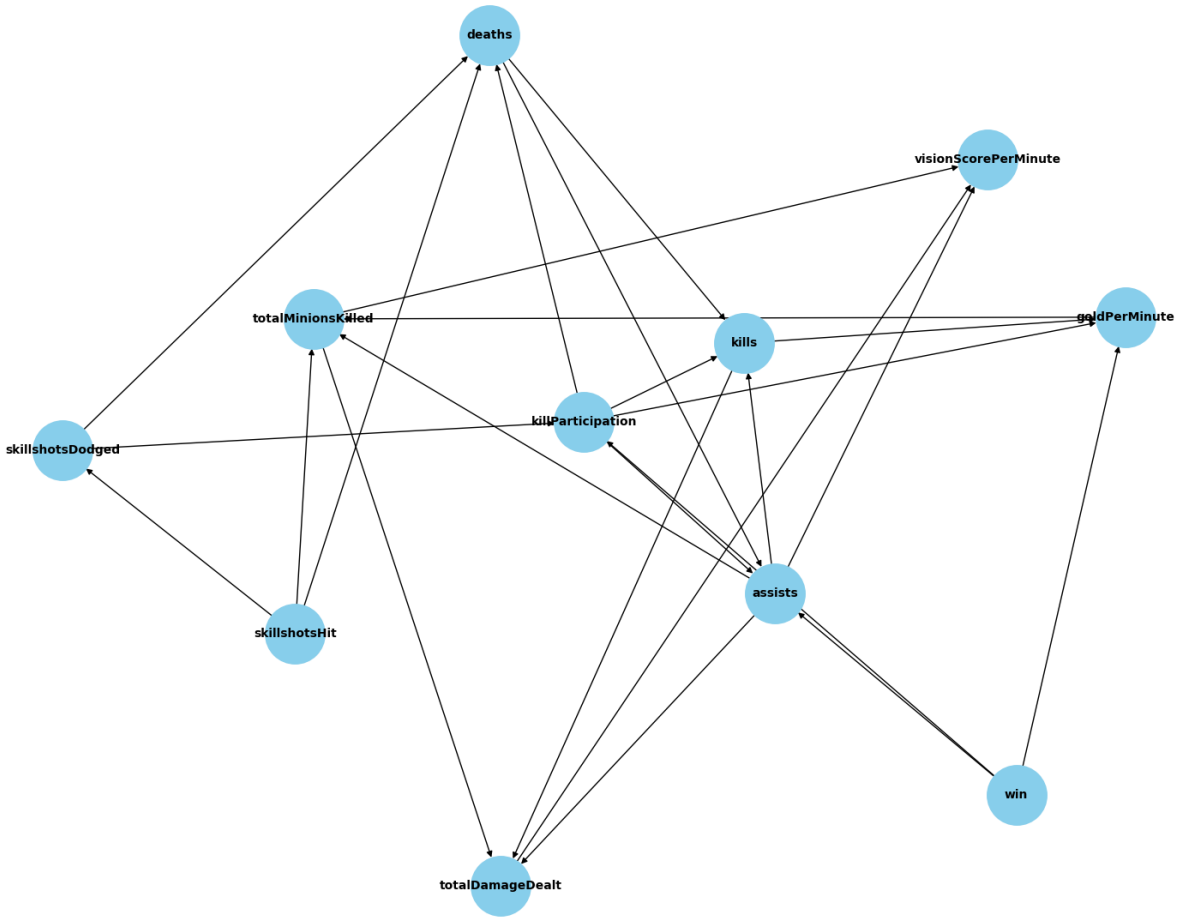
L'apprendimento della struttura rappresenta un passaggio cruciale all'interno del progetto, in quanto consente di individuare le dipendenze e le correlazioni esistenti tra le variabili che descrivono una partita. A differenza dei modelli supervisionati, che si concentrano principalmente sulla predizione dell'outcome (vittoria o sconfitta), consente di analizzare in modo più approfondito quali caratteristiche sono collegate tra loro e come queste connessioni influenzano l'andamento della partita.

