# CLASSIFICAZIONE INTELLIGENTE DEI FUNGHI

ANNO ACCADEMICO 2024/2025

Componenti del gruppo:

- Christian Durante, 774759, c.durante8@studenti.uniba.it

Link repository del progetto:

- https://github.com/ChristianChR/Ingegneria-della-conoscenza

# INDICE

# INTRODUZIONE DEL PROGETTO

Questo progetto ha come obiettivo quello di riuscire a distinguere quando un fungo è velenoso e quando è edibile.

Per fare ciò ho scelto di utilizzare un dataset del 1987 della UCI Machine Learning:

https://www.kaggle.com/datasets/uciml/mushroom-classification

Il dataset è composto da:

- 8124 campioni ipotetici di funghi corrispondenti a 23 specie di funghi
- ogni specie è identificata come sicuramente commestibile (denotato con l'etichetta e), sicuramente velenosa o di commestibilità sconosciuta e non raccomandata, quest'ultima classe è stata combinata con quella velenosa (denotate con l'etichetta p)
- varie caratteristiche di ogni fungo consultabili sul link precedentemente allegato

Come modelli di apprendimento supervisionato ho scelto:

- Random Forest, che è un modello basato su un insieme di alberi decisionali che utilizza il bagging per migliorare la precisione e ridurre l'overfitting
- Support Vector Classifier, che cerca di trovare un iperpiano che separa le classi nel miglior modo possibile
- Decision Tree Classifier, che utilizza un albero decisionale per prendere decisioni basate sui valori delle feature
- Bernoulli Naive Bayes, un classificatore probabilistico basato sul teorema di Bayes, progettato per gestire dati con caratteristiche booleane o binarie

# PREPROCESSING DEI DATI

## One Hot Encoding e test Chi-quadro

Prima di utilizzare i dati poiché le feature si presentavano sotto forma categorica (ad esempio cap-shape: bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s) ho scelto di utilizzare il One Hot Encoding sul dataset per trasformare le feature da valori categorici a valori binari (True e False), e salvarle in mushrooms_encoded.csv.

Per la scelta delle feature più rilevanti è stata utilizzato il test del Chi-quadro, è stato scelto perché è utile per misurare la dipendenza tra le variabili e la classe target (che nel nostro caso è la classe p).

Il test del Chi-quadro restituisce due valori:

- Il $Chi^2$ score che misura quanto la distribuzione delle frequenze osservate per una feature si discosti da quelle attese se non ci fosse alcuna relazione con la classe target. Maggiore è il punteggio, più significativa è la relazione tra la feature e la classificazione del fungo.
- Il p-value, che indica la probabilità che la relazione osservata tra la feature e la variabile target sia dovuta al caso. Un valore di p inferiore a 0.05 suggerisce che la feature ha un impatto significativo sulla classificazione e non è semplicemente frutto di fluttuazioni casuali nei dati.

I punteggi relativi al test sono stati calcolati e memorizzati nel file chi2_result.txt attraverso lo script select_chi.py.

## Creazione della knowledge base e definizione di regole

Ho deciso di selezionare le 20 feature più rilevanti risultate dal test, da qui ho eseguito lo script generate_kb.py per convertire i funghi memorizzati nel file mushrooms_encoded.csv in fatti per la mia knowledge base scritta in prolog.

```python
import pandas as pd

df = pd.read_csv('data/mushrooms_encoded.csv')

# Lista delle feature più rilevanti basata sul test Chi²
selected_features = [
    "odor_n", "odor_f", "stalk-surface-above-ring_k", "stalk-surface-below-ring_k",
    "gill-color_b", "gill-size_n", "spore-print-color_h", "ring-type_l", "ring-type_p",
    "bruises_t", "spore-print-color_n", "spore-print-color_k", "bruises_f", "gill-spacing_w",
    "population_v", "spore-print-color_w", "gill-size_b", "habitat_p", "stalk-surface-above-ring_s",
    "odor_y"
]

with open('data/mushrooms_kb.pl', 'w') as f:
    f.write('% Definizione dei fatti basati sulle feature selezionate\n')
    for index, row in df.iterrows():
        features = ', '.join([f"{col}_{val}" for col, val in row.items() if col in selected_features])
        f.write(f"mushroom({index + 1}, [{features}]).\n")

    f.write('\n% Regole per determinare se un fungo è velenoso o commestibile\n')
    for feature in selected_features:
        f.write(f'isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member({feature}_True, Features).\n')

    f.write('\n% Un fungo è edibile se non è velenoso\n')
    f.write('isEdible(Mushroom) :- mushroom(Mushroom, Features), \\+ isPoisonous(Mushroom).\n')
```

Di seguito un esempio di fatto (fungo) memorizzato nella KB:

```
mushroom(8064, [bruises_f_True, bruises_t_False, odor_f_False, odor_n_True, odor_y_False, gill-spacing_w_True, gill-si
```

Per creare delle regole, ho deciso di utilizzare le feature rilevanti trovate col test, poiché la classe target è p, le feature selezionate con valore True dovrebbero identificare un fungo velenoso, da qui creo le regole per identificare un fungo

velenoso:

```
% Regole per determinare se un fungo  velenoso o commestibile
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(odor_n_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(odor_f_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(stalk-surface-above-ring_k_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(stalk-surface-below-ring_k_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(gill-color_b_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(gill-size_n_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(spore-print-color_h_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(ring-type_l_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(ring-type_p_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(bruises_t_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(spore-print-color_n_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(spore-print-color_k_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(bruises_f_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(gill-spacing_w_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(population_v_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(spore-print-color_w_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(gill-size_b_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(habitat_p_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(stalk-surface-above-ring_s_True, Features).
isPoisonous(Mushroom) :- mushroom(Mushroom, Features), member(odor_y_True, Features).
```

Mentre un fungo è edibile se non è velenoso, di seguito la regola in prolog:

```
% Un fungo  edibile se non  velenoso
isEdible(Mushroom) :- mushroom(Mushroom, Features), \+ isPoisonous(Mushroom).
```

