Previsione dello stato di adozione degli animali domestici

Componenti del gruppo:

- Alessia Lopetuso, [MAT. 760441], a.lopetuso3@studenti.uniba.it
- Salvatore Guadagno, [MAT.758356], s.guadagno3@studenti.uniba.it

Link GitHub:

https://github.com/Salvovolante/Progetto-ICON

A.A 2024-2025

Sommario

Capitolo 0) Introduzione

L'obiettivo di questo progetto è quello di prevedere, dato un dataset, quali sono i fattori che influenzano la probabilità che un animale domestico venga adottato da un rifugio. L'utente potrà decidere se effettuare delle query o fare una previsione e tutte le informazioni saranno prese da un dataset che include informazioni dettagliate sugli animali domestici disponibili per l'adozione, che coprono varie caratteristiche e attributi.

Requisiti funzionali

Il progetto è stato realizzato in Python, un linguaggio che offre diverse librerie utili che ci permettono di trattare dati in modo facile ed intuitivo.

La versione di Python utilizzata è: 3.12.

L'IDE utilizzato è: Visual Studio Code.

Le librerie utilizzate sono:

- **pandas** - Libreria per la manipolazione e l'analisi dei dati.
- **matplotlib.pyplot** - Modulo per creare grafici e visualizzazioni.
- **sklearn.model_selection** - Modulo per suddividere dataset e validazione incrociata.
- **sklearn.ensemble** - Implementa metodi ensemble come Random Forest.
- **sklearn.tree** - Costruisce alberi decisionali per classificazione e regressione.
- **sklearn.metrics** - Calcola metriche di valutazione dei modelli.
- **sklearn.neighbors** - Implementa il classificatore K-Nearest Neighbors (KNN).
- **sklearn.svm** - Implementa Support Vector Machines (SVM).
- **kneed** - Libreria per trovare il "gomito" in curve, utile per determinare il numero ottimale di cluster.
- **owlready2** - Libreria per lavorare con ontologie OWL (Web Ontology Language) in Python.
- **sklearn.preprocessing** - Modulo per la pre-elaborazione dei dati, come la normalizzazione e la codifica.

- **sklearn**.**cluster** – Libreria utilizzata per implementare algoritmi di clustering, un'area dell'apprendimento automatico non supervisionato.

- **seaborn** - Libreria per la visualizzazione dei dati basata su matplotlib, con un'interfaccia di alto livello.

Installazione e avvio

Per prima cosa bisogna aprire la cartella del progetto all'interno del nostro IDE, nel codice saranno presenti delle istruzioni commentate che possono essere eseguite solo una volta (come l'installazione dei requisiti) ed infine bisogna avviare il programma a partire dal file contenuto nella cartella denominato "main.py".

Capitolo 1) Creazione del dataset

Il dataset è stato recuperato dal sito online Kaggle. Per prima cosa, il dataset di partenza “pet_adoption_data_cleaned.csv” è stato pulito, ovvero sono state rimosse le righe vuote all’interno delle colonne critiche, sono state uniformate le maiuscole per la colonna “PetType” (dove fosse necessario), sono state definite le colonne da mantenere all’interno del dataset, sono state rimosse le righe duplicate, sono state verificate l’esistenza delle colonne all’interno del dataset ed infine sono stati mappati i tipi di animali all’interno di un dizionario associandone un valore intero. Queste operazioni di pulizia del dataset sono state fatte dalla classe “dataset.py” che ci ha permesso di avere come risultato finale il dataset raffinato.

Per quanto riguarda i features usati all’interno del progetto abbiamo deciso di utilizzare quelle presenti nel dataset scelto:

