

---

# **VALAI - Progetto Icon 2024/2025**

Porcelli Andrea [775736]

2025-01-13

# VALAI - Progetto Icon 2024/2025

Porcelli andrea mat. [775736]

[github](#)

[web\\_visualization](#)

## Tabella dei Contenuti

- **Capitolo 0**
  - [Introduzione](#)
  - [Valorant](#)
  - [Definizione formale del gioco](#)
  - [Struttura del Progetto](#)
- **Capitolo 1 - Dataset & Preprocessing**
  - [Raccolta dei Dati](#)
  - [Analisi Esplorativa sui Dati](#)
  - [Preprocessing](#)
- **Capitolo 2 - Apprendimento Supervisionato**
  - [Approccio](#)
  - [Suddivisione dei Dati](#)
  - [Scelta dei Modelli](#)
  - [Tuning degli Iperparametri](#)
  - [Addestramento & Test dei Modelli](#)
  - [Valutazione dei Modelli](#)
  - [Confronto dei Modelli](#)
    - \* [Decision tree](#)
    - \* [Random forest](#)
    - \* [Logistic regression](#)
    - \* [Support vector machine](#)
    - \* [Artificial neural network](#)

- [Conclusioni](#)
- **Capitolo 3 - Ragionamento Probabilistico**
  - [Motivazioni](#)
  - [Approccio](#)
  - [Pareri e conclusioni](#)
- **Capitolo 4 - Knowledge Base**
  - [Costruzione della Knowledge Base](#)
  - [Fatti e Regole](#)
  - [Esempi di query](#)
  - [Upgrade](#)
- **Capitolo 5 - Conclusioni**
  - [Risultati Ottenuti](#)
  - [Possibili Sviluppi Futuri](#)

## Capitolo 0 - Introduzione

### Introduzione

Il progetto qui presentato è stato realizzato per il corso di ICON 2024/2025, tenuto dal Professore Fanizzi Nicola presso l'Università degli Studi di Bari Aldo Moro.

Il progetto si pone con l'obiettivo di dimostrare le conoscenze acquisite durante il corso, applicandole a un dataset di un videogioco online.

### Valorant

[Valorant](#) è un videogioco multiplayer competitivo della categoria FPS, che coinvolge due squadre da cinque giocatori ciascuna. Le due squadre si affrontano in due tempi da 12 round ciascuno su una mappa, i due tempi sono chiamati attacco e difesa. Ogni giocatore può scegliere un personaggio, chiamato agente, che possiede abilità uniche ed ha a disposizione delle armi.

Le condizioni di vittoria sono due per ogni round:

- **Se si è in**
  - **Attacco:** la squadra attaccante deve innescare la bomba e difenderla fino alla sua esplosione.
  - **Difesa:** se innescata la bomba, la squadra difensiva deve disinnescarla per vincere il round.
- **Condizioni di vittoria:**
  - **Vittoria:** Vince la squadra che arriva per prima a 13 o in caso di pareggio (12 - 12) si continua come in pallavolo con tre round aggiuntivi. La vittoria viene attribuita a chi ne vince due su tre, nel caso di ulteriore pareggio si continua con la stessa modalità fino a determinare un vincitore.
- **Condizioni di pareggio:**
  - **Pareggio:** Il pareggio avviene su votazioni di entrambe le squadre se la partita si prolunga troppo, ma è molto raro.
- **Condizioni di sconfitta:**
  - **Sconfitta:** opposte a quelle di vittoria.

## Definizione formale del gioco

Ai fini del progetto, si vuole considerare un modello semplificato del gioco, facile da trattare e da analizzare pure da esterni. Una motivazione aggiuntiva è che 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 come segue:

- Dato un insieme di partite  $M$  di un giocatore  $p$  per ogni  $m \in M$  si ha che:
  - $A$  l'insieme dei personaggi,
  - la partita  $m$  è giocata con uno e un solo personaggio  $a \in A$ ,
  - $out \in \{win, loss, draw\}$  l'outcome del gioco, ovvero se la partita è stata vinta/persa o pareggiata,

- azioni durante un round:
  - $p$  può uccidere un nemico *kill*,
  - $p$  può essere ucciso da un nemico *death*,
  - $p$  può aiutare un amico a uccidere un nemico *assist*.

## Obiettivo del Progetto

Ho deciso di avere due finalità per il progetto:

- Dimostrare le competenze acquisite durante il corso di ICON.
- Presentare una demo alla community di valorant per ottenere una API production key, così posso estendere il progetto.

Ho pensato di suddividere il progetto in due task, uno che realizzerò adesso e uno esteso che dopo uno studio più approfondito cercherò di realizzare.

Sia  $SR$  un vettore di dimensione  $n$  dove  $n$  è il numero di round giocati nella partita  $m$ .  $SR[i]$  contiene le statistiche cumulative del giocatore  $p$  al round  $i$ .

- **Task target in questo progetto:** Dato  $SR[n]$  fornire un feedback al giocatore e determinare se la partita  $m$  è più prona alla vittoria o alla sconfitta.
- **[NON REALIZZATO] estensione finale:**  $\forall i$  feedback in real-time e determinare se la partita  $m$  è più prona alla vittoria o alla sconfitta in base a  $SR[i]$ .

Ho deciso di suddividere il progetto in tre sub-task:

- **Sub\_Task 1:** Apprendimento Supervisionato - predire la vittoria o la sconfitta di una partita in base alle statistiche di un giocatore può essere interpretato come “quanto un giocatore ha influenzato positivamente o negativamente la partita”.
- **Sub\_Task 2:** Apprendimento della struttura - Apprendere la struttura del problema e come le varie statistiche dipendo l’una dall’altra, è un informazione cruciale per capire quali sono più importanti per la predizione dell’outcome.
- **Sub\_Task 3:** Creazione di una knowledge base - Creare una knowledge base con la quale si possa fare inferenza sulle partite di un giocatore fornendo una valutazione.

- **prodotto finale:** una base per un sistema di feedback per poi estenderlo a un “online judge”.

**Production\_key:** Avere una production key consente di avere a disposizione uno spettro più ampio di dati e di poter fare analisi più approfondite.

## Struttura del Progetto

```
1 +---bayes # ragionamento probabilistico
2 +---datasets # dataset pre e post processing
3 +---displayers # funzioni per visualizzare i risultati
4 +---img
5 |   +---ANN
6 |   +---DT
7 |   +---LR
8 |   +---RF
9 |   +---SVM
10 |   \---_general
11 +---pre_processing # funzioni per il preprocessing
12 |
13 +---prolog # knowledge base
14 \---supervised_training # supervised learning
```

### Attenzione:

- in `main.py` si trova il codice usato per i capitoli 1 - 2
- in `bayes/bayes.py` si trova il codice usato per il capitolo 3
- in `prolog/prolog_dialog.py` si trova il codice usato per il capitolo 4

## Capitolo 1 - Dataset & Preprocessing

### Raccolta dei Dati

Il dataset che ho scelto di utilizzare è stato trovato su [Kaggle](#), si chiama [my first 1000 valorant games](#).

- Il dataset è composto da 1000 record e 19 Features.

Feature	Descrizione	Dominio
<b>game_id</b>	Identificativo univoco per ogni partita	{1..1000}
<b>episode</b>	Equivalente al significato di stagioni (calcio)	{0..9}
<b>act</b>	Finestre temporali su cui si divide un episodio	{1, 2, 3}
<b>rank</b>	Punteggio competitivo come in scacchi (granmaestro, maestro, maestro internazionale) nel gioco si usano i metalli come sistema di valutazione	{Gold 1, Diamond 3, ..}
<b>date</b>	La data in cui si è svolta la partita	{Date "DD-MM_YYYY"}
<b>agent</b>	Il personaggio giocato in quella partita	{Phoenix, Jett, ..}
<b>map</b>	La mappa in cui si è svolta la partita	{Bind, Ascent, ..}
<b>outcome</b>	L'esito della partita (vittoria/sconfitta/pareggio)	{Win, Loss, Draw}
<b>round_wins</b>	Numero di round vinti	$\mathbb{N}$
<b>round_losses</b>	Numero di round persi	$\mathbb{N}$
<b>kills</b>	Numero di uccisioni di giocatori nemici	$\mathbb{N}$
<b>deaths</b>	Numero di volte in cui si è stati uccisi dai nemici	$\mathbb{N}$
<b>assists</b>	Numero di assist (aiuto ai compagni nell'uccidere un nemico)	$\mathbb{N}$
<b>kdr</b>	Rapporto uccisioni/morti calcolato con la formula $\frac{kills}{max\{1, death\}}$	$\{x \in \mathbb{R} \ x \geq 0\}$
<b>avg_dmg_delta</b>	Differenza media per round tra i danni inflitti e i danni subiti su tutti i round.	$\{x \in \mathbb{R}\}$