Metodologia adottata

Sono stati applicati due approcci distinti: **K2 Score** e **BIC Score**, entrambi implementati attraverso tecniche di ricerca **hill-climbing**.

- **K2**: si basa su un approccio bayesiano che utilizza una metrica che favorisce strutture in grado di spiegare bene i dati osservati senza introdurre troppa complessità. È particolarmente indicato quando si vuole dare priorità a legami più "forti" e diretti tra le variabili.
- **BIC (Bayesian Information Criterion)**: adotta un criterio penalizzato che tiene conto della complessità della rete. Questo approccio, pur individuando relazioni significative, tende a scartare quelle ridondanti o deboli, privilegiando modelli più compatti e robusti.

Modello K2



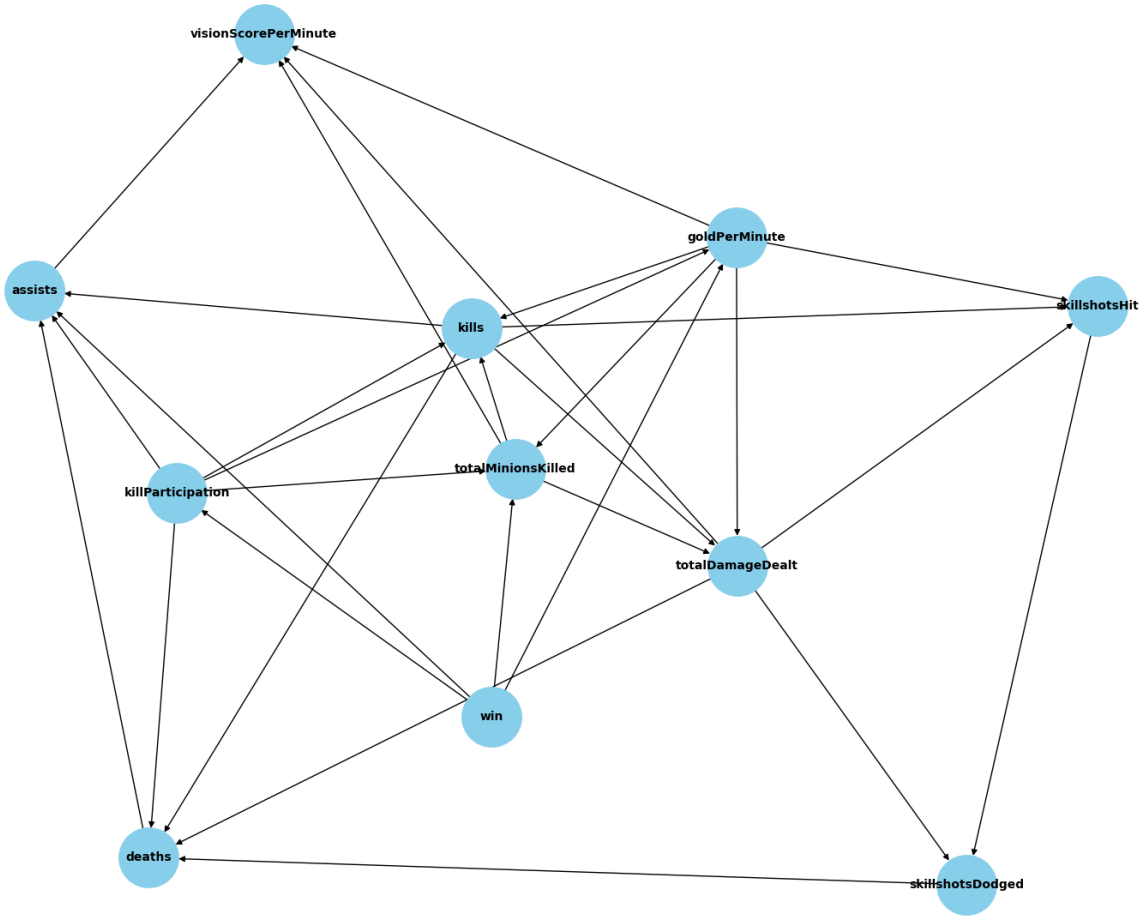
Analisi modello K2

- **Win:** la vittoria è influenzata dagli assist: più assist significano contributi maggiori del giocatore alle kill del team, aumentando le probabilità di vincere.
- **Kills:** le kills influenzano direttamente GoldPerMinute dato che chi ottiene più eliminazioni ottiene più oro per minuto.
- **Deaths:** il numero di morti incide su kills e assist poiché un giocatore che muore spesso contribuisce meno alle eliminazioni e al supporto del team.
- **KillParticipation:** la partecipazione alle kill influenza direttamente le performance complessive del giocatore cioè kills, assist e deaths e contribuisce al guadagno d'oro, riflettendo l'impatto del giocatore sulla partita.
- **TotalMinionsKilled**: Il numero di minion uccisi influisce direttamente sul danno totale, sul guadagno d'oro e sulla visione della mappa dato che i

minion generano oro, quindi un buon farming aumenta le risorse del giocatore, rendendolo più efficace e utile durante la partita.

- **SkillshotsDodged**: la capacità di schivare skillshot influisce sulla partecipazione alle kill e sul numero di morti in quanto un giocatore capace di evitare efficacemente gli attacchi avversari muore meno e così facendo partecipa maggiormente alle uccisioni degli avversari.

Modello AIC



Analisi modello AIC

Il grafo ottenuto con AIC mostra una struttura più densa e connessa, evidenziando numerose interazioni tra le variabili e rappresentando meglio la complessità reale della partita.

Nel modello AIC la vittoria è collegata a quattro feature principali, a differenza di K2, dove **Win** dipendeva soprattutto dagli assist, questo suggerisce un'idea più