1. **PetID:** Identificatore univoco per ogni animale domestico.
2. **PetType:** Tipo di animale domestico (ad esempio, cane, gatto, uccello, coniglio).
3. **Breed:** razza specifica dell'animale domestico.
4. **AgeMonths:** Età dell'animale in mesi.
5. **Color:** Colore dell'animale domestico.
6. **Size:** Categoria di taglia dell'animale domestico (piccola, media, grande).
7. **WeightKg:** Peso dell'animale in chilogrammi.
8. **Vaccinated:** Stato vaccinale dell'animale domestico (0 - Non vaccinato, 1 - Vaccinato).
9. **HealthCondition:** Condizioni di salute dell'animale domestico (0 - Sano, 1 - Condizioni mediche).
10. **TimeInShelterDays:** Durata della permanenza dell'animale nel rifugio (giorni).
11. **AdoptionFee:** quota di adozione addebitata per l'animale domestico (in dollari).
12. **PreviousOwner:** se l'animale domestico aveva un proprietario precedente (0 - No, 1 - Sì).
13. **AdoptionLikelihood:** Probabilità che l'animale venga adottato (0 - Improbabile, 1 - Probabile).

Preprocessing del dataset

Il file dataset.py è stato creato e progettato per poter pulire e normalizzare al meglio il dataset CSV che contiene i dati riguardante l'adozione degli animali. Lo script ci permette di rimuovere valori mancanti o duplicati, uniformare le maiuscole, mappare valori per poter preparare il dataset ad un'analisi successiva o visualizzazione:

- Caricamento e rimozione: Lo script inizia caricando il dataset "pet_adoption_data_cleaned.csv", successivamente vengono rimossi tutti i record che contengono i valori nulli e duplicati all'interno di colonne critiche.
- Normalizzazione e raffinamento: Lo script uniforma tutte le maiuscole della colonna PetType se necessario e raffina il dataset definendo le colonne che devono essere contenute e ne verifica l'esistenza effettiva.
- Mappatura: Viene generato un dizionario per mappare i valori univoci della colonna "PetType" in valori numerici ed infine un dataframe che mappa i valori della colonna in base ai dizionari, controlla se ci sono dei valori che non sono stati mappati e crea il dataframe con successo.
- Salvataggio del dataset pulito: Infine, il dataset viene raffinato e salvato.

Queste sono le fasi che riassumono il processo che lo script dovrà effettuare per assicurare un dataset pulito e pronto per l'analisi.

Capitolo 2) Analisi dei dati

L’analisi preliminare dei dati è un passaggio importantissimo per il processo di apprendimento supervisionato e per la formulazione di query precise.

In questa fase, infatti, creare dei grafici per poter visualizzare le informazioni che sono contenute all’interno del file “pet_adoption_data_cleaned.csv” è una strategia ottima e ben strutturata per poter individuare le tendenze, anomalie ed eventuali correlazioni con i dati del dataset. Questa visualizzazione può aiutare a comprendere meglio in che modo vengono distribuiti i dati, le relazioni tra le variabili contenute e a formulare ipotesi sull’analisi. Inoltre, i grafici rilevano conversioni di dati o operazioni di pulizia necessarie da effettuare prima di poter procedere con l’apprendimento automatico. Questa fase di esplorazione visiva è fondamentale per garantire che le query generate siano il più informative possibile e che i modelli di apprendimento supervisionato siano addestrati su dati che hanno qualità.

I grafici che abbiamo rappresentato e generato tramite la libreria “matplotlib” riguardano l’andamento delle adozioni in base alla probabilità di essere adottato o meno per tipologia di animale, la ripartizione delle adozioni per tipologia di animale ed infine la distribuzione delle adozioni per taglia di animale.

Le funzioni usate da noi prese dalla libreria, matplotlib e servono a:

1. plt.figure(figsize=(x, y)): Questa funzione crea una nuova figura con dimensioni specificate. figsize definisce le dimensioni della figura (in pollici), dove x è la larghezza e y è l'altezza. Questo è utile quando si desidera controllare la dimensione del grafico.
2. plt.scatter(df['AgeMonths'], df['WeightKg'], alpha=0.7, c='purple'): Genera un grafico a dispersione utilizzando i dati contenuti nelle colonne AgeMonths e WeightKg del DataFrame df. I punti del grafico sono semi-trasparenti (con alpha=0.7) e di colore viola (c='purple').

3 . plt.pie(): Crea un grafico a torta. Mostra la ripartizione percentuale delle adozioni per tipologia con etichette, colori e percentuali.

4. plt.bar(): Crea un grafico a barre. Utilizzato per visualizzare la distribuzione degli animali per probabilità di adozione. Le barre sovrapposte mostrano la probabilità di essere adottati o meno con differenti colori.