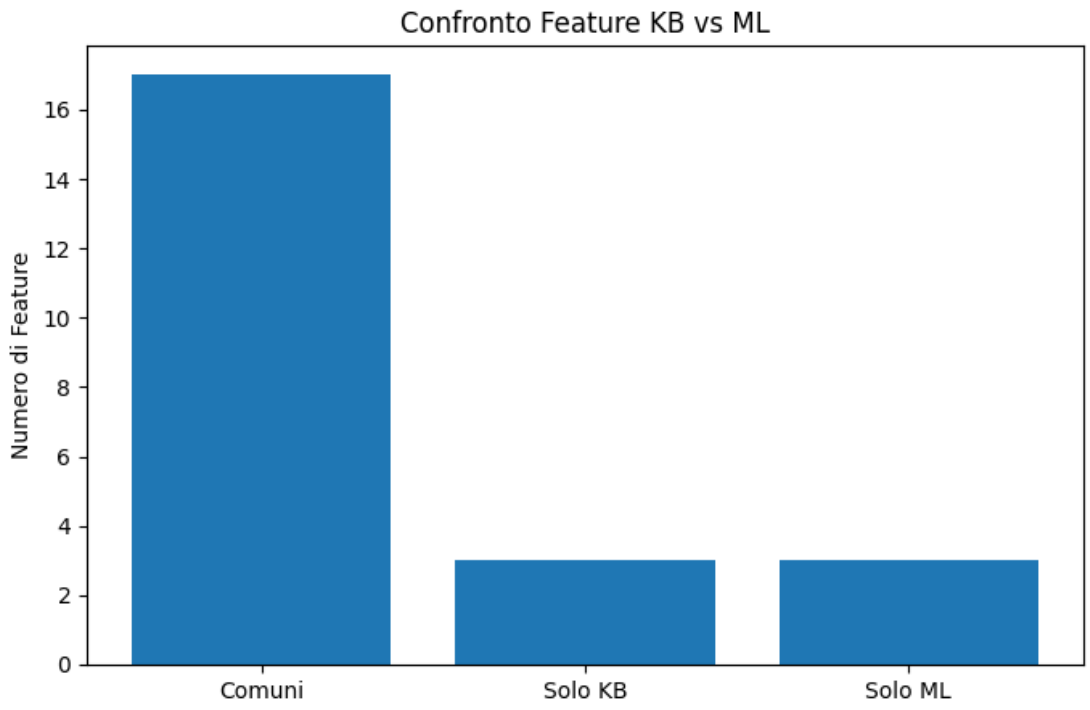
## Selezione delle feature e comparazione con altre metriche

Per verificare la qualità della KB, ho deciso di confrontare le feature selezionate da me con il test chi-quadro con quelle selezionate dal metodo mutal_info_classif della libreria sklearn.feature_selection.

Quindi ho eseguito lo script select_feature.py e feature_comparison.py per innanzitutto salvare in un formato più leggibile per il sistema le feature selezionate col test chi-quadro, nel file important_features_kb.json

Dopodiché ho calcolato il punteggio di informazione mutua tra ogni feature e la variabile target p, e salvato il dataframe di risultati nel file important_features_ml.json, e confrontato i risultati delle due metriche di valutazione. Di seguito le feature comuni e non:


```
"common_features": [
    "spore-print-color_n",
    "ring-type_l",
    "population_v",
    "gill-color_b",
    "gill-size_b",
    "gill-size_n",
    "ring-type_p",
    "stalk-surface-above-ring_s",
    "stalk-surface-above-ring_k",
    "stalk-surface-below-ring_k",
    "spore-print-color_h",
    "spore-print-color_k",
    "bruises_t",
    "gill-spacing_w",
    "odor_n",
    "odor_f",
    "bruises_f"
],
"only_in_kb": [
    "odor_y",
    "spore-print-color_w",
    "habitat_p"
],
"only_in_ml": [
    "class_e",
    "gill-spacing_c",
    "stalk-surface-below-ring_s"
]
```




Il grafico fa notare come entrambe le metriche nella maggior parte dei casi individuano le stesse feature, e ciò conferma la validità della KB, d'altro canto, vi sono tre feature divergenti che potrebbero essere dovute al metodo mutual_info_classif ha individuato pattern non evidenti nella logica del Prolog.

Per le fasi successive ho scelto di salvare le feature comuni trovate da entrambe le metriche nel file mushrooms_common_feature.csv e scartare le 3 feature divergenti per garantire coerenza con la Knowledge Base, evitando possibili incongruenze tra Machine Learning e inferenza logica.

# APPRENDIMENTO SUPERVISIONATO

## Ottimizzazione degli iperparametri

Prima di confrontare i modelli tra loro, decidendo quale avesse le migliori prestazioni, ho ottimizzato gli iperparametri migliori per ciascun modello.

Per il modello Bernoulli Naive Bayes ho deciso di utilizzare i valori predefiniti, in quanto il modello assume indipendenza tra le feature e tende a performare bene senza necessita di ricercare i parametri ottimali.

Gli iperparametri che ho scelto di ottimizzare per ciascun modello sono:

```
"svm": {
    "C": [0.01, 0.001],  # Aumento valori di regolarizzazione
    "kernel": ["linear", "rbf", "poly"]  # Aggiunto kernel polinomiale
},
"random_forest": {
    "n_estimators": [10, 25],  # Aumento numero di alberi
    "max_depth": [1, 2],  # Ridotto ulteriormente max_depth
    "min_samples_split": [3, 5, 10, 15]  # Aumento valori per limitare splitting
},
"decision_tree": {
    "max_depth": [1, 2],  # Ridotto ulteriormente max_depth
    "min_samples_split": [3, 5, 10, 15],  # Aumento valori per limitare splitting
    "criterion": ["gini", "entropy"]  # Aggiunto criterio entropy
}
```