Feature	Descrizione	Dominio
headshot_pct	Percentuale di colpi che hanno colpito la testa del nemico (non include colpi mancati)	$\{x \in \mathbb{R} \ 0 \leq x \leq 100\}$
avg_dmg	Danno medio inflitto per round ai nemici	$\{x \in \mathbb{R} \ x \geq 0\}$
acs	Punteggio di combattimento fornito dal sistema di ranking. Media su tutti i round, <b>non si conosce come viene calcolato.</b>	$\{x \in \mathbb{R} \ x \geq 0\}$
num_frag	Posizione nella classifica della squadra in base al numero di uccisioni (1 = più uccisioni).	$\{1, 2, \dots, 5\}$ dove 5 è il numero di giocatori nella squadra

## Analisi Esplorativa sui Dati

Qui di seguito sono riportate alcune statistiche sui dati raccolti:

- Le statistiche erano già state esplorate dalla community di kaggle [statistiche](#)
- Ho riportato di seguito solo quelle più utili per il progetto:

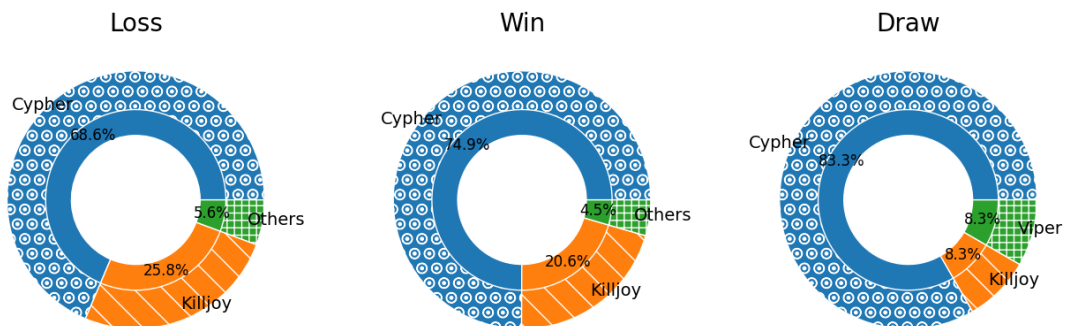
### Nan e duplicati

```
1 NaN values in data:
2 game_id          0
3 episode          0
4 act              0
5 rank             0
6 date             0
7 agent           0
8 map              0
9 outcome          0
10 round_wins      0
```



```
11 round_losses      0
12 kills             0
13 deaths            0
14 assists           0
15 kdr               0
16 avg_dmg_delta     0
17 headshot_pct      0
18 avg_dmg            0
19 acs               0
20 num_frag          0
21
22 Duplicated values in data:0
```

## PERSONAGGIO - OUTCOME



**Figure 1:** agent\_distribution

## KILLS / HEADSHOTS% AVG TIMELINE

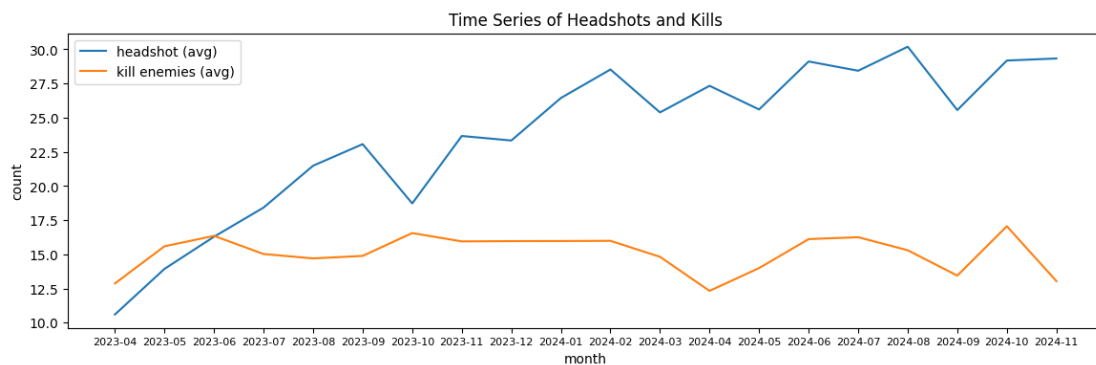


Figure 2: timeline\_headshot\_kills

## DISTRIBUZIONI

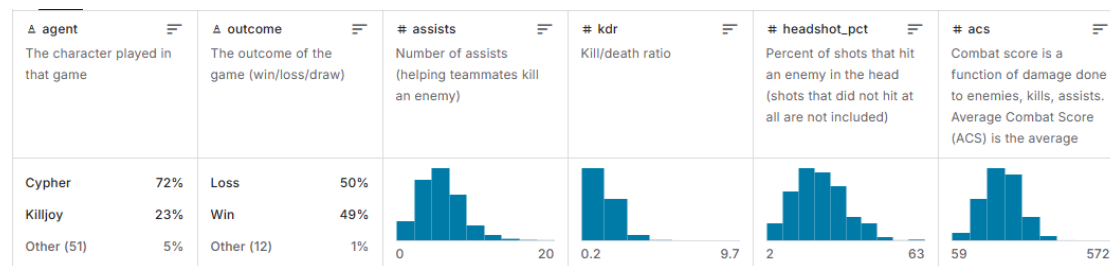


Figure 3: kaggle\_distributions

## Preprocessing

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

Per questo ho deciso di effettuare le seguenti operazioni:

- **Rimozione di Features che non danno informazioni utili ai fini della predizione dell' outcome:** `game_id`, `date`, `episode`, `act`, `rank`, `map`, `num_frag`.
  - `game_id` è un identificativo univoco per ogni partita, non ha senso pensare che sia una feature che influisce sull'outcome.

- `date` la data non influisce sull'outcome.
  - `episode` e `act` sono feature che non influiscono sull'outcome sono dei termini usati in gioco per indicare le finestre temporali di un anno.
  - `rank` è un punteggio che indica il grado del giocatore all'inizio della partita, (analogo a scacchi `granmaestro` / `maestro`) non influisce sulla partita poichè nel sistema di ranking, i giocatori devono avere punteggio simile, il rank quindi non è una feature che contribuisce all' outcome.
  - `map` la mappa non influisce sull'outcome della partita.
  - `num_frag` è la posizione nella classifica della squadra in base al numero di uccisioni, non influisce sull'outcome e inoltre noi vogliamo predire l'outcome basato solo sulle statistiche di un giocatore non su quelle di tutti.
  - `avg_dmg_delta` e `avg_dmg`: dipendono troppo strettamente dal `kdr` più il `kdr` è alto più il `avg_dmg` saranno alti e viceversa.
- **Rimozione di Features Ridondanti:** `round_wins` e `round_losses` sono ridondanti rispetto all'outcome, `deaths` e `kills` sono ridondanti rispetto al `kdr`.
  - **Encoding delle Categorical Features:** ho deciso di utilizzare l'encoding one-hot per la feature categorica `agent` ed `outcome` però come vedremo successivamente regrediscono a features binarie.
  - **Taglio degli outlier:** ho deciso di usare il metodo `IQR` “taglio le due code della distribuzione di una feature in base ai quantili” per eliminare gli outlier su quelle feature che presentavano valori molto distanti dalla media, poichè possono influenzare negativamente i modelli.
    - **motivazioni:** In un game **squilibrato** le statistiche di un giocatore possono essere molto alte o molto basse, ma questo non dipende dal giocatore ma dal fatto che la partita è squilibrata possibili cause (uno o più giocatori si sono disconnessi, uno o più giocatori dimostrano comportamenti lenosi sulla propria squadra)
  - **Normalizzazione delle Features:** I modelli come **SVM** che lavorano sulle distanze sono molto sensibili alla scala delle features, normalizzare le features porta molto spesso a un miglioramento dei risultati.

## Attuazione

Il primo passo nel preprocessing è stato quello di visionare con opportuni grafici la distribuzione delle features.