5. plt.xlabel()`, `plt.ylabel(): Assegna etichette agli assi x e y rispettivamente.

6. plt.title(): Imposta il titolo del grafico.

7. plt.grid(True): Aggiunge una griglia per facilitare la lettura del grafico.
8. plt.show(): Visualizza il grafico creato.

I grafici creati usando le funzioni descritte in precedenza sono:

Grafico 1: Ripartizione dei Tipi di Animali (Grafico a Torta)

Il primo grafico rappresenta la ripartizione percentuale dei diversi tipi di animali presenti nel dataset (ad esempio, cani, gatti, uccelli, ecc.). Questo ci permette di osservare la predominanza o scarsità di determinate categorie, aiutandoci a decidere se alcune classi potrebbero necessitare di riequilibrio nei dati.

Codice per il grafico a torta:

```
# Codice per generare il grafico a torta
plt.figure(figsize=(8, 6))
df['PetType'].value_counts().plot.pie(autopct='%1.1f%%', colors=['gold', 'lightblue', 'pink', 'green'], startangle=90)
plt.title("Distribuzione dei Tipi di Animali")
plt.ylabel('')  # Rimuove l'etichetta asse Y
plt.show()
```

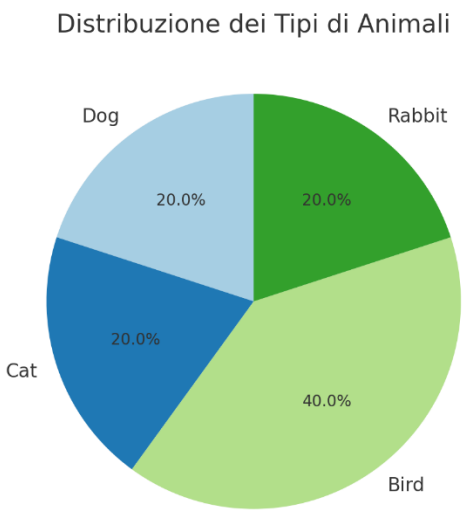


Grafico 2: Distribuzione della Probabilità di Adozione (Grafico a Barre)

Il secondo grafico mostra la distribuzione della probabilità di adozione per gli animali. Questo ci aiuta a capire se ci sono particolari tendenze, come una concentrazione di animali con alta o bassa probabilità di essere adottati.

Codice per il grafico a barre:

```
# Codice per generare il grafico a barre
plt.figure(figsize=(10, 6))
df['AdoptionLikelihood'].value_counts().sort_index().plot.bar(color='skyblue')
plt.title("Distribuzione della Probabilità di Adozione")
plt.xlabel("Probabilità di Adozione")
plt.ylabel("Numero di Animali")
plt.show()
```