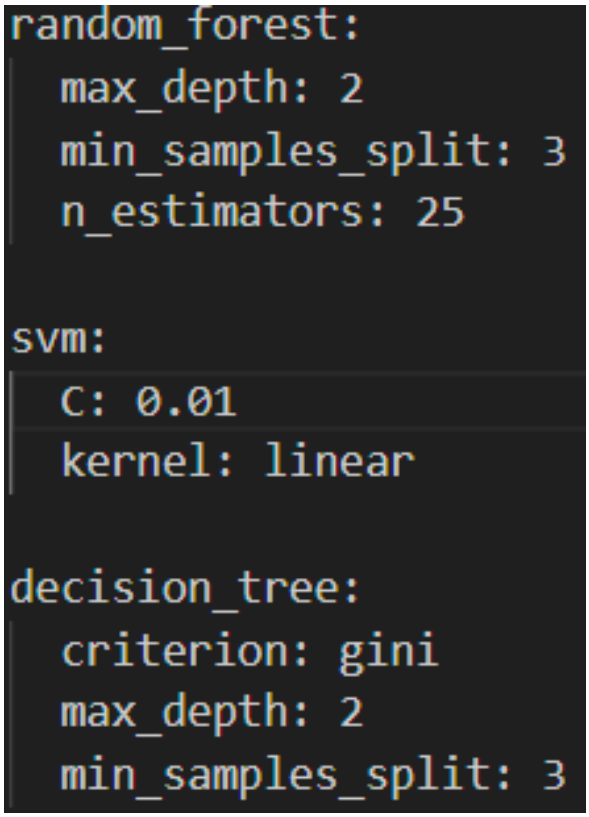
\

- Per il kernel SVM i parametri da ottimizzare sono:
    - C: Parametro di regolarizzazione
    - kernel: Tipo di kernel utilizzato per trasformare i dati
- Random Forest i parametri da ottimizzare sono:
    - n_estimators: numero di alberi nella foresta
    - max_depth: profondità massima degli alberi

    - min_samples_split: numero minimo di campioni richiesti per dividere un nodo
- Decision Tree i parametri da ottimizzare sono:
    - max_depth: profondità massima dell'albero
    - min_samples_split: numero minimo di campioni richiesti per dividere un nodo
    - criterion: funzione di misura della qualità della divisione

L'operazione di ottimizzazione dei parametri è stata eseguita dallo script hyperparameter_tuning.py, per evitare l'overfitting ho deciso di utilizzare la grid search con cross validation in modo tale da valutare il modello su diversi sottoinsiemi di dati, k=10 visto i pochi dati, in modo tale da fornire una stima più robusta delle prestazioni del modello.

Gli iperparametri vengono memorizzati in best_hyperparameters.txt per un utilizzo successivo.



# Implementazione dei modelli

Dallo script learning.py vengono caricati i dati delle feature comuni e la colonna target (class_p) dai dataset mushrooms_common_features.csv e mushrooms_encoded.csv

Dopo viene eseguita una divisione del dataset in Training Set (70%) e Test Set (30%) mantenendo la distribuzione delle classi (stratify = y)

Vengono caricati gli iperparametri dal file apposito creato in precedenza e vengono memorizzati in un dizionario, inizializzando i modelli di Random Forest, SVM e Decision Tree con i propri iperparametri, per il modello Bernoulli Naive Bayes, i valori predefiniti per valutare le prestazioni del modello.

Dopodiché viene creato un oggetto K-fold Cross-Validation con 10 fold per ridurre la possibilità di overfitting ed effettuare una stima più affidabile di ogni modello riducendo la varianza associata alla divisione casuale dei dati, e vengono addestrati i vari modelli.

La tecnica del K-fold con k = 10 suddivide il dataset in 10 sottoinsiemi addestrando il modello su 9 e testandolo sull'ultimo. Il processo viene ripetuto 10 volte, ruotando i sottoinsiemi, e il risultato finale è la media delle 10 valutazioni.

## Confronto dei modelli

Ho deciso di valutare i modelli sul calcolo delle seguenti metriche utilizzando un approccio globale sull'intero dataset, senza distinzione tra le singole classi, in modo da avere delle medie delle performance complessive dei modelli e confrontarli tra loro:

- accuracy: numero di predizioni corrette su quelle totali
- f1-score: metrica che combina precision e recall in un solo valore (per bilanciare
- precision: numero di true positive su true positive + true negative
- recall: numero di true positive su true positive +false negative

Il calcolo di tali metriche è stato eseguito nel modulo learning.py dopo l'addestramento dei modelli (inizializzati con parametri ottimali) e salva tutti i risultati nel file model_metrics.txt.

In futuro potrebbe essere utile calcolare le metriche per singola classe per comprendere meglio le eventuali squilibri nelle predizioni.

Le metriche ottenute per ogni modello sono le seguenti:

```
Model: Random Forest
Accuracy: 0.9783
Precision: 0.9754
Recall: 0.9796
F1 Score: 0.9775

Model: SVM
Accuracy: 0.9528
Precision: 0.9316
Recall: 0.9736
F1 Score: 0.9521

Model: Decision Tree
Accuracy: 0.9204
Precision: 1.0000
Recall: 0.8349
F1 Score: 0.9100

Model: Naive Bayes
Accuracy: 0.9290
Precision: 0.9808
Recall: 0.8698
F1 Score: 0.9220
```