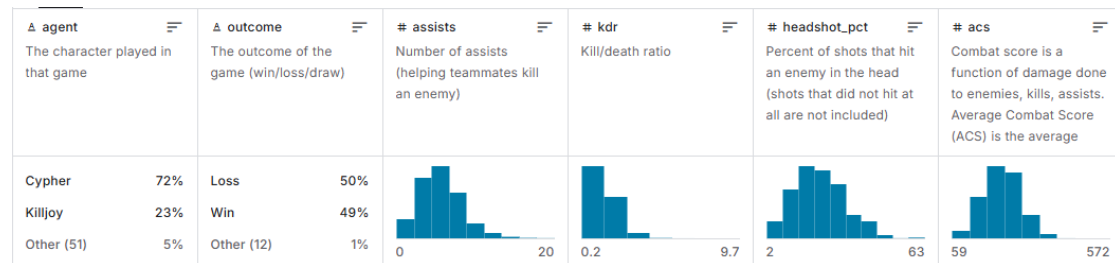


Figure 4: kaggle\_distributions

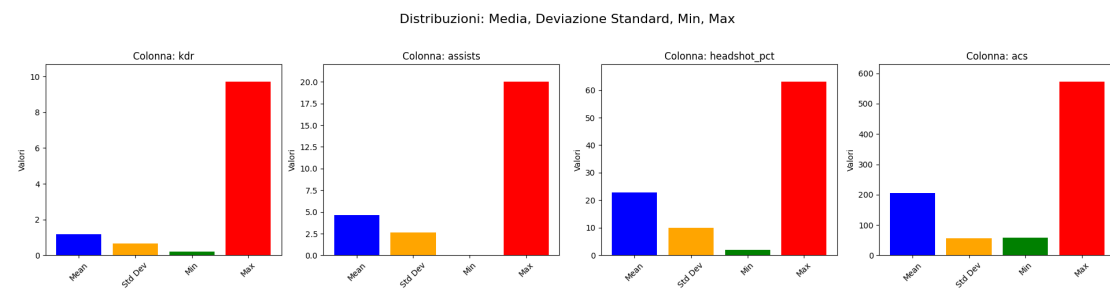


Figure 5: pre\_stats

**Nota:** Si veda la presenza di valori estremi.

**Scelte:**

- L' `agent` è stata trasformata in una feature binaria, in quanto se considerassi quei personaggi con poche partite non avrei sufficienti dati per fare una predizione accurata.
- L' `outcome` è stata trasformata in una feature binaria, in quanto se considerassi il pareggio avrei una motivazione analoga ad `agent`.
- per `assists`, `kdr`, `headshot_pct`, `acs` ho usato il metodo IQR per eliminare gli outlier.

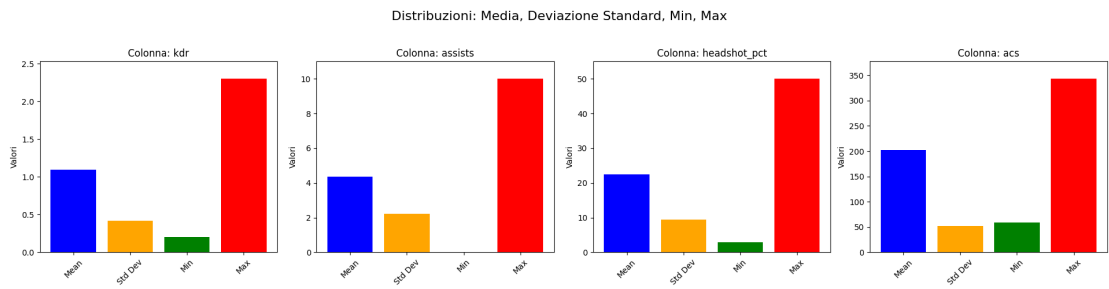


Figure 6: post\_stats

```
1 Valori NaN:
2 game_id      0
3 agent        0
4 kdr          0
5 assists      0
6 headshot_pct 0
7 acs          0
8 outcome      0
9
10 Valori duplicati: 0
```

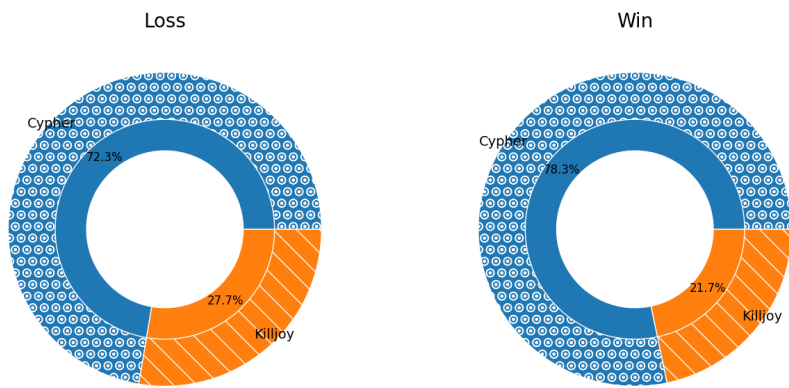
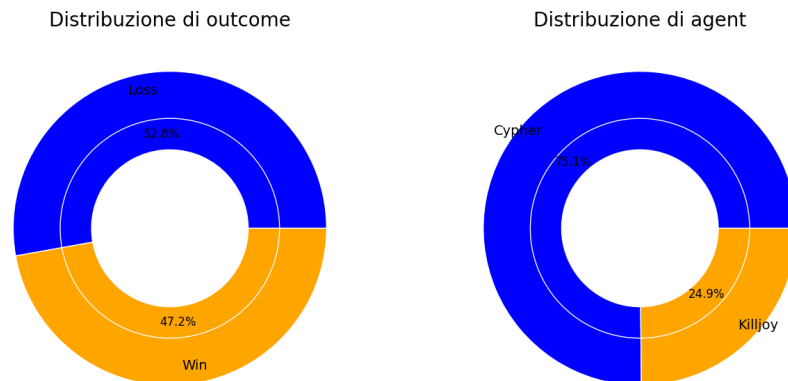


Figure 7: win\_loss\_distribution



**Figure 8:** distribuzioni\_after\_pre\_processing

## Risultato

Come si può vedere abbiamo bilanciato le distribuzioni togliendo eventuali valori irrilevanti o che si distaccavano troppo dalla media.

I grafici a barre mostrano che le medie delle statistiche sono più bilanciate rispetto a prima, ma la standard deviation in qualche caso è aumentata questo non è da considerarsi un problema, poichè è più veritiero che un giocatore abbia delle prestazioni sparse piuttosto che tutte uguali (mediamente un giocatore non è consistente).

Non sono stati fatti vedere grafici post normalizzazione, poichè penso che siano più esplicativi i grafici pre normalizzazione con valori più naturali.

## Capitolo 2 - Apprendimento Supervisionato

L' apprendimento supervisionato è una tecnica di apprendimento automatico che consiste nel fornire al modello un insieme di dati etichettati, ovvero dati per i quali è già nota la risposta, e far apprendere al modello la relazione tra features e labels.

## Approccio

Nel nostro caso, Siamo di fronte a un problema di classificazione binaria, in quanto vogliamo in base alle statistiche di un giocatore predire se la partita è vinta o persa.

Per affrontare il problema ho deciso di utilizzare il seguente approccio:

- **Suddivisione dei Dati:** Che tecnica ho usato per suddividere i dati in training e test set.
- **(opzionale) SMOTE:** ho deciso di confrontare il dataset con e senza SMOTE, la differenza tra le classi è molto piccola quindi non dovrebbe influire molto, se il dataset fosse stato molto più grande e avesse conservato la distribuzione delle vittorie e sconfitte non avrei usato SMOTE, dato l'elevato costo computazionale.
- **Scelta dei Modelli:** ho scelto di utilizzare cinque modelli differenti per la classificazione: `DT`, `Random Forest`, `Logistic Regression`, `SVM`, `ANN`.
- **Tuning degli Iperparametri:** ho utilizzato la tecnica di **GridSearch** per trovare i migliori iperparametri per i modelli.
- **Addestramento & Test dei Modelli:** ho addestrato i modelli sui dati di training.
- **Valutazione dei Modelli:** ho valutato i modelli in base a diverse metriche.
- **Confronto dei Modelli:** ho confrontato i modelli tra loro.
- **Conclusioni**

## Suddivisione dei Dati

La  $k$ -fold cross validation è una tecnica che consiste nel dividere il dataset in  $k$  parti, addestrare il modello su  $k-1$  parti e testarlo sulla parte rimanente, ripetendo l'operazione  $k$  volte.

ho deciso di usare una tecnica di  $k$  - fold cross validation con  $k = 10$  per una iniziale fase di tuning di iperparametri su un tuning set.

Successivamente ho addestrato i modelli come specificato nella fase di addestramento e test con una seconda  $k$  - fold cross validation con  $k = 10$  sul test set.

**Motivazione:** il dataset è molto piccolo ho voluto aumentare il  $k$  a 10. Ho usato  $k = 5$  **thumb rule** ma i risultati erano sensibilmente peggiori. Aumentare  $k$  vuol dire porre enfasi sulla fase di training.

## Scelta dei Modelli

Ho scelto di utilizzare i seguenti modelli per la classificazione:

- Usati perchè solidi con i problemi di classificazione binaria.
  - **Support Vector Machine**: è un modello che cerca di trovare il miglior iperpiano che separa i dati in due classi.
  - **Logistic Regression**: è un modello di classificazione che si basa su una funzione logistica.
- Usati perchè si vuole vedere se ci sono differenze significative tra i modelli.
  - **Albero Decisionale**: è un modello che costruisce un albero di decisione in base alle features.
  - **Random Forest**: è un modello di classificazione che si basa su un insieme di alberi decisionali è usato perchè molte volte gli alberi decisionali sono molto sensibili ai dati e posso andare in overfitting.
  - **Rete Neurale**: è un modello di classificazione che si basa su un insieme di neuroni e layer.