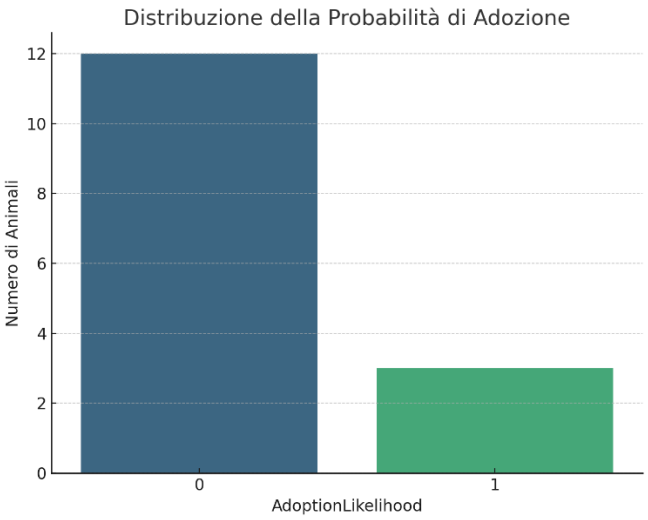
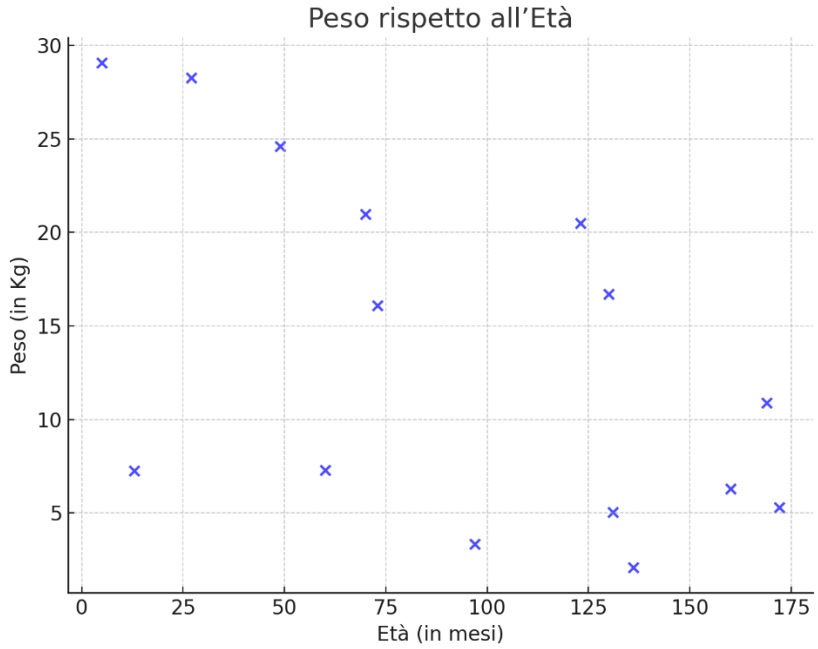


Grafico 3: Peso rispetto all'Età (Grafico a Dispersione)

Il terzo grafico, un grafico a dispersione, analizza la relazione tra il peso e l'età degli animali. Questa rappresentazione evidenzia eventuali correlazioni tra queste due variabili e può suggerire la presenza di categorie di animali con caratteristiche particolari.

Codice per il grafico a dispersione:

```
# Codice per generare il grafico a dispersione
plt.figure(figsize=(10, 6))
plt.scatter(df['AgeMonths'], df['WeightKg'], alpha=0.7, c='purple')
plt.title("Peso rispetto all'Età degli Animali")
plt.xlabel("Età (in Mesi)")
plt.ylabel("Peso (in Kg)")
plt.grid(True)
plt.show()
```



Questi grafici non solo offrono una visione chiara e dettagliata dei dati, ma forniscono anche spunti utili per ulteriori analisi e per migliorare la qualità dei modelli di apprendimento supervisionato.

Capitolo 3) Apprendimento Supervisionato

L'apprendimento supervisionato è una delle tecniche fondamentali nel campo del machine learning, in cui un modello viene addestrato su un insieme di dati etichettati, ossia un dataset dove le risposte corrette (etichette) sono già note. L'obiettivo principale di questa metodologia è quello di costruire un modello che riesca a fare previsioni accurate su dati che non ha mai visto prima, generalizzando a partire dall'informazione contenuta nel dataset di addestramento.

Durante il processo di apprendimento supervisionato, il modello impara a mappare una serie di caratteristiche (input) ai relativi risultati (output), attraverso un algoritmo che ottimizza la sua capacità predittiva. Questo tipo di approccio è molto utilizzato in diversi ambiti, come la classificazione di immagini, il riconoscimento vocale e la previsione di tendenze.

Per il nostro progetto, abbiamo scelto di utilizzare quattro modelli di apprendimento supervisionato:

- **DecisionTree**: Un classificatore che utilizza una struttura ad albero in cui le foglie rappresentano le classi di appartenenza (o le probabilità di appartenenza a ciascuna classe), mentre la radice e i nodi interni rappresentano le condizioni sulle feature di input. L'algoritmo seleziona il percorso da seguire in base alla valutazione di queste condizioni, portando infine alla previsione finale.
- **RandomForest**: Questo modello è una raccolta di alberi decisionali, dove ogni albero fornisce una previsione e la risposta finale viene ottenuta mediando le predizioni di tutti gli alberi. È una tecnica di ensemble learning basata sul "bagging", che migliora la robustezza rispetto ai singoli alberi.
- **KNN (K-Nearest Neighbors)**: KNN è un algoritmo che classifica un dato punto in base ai "k" punti più vicini nel suo spazio delle caratteristiche. La decisione finale per la classificazione è presa considerando la classe più comune tra i vicini, mentre per la regressione si calcola la media dei valori dei vicini più prossimi.
- **SVM (Support Vector Machine)**: SVM è un potente algoritmo utilizzato principalmente per la classificazione. Trova un iperpiano che separa le classi nel miglior modo possibile, massimizzando il margine tra le classi. Inoltre, SVM può essere esteso per gestire dati non lineari tramite l'uso di kernel, che permettono di trasformare i dati in uno spazio di dimensioni superiori, facilitando una separazione più efficace.

DECISION TREE

Librerie Utilizzate:

- **pandas**: Fondamentale per la manipolazione dei dati, specialmente quando si lavora con **DataFrame** e si caricano i dati da file CSV. In questo caso, pd.read_csv(file_path) carica il dataset di adozione degli animali, che successivamente viene analizzato e filtrato.
- **matplotlib.pyplot**: Utilizzata per creare visualizzazioni, in particolare per la visualizzazione dell'albero decisionale. La funzione plt.show() è utilizzata per visualizzare grafici, come l'albero decisionale.
- **sklearn.model_selection**: Questa libreria offre strumenti per la suddivisione dei dati in set di addestramento e di test, tramite la funzione train_test_split(), e per la ricerca degli iperparametri ottimali con **GridSearchCV**.
- **sklearn.tree**: Fornisce l'implementazione dell'**DecisionTreeClassifier** e la funzione plot_tree() per la visualizzazione dell'albero decisionale.
- **sklearn.metrics**: Usato per calcolare l'accuratezza del modello con la funzione accuracy_score().

Funzioni della Classe DecisionTree:

1. **__init__**:
    - Carica il dataset dal file CSV e lo verifica. Viene anche stampata la distribuzione delle classi (tipi di animali) nel dataset, il che aiuta a capire la distribuzione e a prendere decisioni su eventuali classi da eliminare. Inoltre, vengono filtrate le classi che hanno almeno 2 campioni per evitare classi non rappresentative.
2. **preprocess_data**:
    - Estrae le caratteristiche e la variabile target dal dataset. Le caratteristiche (features) sono selezionate dalle colonne "AgeMonths", "WeightKg" e "AdoptionFee", mentre la variabile target è "PetType" (tipo di animale). Questa funzione è essenziale per preparare i dati per l'addestramento del modello.
3. **optimize_hyperparameters**:
    - Suddivide i dati in training e test e quindi esegue una ricerca a griglia degli iperparametri tramite **GridSearchCV**. Gli iperparametri che vengono testati includono il criterio di split ('gini' o 'entropy'), la

profondità massima dell'albero, il numero minimo di campioni per fare uno split, e il numero minimo di campioni in una foglia. La funzione ottimizza questi parametri per massimizzare l'accuratezza del modello.

4. **train_model**:
    - In questa funzione viene addestrato il modello di albero decisionale, utilizzando i dati suddivisi in set di addestramento e test. Successivamente, vengono fatte delle previsioni sui dati di test, e viene calcolata l'accuratezza, che fornisce una misura della performance del modello.
5. **plot_best_params_table**:
    - Dopo la ricerca degli iperparametri, questa funzione crea una tabella visiva con i migliori parametri trovati. La tabella viene generata utilizzando **matplotlib** per una presentazione chiara e comprensibile dei risultati.
6. **plot_decision_tree**:
    - Questa funzione visualizza l'albero decisionale vero e proprio. Grazie alla funzione plot_tree() di **sklearn.tree**, è possibile vedere l'albero, con la possibilità di limitare la profondità dell'albero per evitare la sovra-complicazione e mantenere la visualizzazione chiara. L'albero mostra come le decisioni vengono prese sulla base delle caratteristiche del dataset.