articolata del gioco dato che la vittoria non dipende da un singolo fattore, ma da un insieme di performance sia individuali sia di squadra. **Kills**, **deaths**, **assists**, **killParticipation** e **goldPerMinute** mostrano numerose connessioni con altre variabili, il che le rende centrali nel modello. Questa struttura evidenzia come il modello AIC consideri il classico KDA, insieme alla partecipazione alle kill e al guadagno d'oro, come fulcro delle performance che determinano il risultato finale della partita.

Confronto dei modelli e conclusioni

Per valutare i modelli appresi, ho confrontato gli score K2 e AIC:

- **K2 Score**: -35211.1059
- **AIC Score**: -34648.3861

Gli score sono valori negativi: più alto (meno negativo) è lo score, migliore è il modello rispetto al criterio utilizzato. Nel nostro caso, AIC (-34648.3861) risulta essere migliore di K2 (-35211.1059).

Questo indica che il modello AIC si adatta meglio ai dati osservati, catturando un maggior numero di interazioni tra le variabili.

Ho deciso di generare degli esempi per vedere se il modello è coerente con la realtà.

kills	deaths	assists	win
2	0	2	1
2	1	1	1
1	1	1	0
0	2	0	0
2	1	2	1

Si noti come buone performance (kills e assists elevate e deaths contenute) corrispondano a Win = 1 (vittoria), mentre il contrario (kills e assists bassi e deaths alti) corrispondano a Win = 0 (sconfitta), confermando l'impatto delle principali metriche individuali sul risultato della partita.

In conclusione, il modello **K2** privilegia la semplicità e la leggibilità, mentre il modello con **AIC** punta a una descrizione più ricca e articolata del dominio, sacrificando in parte l'interpretabilità. L'utilizzo congiunto dei due approcci permette quindi di ottenere sia una visione chiara dei fattori determinanti la vittoria, sia una panoramica più dettagliata delle relazioni interne tra le variabili della partita.

Capitolo 4 - Knowledge Base

A questo punto del progetto possiamo concludere integrando una **knowledge base** per inferire conoscenza dai dati, utilizzando **Prolog**, un linguaggio di programmazione logica.

Costruzione della Knowledge Base

L'obiettivo è ragionare su una partita di **League of Legends** e utilizzare la knowledge base per fare inferenze sulle statistiche dei giocatori.