## Motivazioni e pareri personali

A priori non si può sapere quale modello sarà il migliore per il nostro problema, però voglio comunque fare delle considerazioni personali:

- **SVM** e **Logistic Regression** sono modelli nei quali ripongo più fiducia sulle performance in quanto sono ideati per affrontare questi problemi e tendenzialmente richiedono uno sforzo computazionale minore rispetto ad altri approcci.

Invece:

- **DT** è molto sensibile ai dati e può presentare overfitting.
- **RF** è un modello composto da più alberi decisionali per mitigare l' overfitting.
- **DT** e **RF** funzionano molto bene quando le features sono facilmente separabili, ma nel caso di classificazione binaria, soprattutto su delle features di input a valori continui, non credo che siano i modelli migliori.



- ANN non so cosa aspettarmi, è un modello complesso quindi solo il confronto ci dirà se è un modello adatto al nostro problema.

## Tuning degli Iperparametri

Per trovare i migliori iperparametri per i modelli ho utilizzato la tecnica di `GridSearch`, che a differenza di una `random search` o di una `bayesian search`, esplora tutte le combinazioni possibili di iperparametri.

Non è consigliabile utilizzare `Gridsearch` poichè stiamo ponendo un fattore computazionale enorme.

Dato un modello con 3 iperparametri e 3 valori per ogni iperparametro, il numero di combinazioni possibili è  $3^3 = 27$

formalizzabile come:

$$N_{combinazioni} = \prod_{i=1}^n |V_i|$$

Questo fattore va moltiplicato con il tempo di addestramento del modello. Su Datasets molto grandi è un approccio futile, ci possiamo permettere questo sforzo computazionale perchè il dataset è molto piccolo.

### Motivazioni

Nel mio caso, ho scelto di porre più attenzione al confronto generale tra i modelli piuttosto che entrare nel dettaglio di ogni singolo includendo tutti gli iperparametri.

## Addestramento dei Modelli

Ho addestrato i modelli sui dati attraverso la `k-fold cross validation` con  $k = 10$  come descritto sopra.

Nell' addestramento l' unica problematica che ho riscontrato è stato il tempo di addestramento della rete neurale, che con un numero basso di epoche non riusciva a convergere, però alla fine ho risolto aumentandole e trovando un upper bound per il numero di epoche.

**SVM, DT, RT e LR** non hanno avuto grandi costi computazionali.

## Valutazione dei Modelli

Per valutare i modelli ho utilizzato le seguenti metriche:

- una learning curve per vedere se il modello presenta overfitting o underfitting.
  - **Precision**: è la percentuale di predizioni positive fatte dal modello che sono corrette.
  - **Recall**: è la percentuale di predizioni positive corrette fatte dal modello rispetto a tutte le predizioni positive.
  - **F1-Score**: è la media armonica tra precision e recall.
- Invece di usare le curve di apprendimento che valutano la prestazione di una metrica descritta ho usato quelle per l'errore sulla metrica ovvero  $1 - \text{metrica}$  invece di avere un grafico che dovrebbe aumentare al numero di elementi nel test set, con l'errore diventano dei rami di iperbole.

Per ogni modello ho confrontato il dataset con e senza [SMOTE](#) per vedere se ci fossero differenze significative.

## Confronto dei Modelli

Per ogni modello ho strutturato la seguente lista:

- breve descrizione
- hyperparameters
- risultati senza SMOTE
- risultati con SMOTE

ed infine conclusioni personali

## differenza SMOTE e NO SMOTE

```
1 smote is False the outcome distribution is
2 0.0      459
3 1.0      410
4
5 smote is True the outcome distribution is
```

6	0.0	459
7	1.0	459

## Decision Tree

Un decision tree è un modello usato per compiti di classificazione e regressione. È un albero binario, dove i nodi interni rappresentano le decisioni basate sui valori di una feature e le foglie rappresentano le previsioni finali.

### Hyperparameters:

- **resources:**
  - [gini impurity](#)
- **criterion:** Specifica la funzione per valutare la qualità delle suddivisioni.
  - `'gini'`: Imposta l'impurità di Gini come criterio.
  - `'entropy'`: Usa l'informazione di entropia per la suddivisione.
  - `'log_loss'`: Utilizza la perdita logaritmica per problemi di classificazione.
- **max\_depth:** La profondità massima dell'albero.
  - `[5, 10, 15]`: Imponi un upper bound di  $x \in max\_depth$ .
- **min\_samples\_split:** Il numero minimo di campioni richiesti per dividere un nodo.
  - `[2, 5, 10]`: Valori più alti riducono il rischio di overfitting, ma aumentano il rischio di underfitting.

```
1 NO SMOTE
2
3 Best params for decision_tree:
4 {
5   'criterion': 'log_loss',
6   'max_depth': 10,
7   'min_samples_split': 10
8 }
```

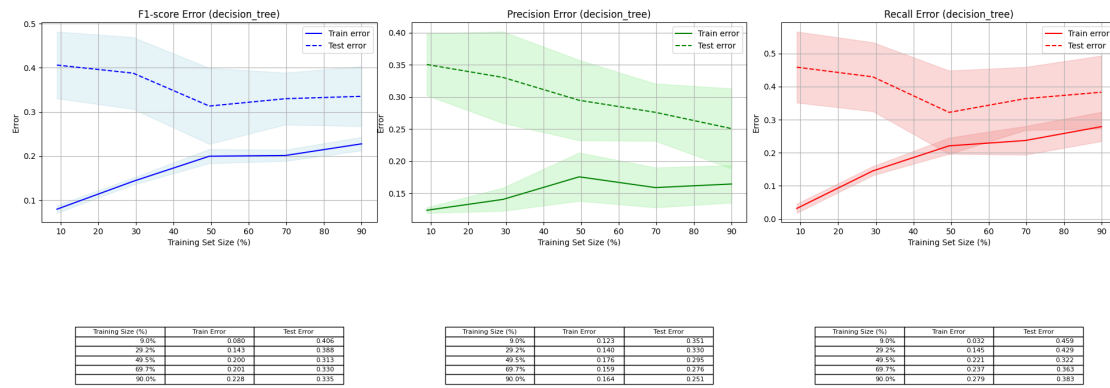


Figure 9: DT\_NOSMOTE

```

1 SMOTE
2
3 Best params for decision_tree:
4 {
5     'criterion': 'gini',
6     'max_depth': 5,
7     'min_samples_split': 2
8 }

```

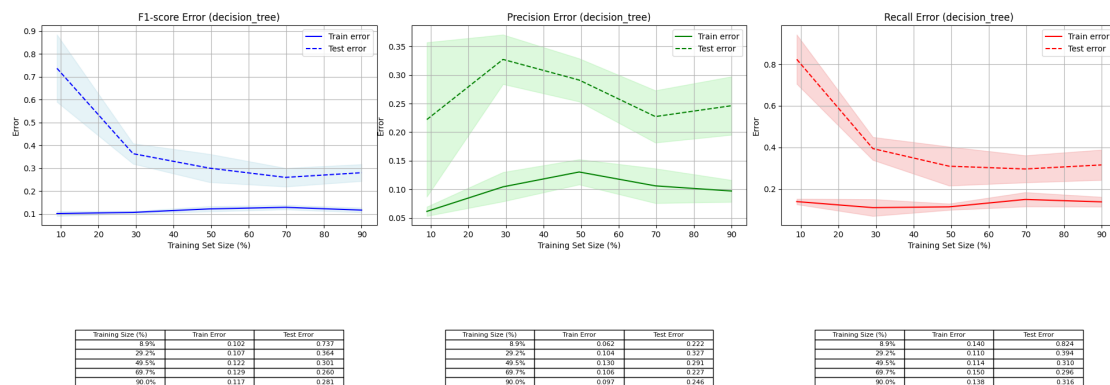


Figure 10: DT\_SMOTE

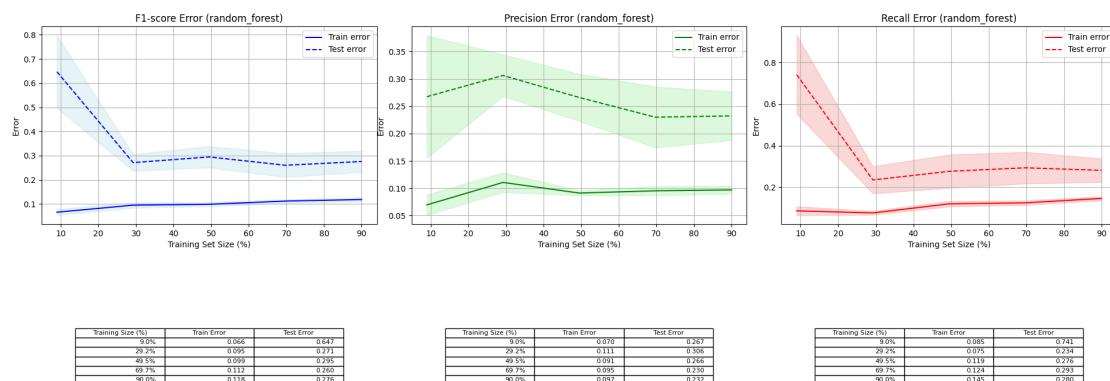
## Random Forest

- **n\_estimators**: Numero di alberi nella foresta.
  - [25, 50, 100]
- **max\_depth**: La profondità massima di ogni 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.

```

1 NO SMOTE
2
3 Best params for random_forest:
4 {
5     'max_depth': 10,
6     'min_samples_split': 10,
7     'n_estimators': 100
8 }