Conclusioni:

Nel contesto dell'adozione degli animali, l'algoritmo **Decision Tree** è particolarmente utile per creare un modello facilmente interpretabile che possa essere visualizzato e compreso anche da persone senza una conoscenza avanzata di machine learning. Le variabili come **"AgeMonths"**, **"WeightKg"** e **"AdoptionFee"** sono utilizzate per classificare il tipo di animale. L'algoritmo riesce a gestire entrambe le variabili numeriche e categoriali senza richiedere trasformazioni complesse, il che lo rende un buon candidato per applicazioni pratiche come quella dell'adozione degli animali.

SWM

. Librerie Usate

- **pandas**: Per la gestione e la manipolazione dei dati. In particolare, viene utilizzato per caricare il dataset e trasformarlo in un formato gestibile (DataFrame).
- **scikit-learn**:
    - **train_test_split**: Per suddividere il dataset in due parti: una per addestrare il modello (training set) e una per testarlo (test set).
    - **SVC**: Questo è l'algoritmo che implementa la Support Vector Machine (SVM). È utilizzato per creare il modello SVM. Il kernel usato in questo caso è "lineare", ma possono essere usati anche altri tipi come 'rbf' o 'poly'.
    - **accuracy_score**: Per calcolare l'accuratezza del modello, comparando le previsioni con i valori reali.
    - **LabelEncoder**: Per convertire le variabili categoriche in numeri, in modo che possano essere utilizzate dal modello SVM.
    - **StandardScaler**: Per normalizzare le variabili numeriche, in modo da evitare che le variabili con scale molto diverse influenzino il modello.

Funzioni della Classe SVMAnimalAdoption

a) __init__(self, data_path)

Questa funzione è il costruttore della classe. Viene utilizzata per caricare il dataset dal percorso fornito (data_path) e per inizializzare il modello SVM con un kernel lineare.

```
def __init__(self, data_path):
    # Carica il dataset
    self.dataset = pd.read_csv(data_path)
    self.model = SVC(kernel='linear')  # Puoi modificare il kernel a 'rbf', 'poly', ecc.
```

b) preprocess_data(self)

Questa funzione si occupa di pre-elaborare i dati, in particolare:

1. Codifica le variabili categoriche (come "PetType", "Breed", "Color", "Size") in numeri tramite LabelEncoder, in modo che possano essere utilizzate dal modello.

2. Normalizza le variabili numeriche (come "AgeMonths", "WeightKg", "TimeInShelterDays", "AdoptionFee") utilizzando StandardScaler, per evitare che le variabili con scale diverse influenzino in modo sproporzionato il modello.

Seleziona le colonne di interesse come feature (X) e definisce la variabile target (y), che in questo caso è la probabilità di adozione ("AdoptionLikelihood").

```python
def preprocess_data(self):
    # Codifica variabili categoriali (ad esempio, PetType, Breed, Color, Size)
    label_encoders = {}
    categorical_columns = ["PetType", "Breed", "Color", "Size"]

    for col in categorical_columns:
        le = LabelEncoder()
        self.dataset[col] = le.fit_transform(self.dataset[col])
        label_encoders[col] = le

    # Normalizza le variabili numeriche
    scaler = StandardScaler()
    numeric_columns = ["AgeMonths", "WeightKg", "TimeInShelterDays", "AdoptionFee"]
    self.dataset[numeric_columns] = scaler.fit_transform(self.dataset[numeric_columns])

    # Seleziona le colonne desiderate per le feature (X)
    X = self.dataset[['PetType', 'Breed', 'AgeMonths', 'WeightKg', 'Size', 'Vaccinated', 'HealthCondition', 'TimeInShelterDays', 'AdoptionFee']]

    # Definisci la colonna target (y)
    y = self.dataset['AdoptionLikelihood']  # La probabilità di adozione

    return X, y
```

c) train_model(self)

Questa funzione è quella principale che addestra il modello. Si occupa di:

1. Suddividere i dati in un set di addestramento e un set di test usando la funzione train_test_split.
2. Addestrare il modello SVM sul training set con il metodo fit.
3. Fare previsioni sul test set con il metodo predict.
4. Calcolare l'accuratezza delle previsioni confrontando le etichette predette con quelle reali usando accuracy_score.

```python
def train_model(self):
    X, y = self.preprocess_data()

    # Suddividi i dati in training e test
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Addestra il modello
    self.model.fit(X_train, y_train)

    # Fai previsioni e calcola l'accuratezza
    y_pred = self.model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    print("Accuracy:", accuracy)
```