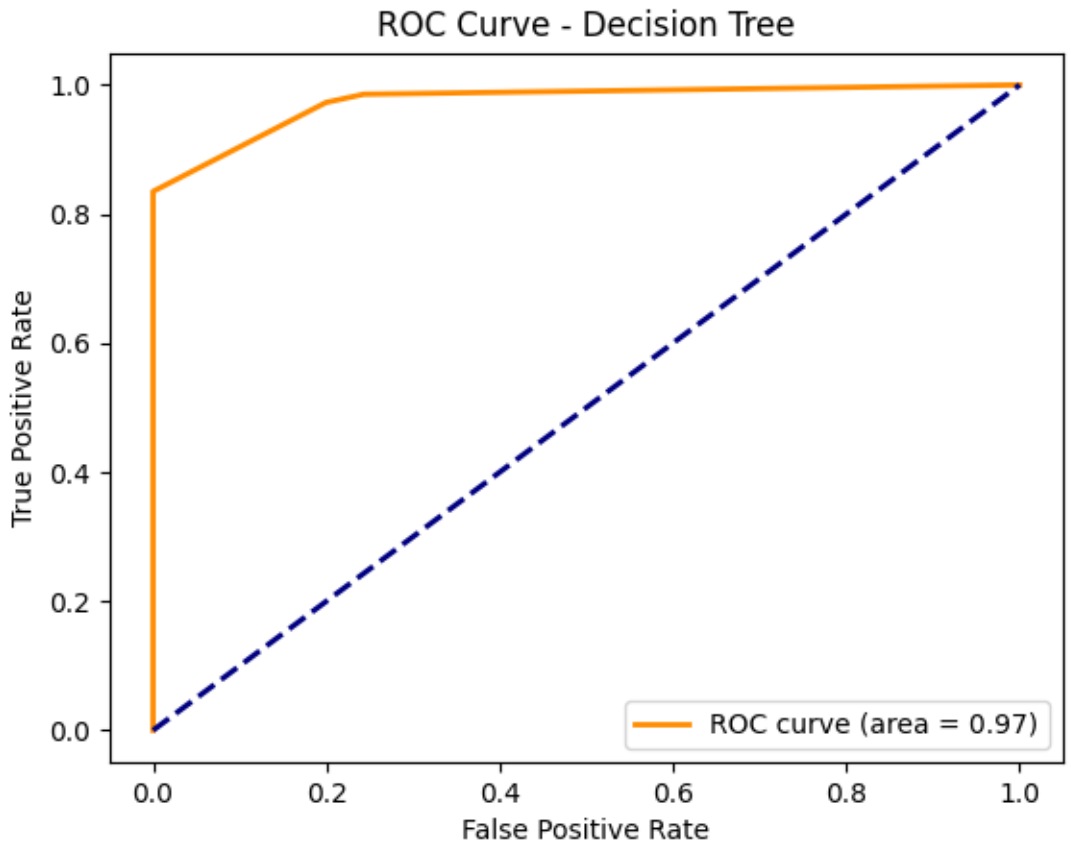
Come si può notare, emerge chiaramente che il Random Forest è il modello con performance complessive migliori, poiché riduce al minimo gli errori di classificazione sia in termini di falsi positivi che di falsi negativi. Sebbene l'SVM offre risultati competitivi (in particolare per quanto riguarda il recall) le sue performance risultano inferiori a quelle del Random Forest.

Il Decision Tree, sebbene sia eccellente nel garantire l'assenza di falsi positivi (Precision = 1), mostra evidenti limitazioni nel riconoscimento dei casi positivi (Recall basso e quindi falsi negativi elevati).

Infine, il modello Naive Bayes presenta una precisione molto elevata, ma come per il Decision Tree il suo basso recall ne compromette l'efficacia globale

**Decision Tree**

Di seguito la curva ROC:




La curva ROC conferma che il modello ha una alta capacità discriminativa tra le classi, ma il basso recall si riflette sulla curva, che non raggiunge il massimo possibile.

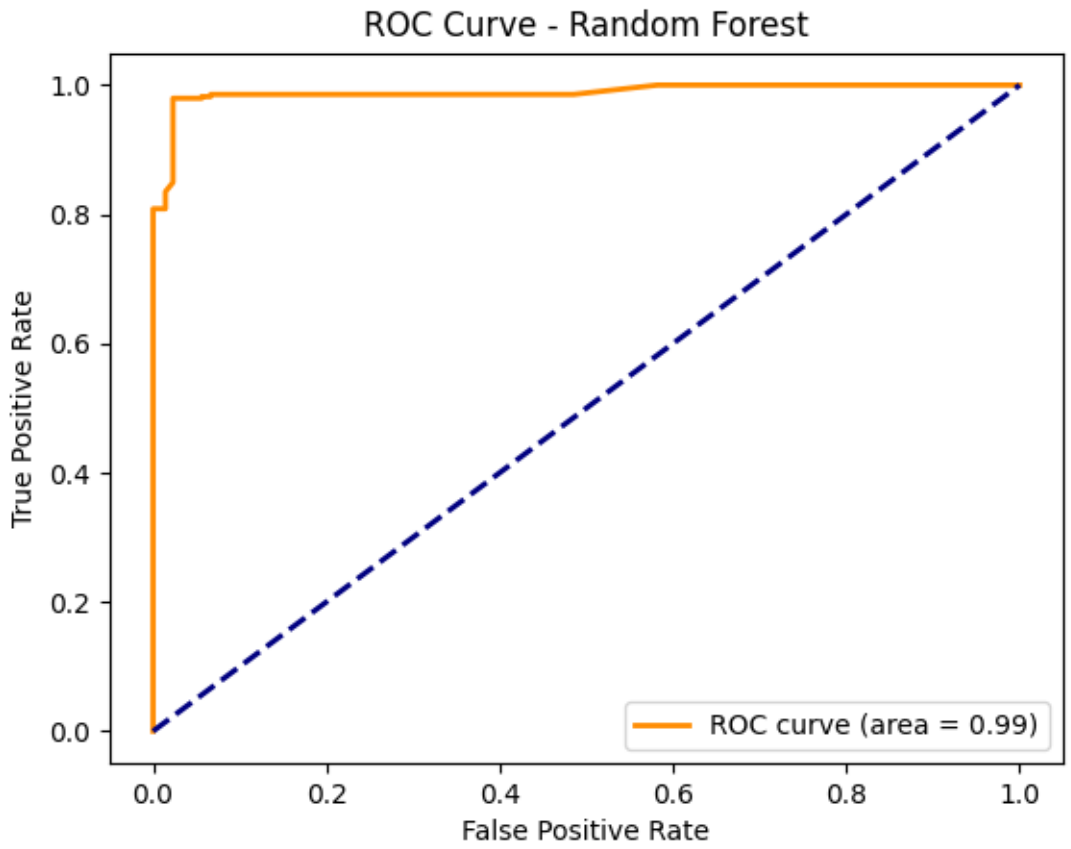
Di seguito la curva di apprendimento:


Il train error cresce leggermente con l'aumentare della dimensione del training set, mentre il test error rimane costante, ma poiché la differenza tra di essi e minima si può dedurre che il modello e ben generalizzato e non soffre di overfitting.

A causa probabilmente della semplicità (max_depth = 2), al modello viene limitata la capacità di adattarsi troppo ai dati di training, ma gli impedisce di catturare i pattern più complessi.

**Random Forest**

Di seguito la curva ROC:




La curva ROC conferma che il modello ha una capacità discriminativa molto alta; tuttavia, l'andamento non perfettamente lineare suggerisce che ci sono problemi nella classificazione di dati complessi.