```



**Figure 11: RF\_NOSMOTE**

```

1 SMOTE
2
3 Best params for random_forest:
4 {

```

```

5     'max_depth': 10,
6     'min_samples_split': 10,
7     'n_estimators': 50
8 }

```

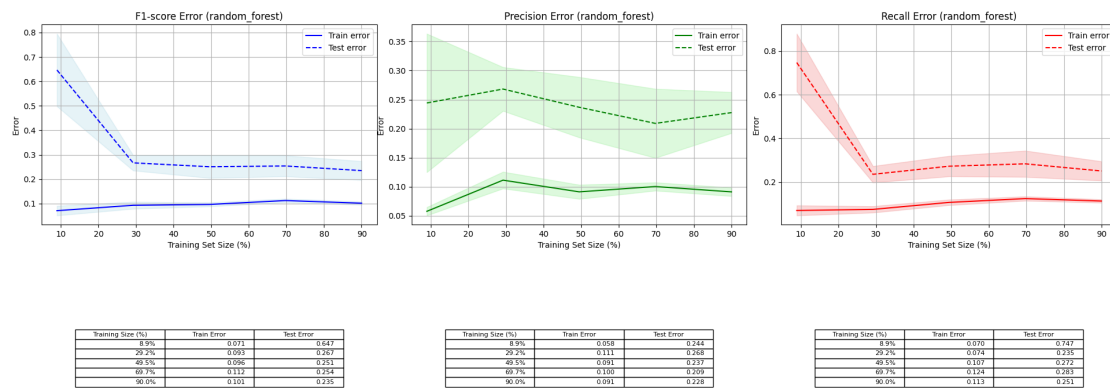


Figure 12: RF\_SMOTE

## Logistic Regression

La Logistic Regression è un modello utilizzato principalmente per problemi di classificazione binaria, usa una squashed function chiamata sigmoide per mappare i valori da uno spazio multi-dimensionale in un intervallo  $[0, 1]$  in  $\mathbb{R}$ .

- **resources:**
  - [logistic regression](#)
- **penalty:** Specifica il tipo di regolarizzazione applicata.
  - la funzione obbiettivo con il parametro  $l2$  diventa:

$$l2 = \text{Minimizzare: } \frac{1}{n} \sum_{i=1}^n \log\text{-loss}(y_i, \hat{y}_i) + \frac{\lambda}{2} ||w||_2^2$$

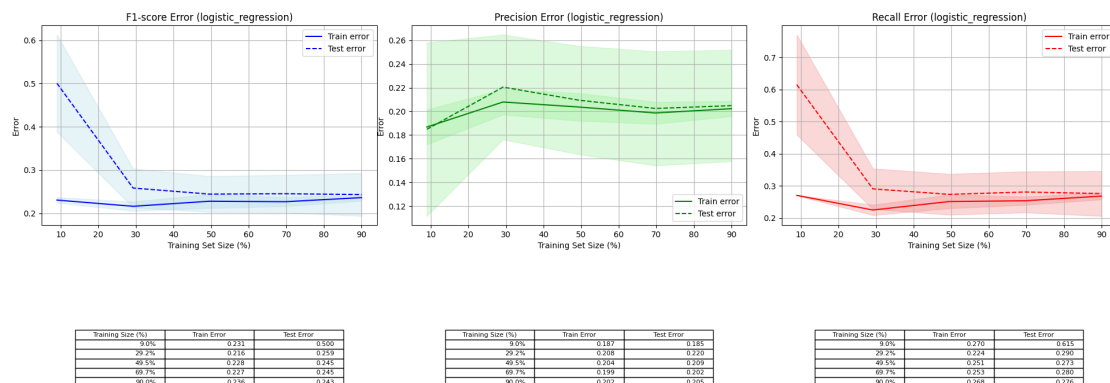
- **C:** Penalizzatore.
  - $[0.001, 0.01, 0.1, 1, 10]$ :

- **solver**: Algoritmo per l'ottimizzazione.
  - **'liblinear'**: coordinate descent
  - **'saga'**: Algoritmo basato sul gradiente stocastico, ma con una variante che sfrutta gradienti medi per velocizzare la convergenza che supporta regolarizzazione  $l_1$ ,  $l_2$ .
- **max\_iter**: Numero massimo di iterazioni per la convergenza.
  - [100000, 150000]

```

1 NO SMOTE
2
3 Best params for logistic_regression:
4 {
5     'C': 10,
6     'max_iter': 100000,
7     'penalty': 'l2',
8     'solver': 'saga'
9 }

```



**Figure 13:** LR\_NOSMOTE

```

1 SMOTE
2
3 Best params for logistic_regression:
4 {

```

```

5     'C': 10,
6     'max_iter': 100000,
7     'penalty': 'l2',
8     'solver': 'saga'
9 }

```

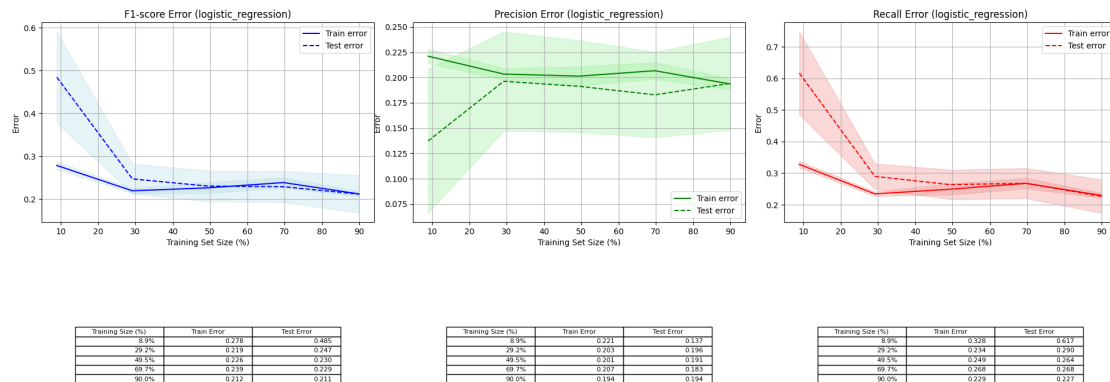


Figure 14: LR\_SMOTE

## Support Vector Machine (SVM)

Le SVM sono una delle prime tecniche di apprendimento supervisionato, cercano di trovare l'iperpiano che massimizza il margine tra le classi.

- **resources:**
  - [svm](#)
  - [kernel trick](#)
  - [What is the purpose of the C parameter in SVM?](#)
- **C:** Parametro di regolarizzazione.
  - $[0.1, 0.5, 1, 2]$ : Controlla il bilanciamento tra massimizzazione del margine e minimizzazione dell'errore.
- **kernel:** Tipo di funzione kernel utilizzata per mappare i dati in uno spazio ad alta dimensione.



- `'linear'`: Usa un kernel lineare (nessuna trasformazione).
- `'rbf'`: Kernel a base radiale, ideale per dati non lineari.
- `'poly'`: Kernel polinomiale.
- `'sigmoid'`: Funzione sigmoide come kernel.
- **gamma**: Coefficiente del kernel RBF, polinomiale o sigmoide.
  - `'scale'`: Basato sull'inverso della somma delle varianze delle feature.
  - `'auto'`: Usa l'inverso del numero di feature.

```

1 NO SMOTE
2
3 Best params for svm:
4 {
5     'C': 2,
6     'gamma':
7     'scale',
8     'kernel': 'linear'
9 }