Ci poniamo quindi il seguente obiettivo: data una base di conoscenza composta da fatti relativi alle statistiche di una partita e da regole che descrivono le relazioni tra queste statistiche, vogliamo fornire una **valutazione complessiva del giocatore**.

Fatti

In Prolog un **fatto** rappresenta un'informazione considerata vera. Nel nostro caso, ogni fatto descrive una singola partita presente nel dataset. Ogni partita è rappresentata come un fatto nel formato:

partita(GameID, Kills, Deaths, Assists, KillParticipation, GoldPerMinute, TotalMinionsKilled, TotalDamageDealt, VisionScorePerMinute, SkillshotsDodged, SkillshotsHit, Win).

Un esempio di fatto potrebbe essere:

partita(g25, 11.0, 4.0, 18.0, 7.25, 877.3106463241417, 72.0, 64282.0, 0.0, 50.0, 45.0, true).

Nella partita identificata da g25, il giocatore ha realizzato 11 uccisioni, 4 morti e 18 assist, con una kill participation di 7.25. Ha ottenuto circa 877 gold per minuto, 72 minion uccisi e 64.282 danni inflitti. In più ha schivato 50 skillshot, colpito 45 skillshot e infine ha vinto la partita.

Regole

Una **regola** in Prolog è una dichiarazione che descrive una relazione tra fatti. Le regole usano una combinazione di fatti e altre regole per dedurre nuove informazioni. Ogni regola ha una condizione che deve risultare vera affinché la conclusione (posta dopo :-) sia verificata.

- **Regola not_newbie(Game):** il giocatore non è più inesperto se ha giocato più di 50 partite.

```
not_newbie(Game) :-
    sub_atom(Game, 1, _, 0, ID),
    atom_number(ID, Num),
    Num > 50.
```

  Questo significa che le partite con un GameID maggiore di 50 sono considerate come partite in cui il giocatore ha un minimo di esperienza nel gioco.

- **Regola kda(Game, KDA):** calcola il KDA della partita come (kills+assists)/max(1, deaths).

```
kda(Game, KDA) :-
    partita(Game, K, D, A, _, _, _, _, _, _, _, _),
    KDA is (K + A) / max(1, D).
```

- **Regole che definiscono condizioni specifiche:**
  - **Regola high_kda(Game):** Game ha un KDA alto se il KDA calcolato è maggiore di 3.

```
high_kda(Game) :-
    kda(Game, KDA),
    KDA > 3.
```

  - **Regola high_kill_participation(Game):** Game ha una partecipazione alle kill alta se KP > 4.

```
high_kill_participation(Game) :-
    partita(Game, _, _, _, KP, _, _, _, _, _, _, _),
    KP > 4.
```

- **Regola high_gpm(Game):** Game ha un efficienza economica alta se gold per minuto > 400.

```
high_gpm(Game) :-
    partita(Game, _, _, _, _, GPM, _, _, _, _, _, _),
    GPM > 400.
```

- **Regola good_farm(Game):** Game ha un buon farm se il numero totale di minion uccisi > 180.

```
good_farm(Game) :-
    partita(Game, _, _, _, _, _, CS, _, _, _, _, _),
    CS > 180.
```

- **Regola high_damage(Game):** Game ha un danno totale elevato se Damage > 160000.

```
high_damage(Game) :-
    partita(Game, _, _, _, _, _, _, Damage, _, _, _, _),
    Damage > 160000.
```

- **Regola good_vision(Game):** Game ha un buon vision score se il punteggio visione per minuto > 1.2.

```
good_vision(Game) :-
    partita(Game, _, _, _, _, _, _, _, Vision, _, _, _),
    Vision > 1.2.
```

- **Regola skillful_offense(Game):** Game ha precisione offensiva elevata se skillshots andati a segno > 40.

```
skillful_offense(Game) :-
    partita(Game, _, _, _, _, _, _, _, _, _, SHit, _),
    SHit > 40.
```

- **Regola skillful_defense(Game):** Game ha abilità difensiva elevata se skillshots evitati > 80.

```
skillful_defense(Game) :-
    partita(Game, _, _, _, _, _, _, _, _, SDodged, _, _),
    SDodged > 80.
```