DI seguito la curva di apprendimento:

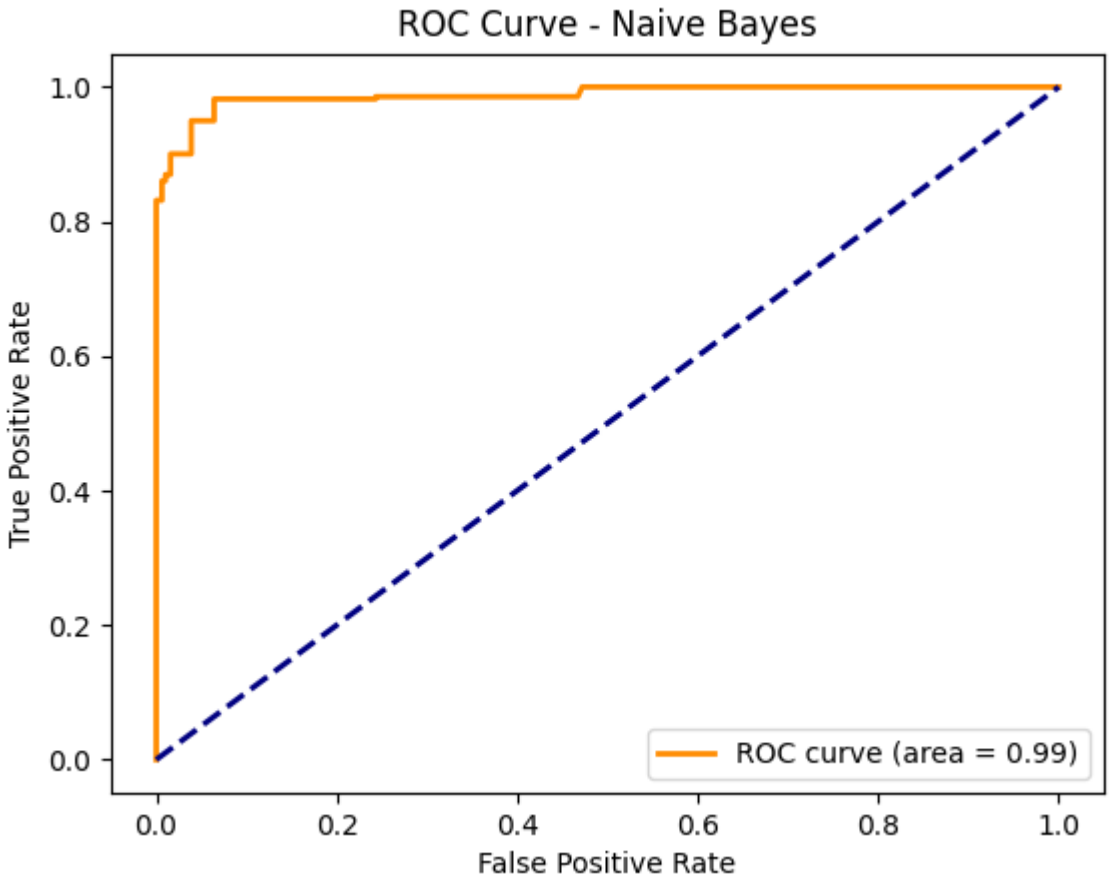

La differenza tra le due curve è piccola ma evidente, il che indica che il modello generalizza bene senza soffrire di overfitting, ma l'aumento del test error con dimensioni crescenti del train error indica che il modello non sta catturando completamente la complessità del dataset.

I possibili miglioramenti potrebbero essere:

- effettuare una maggiore regolarizzazione aumentando i valori di min_samples_split o di min_samples_leaf per limitare la crescita degli alberi
- una riduzione del numero delle feature

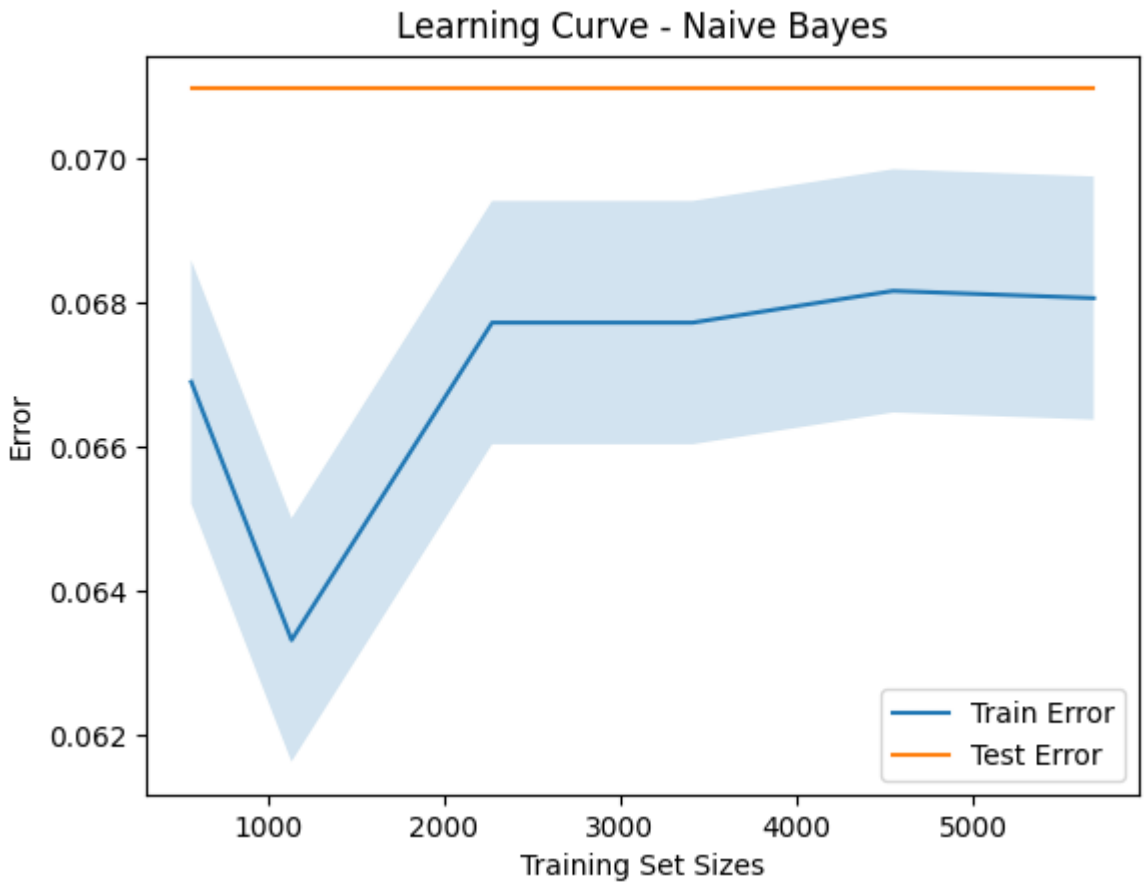
**Naive Bayes**

Di seguito la curva ROC:




La curva ROC conferma che il modello ha una capacità discriminativa molto alta, ma con un andamento non perfettamente lineare, questo comportamento è tipico del Naive Bayes poiché assume l'indipendenza tra le feature e può risultare meno efficace quando le feature sono fortemente correlate.

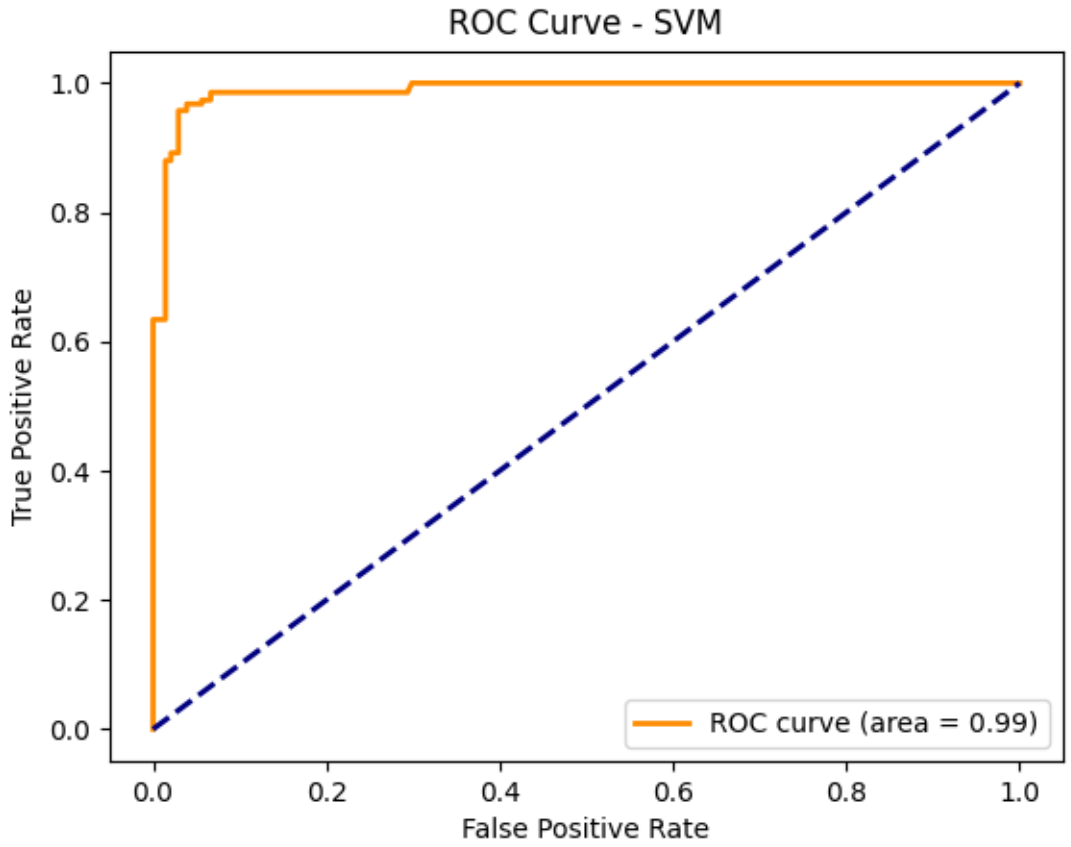
Di seguito la curva di apprendimento:




A differenza di modelli più complessi come Random Forest, la differenza tra Train Error e Test Error è minima sin dalle prime iterazioni. Questo comportamento indica che il modello non soffre di overfitting, ma d'altra parte suggerisce che potrebbe essere incapace di catturare relazioni più complesse tra le feature. Beneficia meno dell'aumento dei dati di training questo è dovuto al fatto che il Naive Bayes assume l'indipendenza tra le feature, e quindi, all'aumentare il numero di campioni, non può sfruttare meglio la correlazione tra gli attributi per migliorare le predizioni.