```

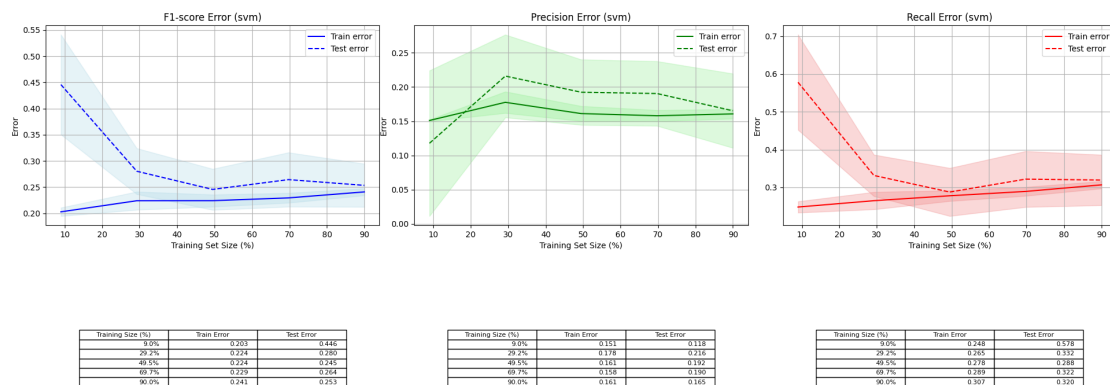


Figure 15: SVM\_NOSMOTE

```

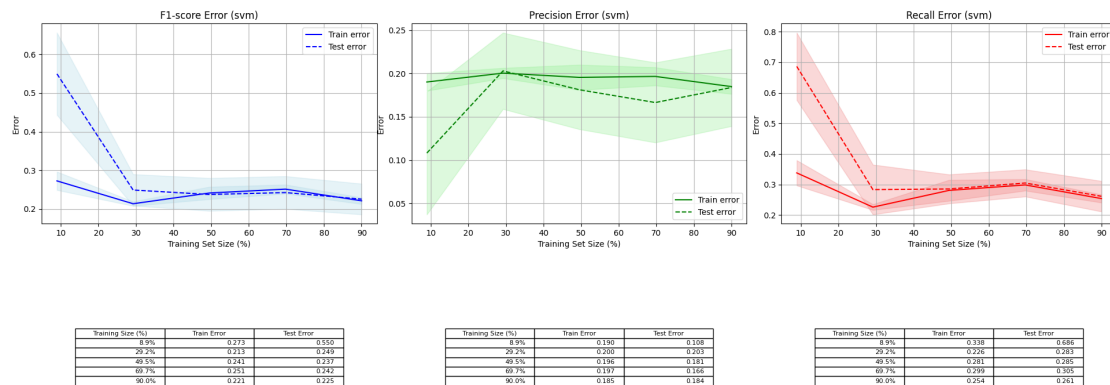
1 SMOTE
2
3 Best params for svm:
4 {
5     'C': 2,

```

```

6     'gamma': 'scale',
7     'kernel': 'poly'
8 }

```



**Figure 16:** SVM\_SMOTE

## Artificial Neural Network (ANN)

Le **Artificial Neural Networks (ANN)** sono modelli di apprendimento ispirati al funzionamento del cervello umano. Sono costituite da **neuroni artificiali** organizzati in strati:

### 1. Strato di input:

- Riceve i dati delle features.

### 2. Strati nascosti:

- Qui avviene l'elaborazione. Ogni neurone prende segnali dagli altri, li combina, li modifica con una funzione matematica (detta **funzione di attivazione**) e passa il risultato al livello successivo.

### 3. Strato di output:

- Restituisce il risultato finale (es. una classificazione o un valore numerico).

**Funzionamento:**

- Ogni connessione tra i neuroni ha un peso che determina l'importanza del segnale.
- Il modello impara aggiustando i pesi durante l'allenamento per migliorare la precisione delle predizioni.
- Il processo di calibrazione dei pesi si chiama *backpropagation*, e usa un metodo di ottimizzazione come il **gradient descent** o **adam**.
- **resources**:
  - ANN
  - adam & sgd
- **hidden\_layer\_sizes**: Configurazione delle dimensioni dei layer nascosti.
  - [(20, ), (40, ), (20, 10), (40, 20)]: Specifica il numero di nodi per ciascun layer.
- **activation**: Funzione di attivazione per i layer nascosti.
  - 'logistic': Funzione sigmoide.
  - 'relu': Funzione Rectified Linear Unit, molto comune.
- **solver**: Ottimizzatori
  - 'sgd': Discesa del gradiente stocastico.
  - 'adam': **non ho compreso molto bene come funziona l'ottimizzatore ma l'ho messo poichè ho trovato che è uno dei migliori**
- **alpha**: Parametro di regolarizzazione L2.
  - [0.0001, 0.05]: Valori più alti aumentano la penalizzazione per evitare overfitting.
- **learning\_rate**: Politica di aggiornamento del learning rate.
  - 'constant': Tasso di apprendimento fisso.
  - 'adaptive': Il tasso diminuisce se le prestazioni del modello non migliorano.
- **max\_iter**: Numero massimo di iterazioni per l'allenamento.

- [2000]: Più iterazioni permettono di affinare il modello, ma aumentano il costo computazionale.

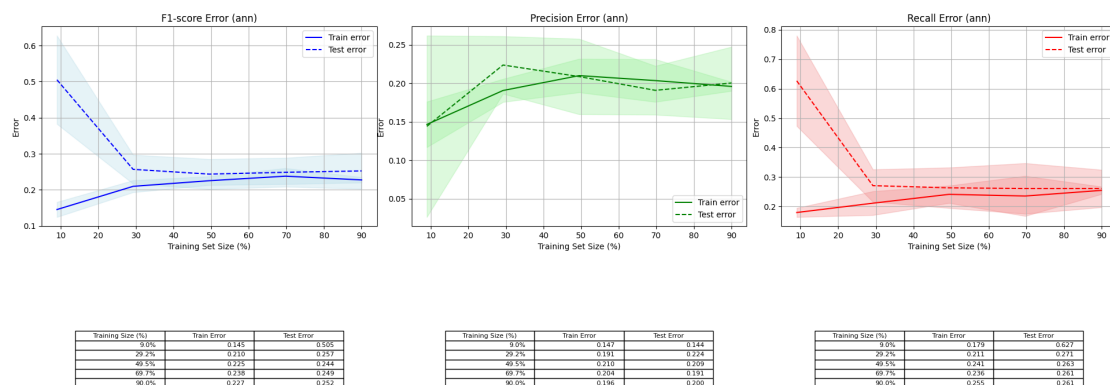
• **Note:**

- ho provato più valori di `max_iter` partendo da 300 fino a 3000, ho notato che il modello convergeva mediamente a 2000 iterazioni.

```

1 NO SMOTE
2
3 Best params for ann:
4 {
5     'activation': 'relu',
6     'alpha': 0.0001,
7     'hidden_layer_sizes': (40, 20),
8     'learning_rate': 'constant',
9     'max_iter': 2000,
10    'solver': 'adam'
11 }

```



**Figure 17:** ANN\_NOSMOTE

```

1 SMOTE
2
3 Best params for ann:
4 {
5     'activation': 'relu',
6     'alpha': 0.05, 'hidden_layer_sizes': (40, 20),

```

```

7     'learning_rate': 'constant',
8     'max_iter': 2000,
9     'solver': 'adam'
10 }

```

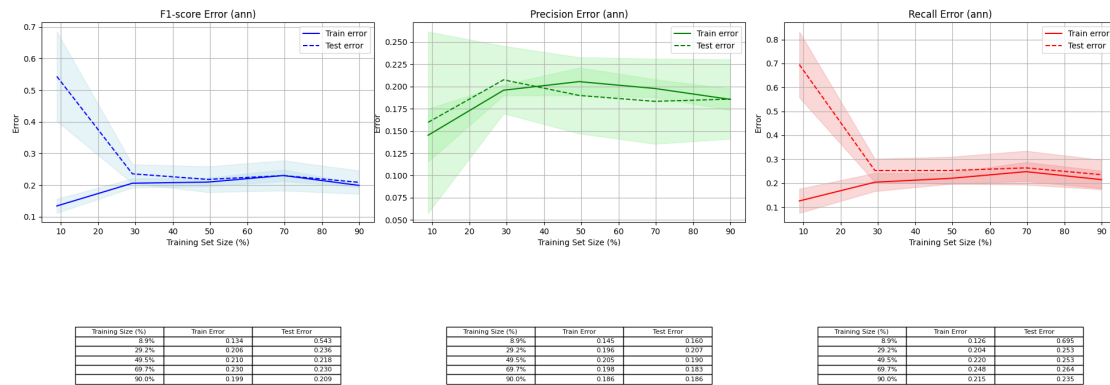


Figure 18: ANN\_SMOTE

## Conclusioni

Come si può vedere dai grafici, **Decision tree** e **Random forest** non danno risultati significativi e indipendentemente dal dataset senza **SMOTE** presentano overfitting.

**Logistic regression**, **SVM** e **ANN** presentano risultati simili fra di loro indipendentemente dall'applicazione di **SMOTE**, con **ANN** che è leggermente migliore, ma c'è da considerare che il dataset è molto piccolo e ci possiamo permettere questo sforzo computazionale nell'eventualità di un dataset più grande i migliori modelli sarebbero stati **SVM** e **Logistic Regression**.

## Capitolo 3 - Apprendimento della Struttura

Il ragionamento probabilistico si basa sulla teoria delle probabilità per fare inferenze sui dati.

L'obiettivo del capitolo è quello di apprendere la struttura di un modello probabilistico basato sui dati, ovvero trovare le relazioni tra le variabili e le loro probabilità condizionate.