- **Regole per la classificazione delle partite:**
  - **Regola astonishing(Game):** Game è considerato astonishing (straordinario) se il giocatore non è più newbie e soddisfa almeno due dei criteri chiave: high KDA, high kill participation, high GPM, good farm, high damage, good vision.

```
astonishing(Game) :-
    not_newbie(Game),
    findall(Criterion, (
        member(Criterion, [high_kda, high_kill_participation, high_gpm, good_farm, high_damage, good_vision]),
        Goal =.. [Criterion, Game],
        call(Goal)
    ), CriteriaMet),
    length(CriteriaMet, Count),
    Count >= 2.
```

  - **Regola normal(Game):** Game è considerato normale se il giocatore non è più newbie e soddisfa almeno uno dei criteri chiave ma meno di due.

```
normal(Game) :-
    not_newbie(Game),
    findall(Criterion, (
        member(Criterion, [high_kda, high_kill_participation, high_gpm, good_farm, high_damage, good_vision]),
        Goal =.. [Criterion, Game],
        call(Goal)
    ), CriteriaMet),
    length(CriteriaMet, Count),
    Count =:= 1.
```

  - **Regola bad(Game):** Game è considerato negativo se il giocatore non è più newbie e non soddisfa nessuno dei criteri chiave.

```
bad(Game) :-
    not_newbie(Game),
    \+ high_kda(Game),
    \+ high_kill_participation(Game),
    \+ high_gpm(Game),
    \+ good_farm(Game),
    \+ high_damage(Game),
    \+ good_vision(Game).
```

I fatti rappresentano le partite con le principali statistiche del giocatore, mentre le regole combinano questi dati per classificare le partite in categorie come Astonishing, Normal o Bad, basandosi sulle performance complessive in termini di KDA, partecipazione alle kill, farm, danno e visione.

Esempi di query

Di seguito vengono riportati tre esempi per ciascuna delle tre categorie di partite (Astonishing, Normal, Bad), per mostrare come le diverse statistiche influenzino la valutazione.

Astonishing Games

Game ID	Kills	Deaths	Assists	Kill Participation	GPM	CS	Damage	Vision	Skillshots Dodged	Skillshots Hit	Win
50	11	7	10	3	555.003	221	205473	1.1201	36	30	True
51	5	1	5	10	480.346	126	46096	0.515743	51	22	True
52	10	4	8	4.5	552.129	230	172164	0.920796	152	26	True

Normal Games

Game ID	Kills	Deaths	Assists	Kill Participation	GPM	CS	Damage	Vision	Skillshots Dodged	Skillshots Hit	Win
82	3	12	11	1.16667	266.1	41	25364	2.30523	77	27	True
89	0	4	12	3	229.373	7	13319	2.3644	58	22	False
90	2	6	15	2.83333	256.054	40	40072	2.0104	140	71	True

Bad Games

Game ID	Kills	Deaths	Assists	Kill Participation	GPM	CS	Damage	Vision	Skillshots Dodged	Skillshots Hit	Win
61	1	6	4	0.833333	328.904	130	69093	1.07966	25	31	False
62	3	2	1	2	321.521	124	60369	0.737031	34	27	False
63	3	3	3	2	363.126	124	55504	0.591752	29	41	False

Capitolo 5 – Conclusioni

In questo progetto abbiamo analizzato il dataset, costruito modelli di apprendimento supervisionato, applicato reti bayesiane e realizzato una knowledge base in Prolog.

Dai risultati emerge chiaramente come alcune statistiche, come Kills, Deaths, Assists e Kill Participation, influenzino l'esito delle partite. La knowledge base, inoltre, permette di valutare le partite basandosi sulle performance dei giocatori e di capire quali fattori contribuiscono maggiormente al risultato finale.

Tra gli sviluppi futuri possibili ci sono l'inserimento di nuovi dati per aumentare il numero di partite analizzate, l'aggiunta di altre feature per rendere le valutazioni più dettagliate, lo studio dell'evoluzione dei giocatori nel tempo e l'espansione della knowledge base con nuove regole e fatti.