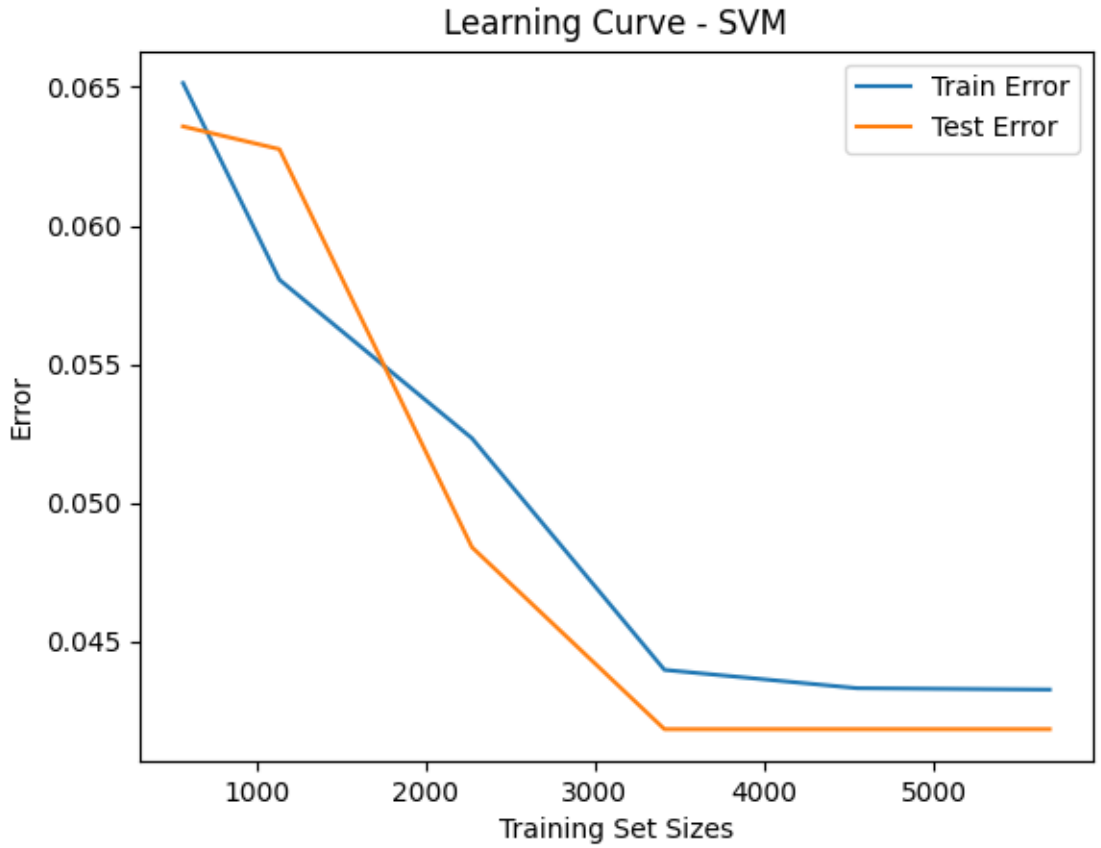
**SVM**

Di seguito la curva ROC:




Come possiamo notare la curva ROC è quasi perfetta indicando che il modello distingue chiaramente tra le due classi, come per il Random Forest e il Naive Bayes però vi è un leggero andamento irregolare.

Di seguito la curva di apprendimento:




La curva del Train Error decresce rapidamente con l'aumento dei dati, quella del Test Error segue un andamento simile, stabilizzandosi come quella del Train Error.

Il calo del Test Error con l'aumentare dei dati dimostra che il modello beneficia di più dati di training.

## Analisi della varianza e deviazione standard degli errori

La varianza e la deviazione standard degli errori sono state calcolate per valutare la stabilità di ogni modello su diverse dimensioni del Training Set e del Test Set.

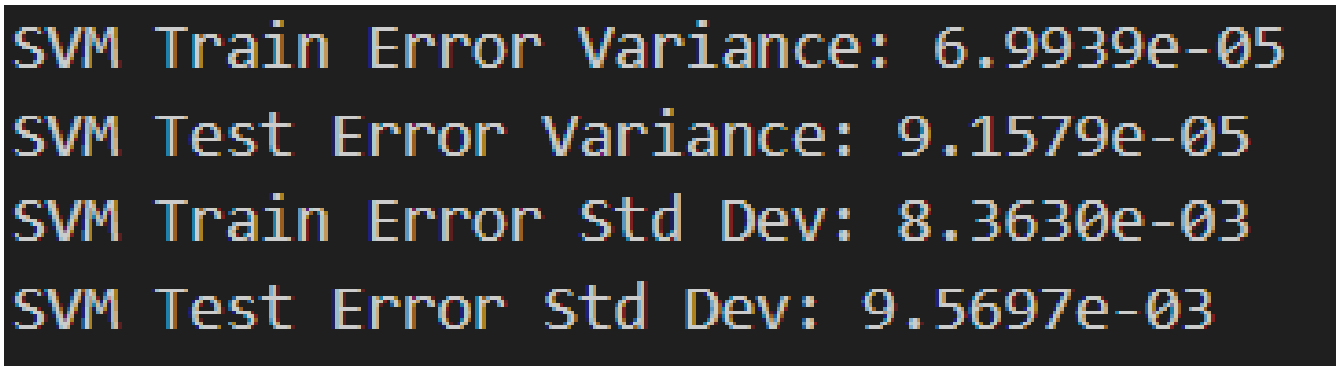
Per ogni modello, la varianza indica quanto varia l'errore tra diverse iterazioni di training e test, mentre la deviazione standard misura quanto queste variazioni si discostano dalla media.

```
Decision Tree Train Error Variance: 4.0373e-06
Decision Tree Test Error Variance: 0.0000e+00
Decision Tree Train Error Std Dev: 2.0093e-03
Decision Tree Test Error Std Dev: 0.0000e+00
```

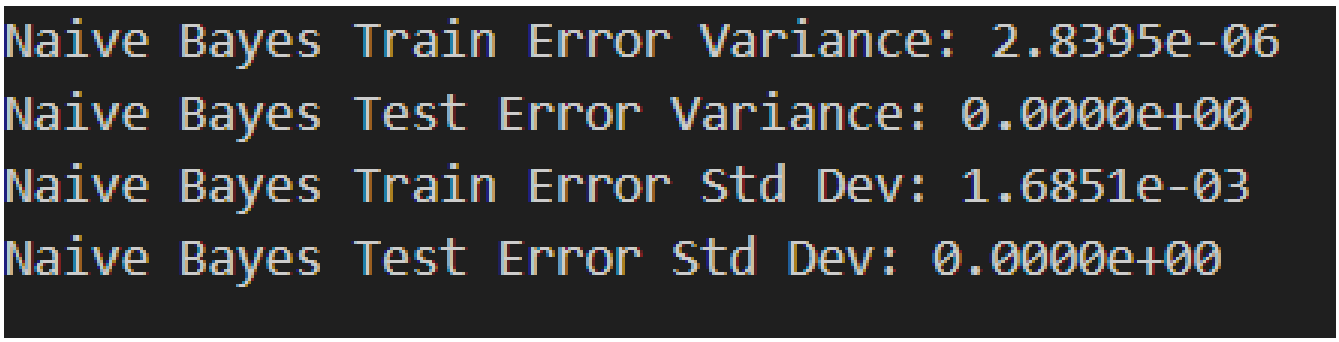
La bassa varianza del train error indica che l'errore è molto stabile su diverse dimensioni del training set, visto che la varianza e deviazione standard sono nulle per il test error il Decision Tree produce risultati completamente stabili sul set di test, ma come detto in precedenza adattandosi facilmente ai dati potrebbe non catturare pattern complessi.

```
Random Forest Train Error Variance: 1.5852e-04
Random Forest Test Error Variance: 2.4017e-04
Random Forest Train Error Std Dev: 1.2591e-02
Random Forest Test Error Std Dev: 1.5498e-02
```

La varianza e la deviazione standard sono le più alte tra i modelli, ciò suggerisce che il Random Forest è più sensibile alla variabilità dei dati di training, il che può essere un segnale di maggiore flessibilità ma anche di potenziale overfitting.