## Motivazione

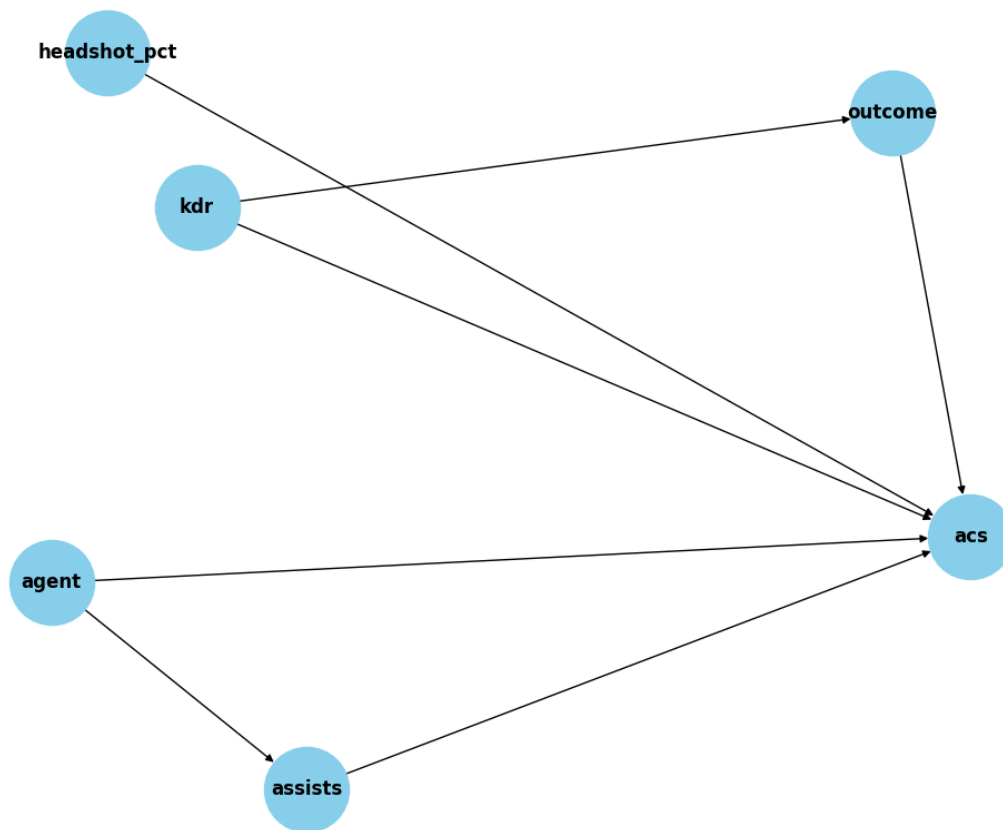
Possiamo comprendere meglio le relazioni tra le variabili, capire quali variabili influenzano di più l'outcome e quali sono meno rilevanti. È molto utile poiché nell'eventualità di estensione del progetto possiamo usare il modello come base per dare dei pesi alle statistiche del player, per poi ricavare una stima di valutazione.

## Approccio

### hill climb search

Per affrontare il problema ho deciso di usare l'algoritmo di ricerca Hill Climb, che è un algoritmo di ricerca locale per massimizzare la funzione obiettivo che nel nostro caso è il **K2 score**.

Ho usato il **K2score**, perché sono riuscito a ottenere un risultato migliore rispetto al metodo di score **BIC**, **BIC** mi forniva un DAG con 2 nodi.



**Figure 19:** bayesian\_network

## Pareri e conclusioni

La struttura del modello è molto interessante, possiamo vedere che:

- l'`acs` che è un indicatore medio è l'unica foglia, il suo metodo di calcolo è sconosciuto poichè proprietario di Riot Games, ha molto senso che sia l'unica foglia, poichè rappresenta un indice medio di valutazione.
- l'`outcome` dipende principalmente dal `kdr`, logicamente ha senso poichè, se un giocatore ha un `kdr` alto vuol dire che ha eliminato più nemici di quante volte sia morto, il che implica che le probabilità di vittoria aumentano.
- Il `personaggio` influenza gli `assists` è perfetto, poichè un personaggio che ha funzioni di supporto aiuta i compagni a eliminare i nemici, uno che ha funzioni

di attacco elimina i nemici in maniera avida (senza aiutare i compagni).

- l' `headshot_pct` è l'unico valore che influenza solo l'acs, la motivazione è che i colpi alla testa non danno informazioni su quante volte un giocatore ha eliminato un nemico o quanti assists ha fatto, un giocatore porterebbe avere 70% di headshot\_pct, ma avere poche kills o viceversa poca `headshot_pct`, ma molte kills si pensa a un fucile a pompa e si immagina la rosa è facile comprendere che mediamente non si fanno molti headshot.
- tutte le altre relazioni influenzano l'acs che come abbiamo detto ha molto senso poichè è un indice medio di valutazione.

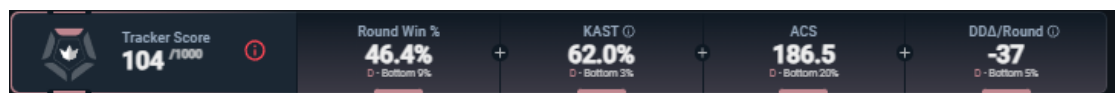
Ho deciso di generare degli esempi per vedere se il modello è coerente con la realtà.

1		agent	outcome	kdr	acs	assists	headshot_pct
2	0	1	1	1.9	298	4	29
3	1	0	0	0.9	98	2	17
4	2	1	1	1.3	163	3	25
5	3	0	0	0.6	69	4	40

**Nota:** Si veda come nel campione generato, la relazione tra kdr e outcome.

## Curiosità

L' immagine mostra delle statistiche di un altro giocatore con rank simile al giocatore del dataset si veda come l' ACS medio del giocatore è vicino a quelli del campione generato.



**Figure 20:** account\_player2

Non ho testato il modello in larga scala poichè sono interessato solo alle relazioni fra le variabili, però facendo previsioni sullo stesso player da cui proviene la foto (stesso rank e stessi agenti utilizzati) ho ottenuto un risultato eccellente.

**La prima partita è vinta la seconda e la terza invece sono perse.**









 13:8 7th	 536	K/D/A 14/15/6	K/D 0.9	DDΔ +10	HS% 19	ADR 139	ACS 204
 11:13 4th	 500	K/D/A 15/20/9	K/D 0.8	DDΔ -29	HS% 29	ADR 131	ACS 197
 5:13 10th	 100	K/D/A 6/17/7	K/D 0.4	DDΔ -91	HS% 22	ADR 70	ACS 105

Figure 21: predicted\_proba

```

1 data = pd.DataFrame({
2     'agent': [1,0,1],
3     'kdr': [0.9,0.8,0.4],
4     'headshot_pct': [19,29,22],
5 })
6
7 probability_agent = model.predict(data)
8 print("Predicted probabilities:\n", probability_agent)

```

Predicted probabilities:			
	outcome	acs	assists
0	0	228	2
1	0	197	2
2	0	59	2

Figure 22: 3games\_player2

In conclusione gli assist non vengono predetti correttamente, ma il modello è molto accurato sia nel predire l'outcome che l'acs.

Questa parte del progetto è molto interessante, poichè si vuol far notare come si possano traslare le statistiche di un player su un altro, questa intuizione deve essere verificata su larga scala, però intuitivamente come descritto prima il rank non influenza le prestazioni di un giocatore, se i giocatori competono con lo stesso livello di abilità (rank), le statistiche nella media saranno simili ergo si può creare un modello che predica mediamente

le statistiche di un gruppo di giocatori con lo stesso livello di abilità e usarlo come base per valutare le prestazioni di un giocatore.

## Capitolo 4 - Knowledge Base

### prolog

Possiamo, perciò concludere il progetto usando una knowledge base per inferire conoscenza dai dati usando il prolog un linguaggio di programmazione logica.

Usiamo l'osservazione fatta nel capitolo 3 dove l' ACS è un valore che viene influenzato da tutte le altre statistiche per creare delle relazioni ad-hoc.

Potrei unire le predizioni fatte dai modelli nel capitolo 2, con le informazioni ottenute e ragionando su di esse mettendo a confronto valutazioni e predizioni però non ho abbastanza dati per fare un confronto significativo.

### Costruzione della knowledge base

Vogliamo ragionare su una partita di Valorant, usare la knowledge base per fare inferenze sulle statistiche di una partita.

Ci poniamo quindi il seguente obiettivo:

- Data una base di conoscenza con fatti relativi alle statistiche di una partita, e regole che descrivono relazioni tra statistiche, vogliamo fornire una valutazione al giocatore.

### Fatti e regole

In questo esempio, la base di conoscenza rappresenta delle partite di gioco (presumibilmente per un gioco come *Valorant*). Ogni partita è associata a vari attributi come `game_id`, `agent`, `kdr`, `assists`, `headshot_pct`, `acs` e `outcome`.

## Fatti:

Un **fatto** in Prolog è una dichiarazione di qualcosa che è vero. In questo caso, un fatto è una rappresentazione di una partita specifica nel dataset. Ogni partita è rappresentata come un fatto nel formato:

```
1 partita(game_id, agent, kdr, assists, headshot_pct, acs, outcome) .
```

Ecco come vengono costruiti i fatti nel codice:

1. **game\_id**: Un identificatore univoco per la partita, ad esempio `g1`.
2. **agent**: L'agente o il personaggio utilizzato nella partita (ad esempio, "Chyper").
3. **kdr**: Il KDR (Kill-Death Ratio) della partita, che rappresenta il numero di uccisioni rispetto al numero di morti.
4. **assists**: Il numero di assist effettuati nella partita.
5. **headshot\_pct**: La percentuale di colpi alla testa.
6. **acs**: L'ACS (Average Combat Score) della partita, un punteggio che misura l'efficacia di un giocatore in combattimento.
7. **outcome**: Il risultato della partita, che può essere "win" (vittoria) o "loss" (sconfitta).

Un esempio di fatto potrebbe essere:

```
1 partita(g1, 'Chyper', 1.2, 5, 30, 250, 'win') .
```

Nella partita identificata da `g1` giocata con l'agente *Chyper*, il giocatore ha avuto un KDR di 1.2, ha effettuato 5 assist, ha avuto il 30% di colpi alla testa, ha ottenuto un ACS di 250 e 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 essere vera affinché una conclusione (la parte dopo `: -`) sia vera.

1. **Regola `prec(X, Y)`**: La regola `prec(X, Y)` afferma che la partita `X` è precedente alla partita `Y` se il `game_id` di `X` è inferiore a quello di `Y`.

```

1 prec(X, Y) :-
2     partita(X, _, _, _, _, _),
3     partita(Y, _, _, _, _, _),
4     sub_atom(X, 1, _, 0, ID_X),
5     sub_atom(Y, 1, _, 0, ID_Y),
6     atom_number(ID_X, Num_X),
7     atom_number(ID_Y, Num_Y),
8     Num_X < Num_Y.

```

Questo significa che se esistono due partite `X` e `Y`, e se l'ID del gioco `X` è numericamente minore di quello di `Y`, allora `X` è considerata precedente a `Y`.

2. **Regola `better(X, Y)`**: La regola `better(X, Y)` afferma che la partita `X` è migliore della partita `Y` se l'ACS di `X` è maggiore di quello di `Y`.

```

1 better(X, Y) :-
2     partita(X, _, _, _, _, ACS_X, _),
3     partita(Y, _, _, _, _, ACS_Y, _),
4     ACS_X > ACS_Y.

```

In altre parole, se l'ACS di `X` è superiore a quello di `Y`, allora `X` è considerata “migliore” di `Y`.

3. **Regola `not_newbie(X)`**: La regola `not_newbie(X)` afferma che il giocatore non è più inesperto nel gioco se ha fatto più di 50 partite.

```

1 not_newbie(X) :-
2     partita(X, _, _, _, _, _, _),
3     sub_atom(X, 1, _, 0, ID_X),
4     atom_number(ID_X, Num_X),
5     Num_X > 50.

```

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

4. **Regole per condizioni specifiche (`high_kdr`, `high_headshot`, `high_assists`)**:

- **Regola `high_kdr(X)`**: Determina se una partita ha un KDR maggiore di 1.5.

- `prolog high_kdr(X) :- partita(X, _, KDR, _, _, _, _), KDR > 1.5.`
- **Regola `high_headshot(X)`**: Determina se una partita ha una percentuale di colpi alla testa maggiore di 25.
- `prolog high_headshot(X) :- partita(X, _, _, _, Headshot_Pct, _, _), Headshot_Pct > 25.`
- **Regola `high_assists(X)`**: Determina se una partita ha un numero di assist maggiore di 5.
- `prolog high_assists(X) :- partita(X, _, _, Assists, _, _, _), Assists > 5.`

5. **Regola `astonishing(X)`**: La regola `astonishing(X)` determina se una partita è “straordinaria” (`astonishing`). Una partita è considerata straordinaria se:

- Non è una delle prime partite giocate nella carriera del giocatore `not_newie(X)`.
- Ha almeno una delle seguenti condizioni: un KDR alto, una percentuale di colpi alla testa alta o un numero elevato di assist.
- Ha una percentuale di partite precedenti peggiori di almeno il 75%.

```

1 astonishing(X) :-
2     not_newbie(X), (high_kdr(X); high_headshot(X);
3         high_assists(X)),
4     findall(Y, (prec(Y, X), better(X, Y)), BetterGames),
5     length(BetterGames, CountBetter),
6     findall(Y, prec(Y, X), PreviousGames),
7     length(PreviousGames, CountPrevious),
8     CountPrevious > 0,
9     Ratio is CountBetter / CountPrevious,
10    Ratio >= 0.75.
```

In questa regola, `findall/3` raccoglie tutte le partite precedenti a `X` che sono peggiori, e la partita è considerata straordinaria se la percentuale di tali partite peggiori è superiore o uguale al 75%.

6. **Regole `normal(X)` e `bad(X)`**: Le regole `normal(X)` e `bad(X)` determinano se una partita è considerata normale o negativa (rispettivamente). Funzionano in modo simile a `astonishing(X)`, ma con diverse soglie:

- **Normal:** Se la partita è migliore del 25% ma inferiore al 75% delle partite precedenti.
- **Bad:** Se la partita è migliore di meno del 25% delle partite precedenti.

```
1 normal(X) :-
2   not_newbie(X), findall(Y, (prec(Y, X), better(X, Y)),
3     BetterGames),
4   length(BetterGames, CountBetter),
5   findall(Y, prec(Y, X), PreviousGames),
6   length(PreviousGames, CountPrevious),
7   CountPrevious > 0,
8   Ratio is CountBetter / CountPrevious,
9   Ratio >= 0.25, Ratio < 0.75.
10
11 bad(X) :-
12   not_newbie(X), findall(Y, (prec(Y, X), better(X, Y)),
13     BetterGames),
14   length(BetterGames, CountBetter),
15   findall(Y, prec(Y, X), PreviousGames),
16   length(PreviousGames, CountPrevious),
17   CountPrevious > 0,
18   Ratio is CountBetter / CountPrevious,
19   Ratio < 0.25.
```

- I **fatti** descrivono i dettagli delle partite, come il `game_id`, l'agente, il KDR, e altri attributi specifici della partita.
- Le **regole** utilizzano i fatti per inferire nuove informazioni, come determinare se una partita è “straordinaria” o “normale” in base ai valori delle partite precedenti, e confrontando il KDR, gli assist, o le percentuali di headshot.

## Esempi di query

Si riportano 3 partite per valutazione (elenco non esaustivo) si noti come le statistiche della partita influenzano la valutazione.

## Astonishing Games

Game ID	Agent	KDR	Assists	Headshot%	ACS	Outcome
62	Cypher	1.5	6	5	258	Win
64	Cypher	1.5	8	7	248	Win
69	Cypher	1.2	6	8	238	Win

### Normal Games

Game ID	Agent	KDR	Assists	Headshot%	ACS	Outcome
51	Cypher	1.3	9	10	211	Win
52	Cypher	0.8	6	8	172	Loss
53	Cypher	0.9	8	9	212	Win

### Bad Games

Game ID	Agent	KDR	Assists	Headshot%	ACS	Outcome
54	Cypher	0.6	1	20	115	Loss
58	Cypher	0.6	3	12	132	Loss
65	Cypher	0.7	5	12	89	Win

### Possibili Upgrade

Questa knowledge base è una demo, può essere utile per valutare le prestazioni di un giocatore, ma la piena potenzialità si può raggiungere affiancandola da dei modelli che fanno predizioni sulle statistiche di un giocatore e usare la knowledge base per valutare le prestazioni.

## Capitolo 5 - Conclusioni

### Risultati ottenuti

Posso concludere di aver ottenuto dei risultati ottimi, sono rimasto molto soddisfatto dell'apprendimento della struttura, poichè ho condiviso questa informazione su un server della community di Valorant e molti hanno concordato sulla struttura ritenendola coerente.

Bisogna ammettere che non mi aspettavo la dipendenza del `kdr` sull' `outcome` e l' `acs` su tutte le altre variabili, ma dopo le eventuali riflessioni abbiamo motivazioni valide per queste relazioni.

Riot mette a disposizione la possibilità di contattare i developer presentando il progetto e l' idea di un possibile sviluppo futuro sarebbe fattibile.

### Possibili Sviluppi Futuri

La knowledge base in prolog si può ampliare con nuove regole e fatti in tal modo che a ogni round si può fare inferenze sulle statistiche di un giocatore e predire l' esito della partita a lunga andata, avvisando il giocatore se sta avendo prestazioni scadenti o viceversa.

Si potrebbe risalire a come l' `acs` viene calcolato apprendendo la struttura su dataset più ampi risalendo dalla probabilità condizionata di `acs` a un vettore di peso sulle altre statistiche.