I valori indicano che il modello SVM è un modello molto stabile, con un errore che varia poco tra le diverse iterazioni di training e test. La sua varianza leggermente superiore rispetto a Decision Tree suggerisce che il modello si adatta meglio rispetto a modelli più semplici, ma senza la variabilità elevata osservata in Random Forest.




Il modello Naive Bayes presenta una varianza del Test Error pari a 0, così come la sua deviazione standard. Questo indica che il modello produce risultati completamente stabili sul Test Set, indipendentemente dai dati utilizzati per il training.

## Test su dati specifici

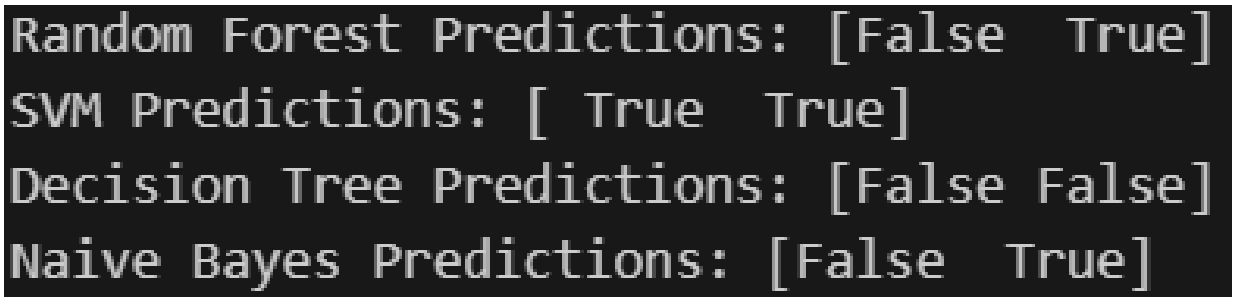
Infine, ho deciso di utilizzare i modelli per fare previsioni su un set di dati specifico, rappresentante due tipi di funghi: uno commestibile e uno velenoso;

per osservare come ogni modello classifica nuovi dati sulle basi delle caratteristiche fornite. I due funghi selezionati per il test sono:

- **Agaricus bisporus (commestibile)**
    - Cappello convesso, superficie liscia, colore bianco
    - Assenza di lividi
    - Odore assente
    - Lamelle di colore rosa
    - Gambo cilindrico con anello
    - Spore di colore marrone
    - Habitat erboso
- **Amanita phalloides (velenoso)**
    - Cappello convesso, superficie liscia, colore verdastro
    - Assenza di lividi
    - Odore penetrante
    - Lamelle di colore bianco
    - Gambo ingrossato alla base con anello
    - Spore di colore bianco
    - Habitat boschivo

Dopo l'addestramento dei modelli ho riempito quindi le varie colonne del dataset con True se il fungo presenta le caratteristiche e False se il fungo non presenta le caratteristiche.

Ricordando che la classe target è p, quindi velenoso i risultati sono i seguenti:




Random Forest e Naive Bayes, sono riusciti entrambi a predire correttamente i funghi.

Decision Tree ha un problema di falso negativo, classifica entrambi i funghi come non velenosi, una possibile modifica per ottimizzare Decision Tree potrebbe essere aumentare la sua complessità (quindi aumentare la max_depth).

SVM invece ha un problema di falso positivo, classifica entrambi i funghi come velenosi, una possibile modifica per ottimizzare SVM potrebbe essere aumentare il parametro di regolarizzazione (C).

# CONCLUSIONI E SVILUPPI FUTURI

L’obiettivo di questo progetto è stato di sviluppare un sistema in grado di distinguere funghi velenosi e edibili utilizzando modelli di Machine Learning e una Knowledge Base in Prolog.

Dopo aver preprocessato i dati e selezionato le feature più rilevanti tramite test statistici (Chi-quadro e Mutual Information), abbiamo addestrato quattro modelli di apprendimento scelti: Random Forest, SVM, Decision Tree e Naive Bayes.

Dalle analisi delle metriche e delle curve di apprendimento, abbiamo osservato quanto segue:

- Random Forest ha ottenuto buoni risultati, dimostrando una buona capacità di generalizzazione. Tuttavia, c'è un rischio di overfitting, una soluzione potrebbe essere l’incremento del parametro min_samples_leaf
- SVM ha mostrato un buon equilibrio tra accuratezza e stabilità, con una varianza e deviazione standard contenute. Tuttavia, ha presentato alcuni falsi positivi, suggerendo che potrebbe essere necessario ottimizzare ulteriormente il parametro C diminuendolo
- Decision Tree ha mostrato difficoltà nel riconoscere tutti i funghi velenosi, con un problema di falsi negativi. Aumentare la profondità massima (max_depth) potrebbe migliorare le prestazioni, ma bisogna fare attenzione al rischio di overfitting
- Naive Bayes si è dimostrato un modello semplice ma efficace, con buone prestazioni nel distinguere tra funghi velenosi e edibili. Tuttavia, la sua assunzione di indipendenza tra le caratteristiche potrebbe limitare la sua capacità di catturare relazioni più complesse nei dati

L’ottimizzazione degli iperparametri attraverso Grid Search e Cross-Validation (k = 10) ha permesso di migliorare ulteriormente le prestazioni dei modelli, garantendo una stima più affidabile delle loro capacità di generalizzazione.

Per migliorare questo progetto, si potrebbe:

- Integrare il dataset con nuovi dati reali per testare il sistema su campioni più vari
- Aumentare la complessità delle regole della Knowledge Base basate su correlazioni statistiche più avanzate e testare l’inferenza logica con altri dataset
- Creare dati sintetici per bilanciare meglio le classi e ridurre il rischio di bias dei modelli
- Regolarizzare i modelli per ottimizzare l’overfitting