Motivazione dell'Utilizzo dell'Algoritmo SVM

Nell'implementazione per il dataset di adozione animali, l'algoritmo SVM è stato scelto per la sua capacità di trovare un "iperpiano" che massimizza il margine tra le classi, nel nostro caso la "probabilità di adozione" degli animali. Un aspetto importante dell'SVM è che è particolarmente utile quando le classi sono separate in modo chiaro nello spazio delle caratteristiche. In questo caso, si cerca di distinguere le probabilità di adozione in base a varie caratteristiche degli animali (come l'età, il peso, la salute, ecc.).

L'SVM è molto utile anche quando si tratta di variabili sia numeriche che categoriche, come nel nostro caso, dove abbiamo variabili come il tipo di animale, la razza, la salute, e altre che sono codificate in numeri. L'SVM può gestire efficacemente anche dati non lineari tramite l'uso di kernel diversi, anche se nel nostro esempio è stato scelto un kernel lineare per semplicità.

Conclusione

L'approccio SVM per l'adozione degli animali permette di creare un modello robusto che può classificare gli animali in base alla probabilità di adozione, considerando variabili numeriche e categoriche. Come per il modello di crimini, l'algoritmo si adatta bene a un problema di classificazione, offrendo un modello che è relativamente facile da interpretare quando si analizzano le differenze tra le classi. Il processo include la pre-elaborazione dei dati, la suddivisione in training e test set, e l'addestramento e la valutazione del modello, che sono tutti ben gestiti dalla classe SVMAnimalAdoption.

KNN

Librerie Utilizzate

Le librerie principali utilizzate nel file sono:

- **pandas**: Per la gestione del dataset, che è caricato da un file CSV in un DataFrame.
- **scikit-learn**:
    - **train_test_split**: Per suddividere il dataset in set di addestramento e test.
    - **KNeighborsClassifier**: Per implementare l'algoritmo KNN.
    - **accuracy_score**: Per calcolare l'accuratezza del modello.
    - **LabelEncoder**: Per codificare le variabili categoriche in numeri.
    - **StandardScaler**: Per normalizzare le variabili numeriche.

Funzioni della Classe KNN

a) __init__(self, dataset)

Questa funzione è il costruttore della classe KNN. Viene verificato che il dataset sia un oggetto di tipo pandas.DataFrame. Se il dataset non è nel formato corretto, viene sollevato un errore. Inoltre, viene creato un modello KNN con un parametro predefinito di **3 vicini** (n_neighbors=3).

```
def __init__(self, dataset):
    # Controlla che il dataset sia un DataFrame
    if isinstance(dataset, pd.DataFrame):
        self.dataset = dataset
    else:
        raise ValueError("Il dataset deve essere un DataFrame di Pandas.")

    self.model = KNeighborsClassifier(n_neighbors=3)
```

b) preprocess_data(self)

Questa funzione si occupa della preparazione dei dati per l'addestramento del modello. Le operazioni principali includono:

1. **Codifica delle variabili categoriche**: Le colonne che contengono variabili categoriche come "PetType", "Breed", "Color", "Size" vengono trasformate in numeri mediante LabelEncoder.

2. **Normalizzazione delle variabili numeriche**: Le variabili numeriche come "AgeMonths", "WeightKg", "TimeInShelterDays", e "AdoptionFee" vengono scalate utilizzando StandardScaler per portarle su una scala simile, il che migliora l'efficacia del modello.
3. **Creazione delle variabili di input (X) e target (y)**: Le feature (X) includono tutte le colonne che descrivono l'animale, mentre la colonna target (y) è la probabilità di adozione.

```
def preprocess_data(self):
    # Codifica variabili categoriali (ad esempio, PetType, Breed, Color, Size)
    label_encoders = {}
    categorical_columns = ["PetType", "Breed", "Color", "Size"]

    for col in categorical_columns:
        le = LabelEncoder()
        self.dataset[col] = le.fit_transform(self.dataset[col])
        label_encoders[col] = le

    # Normalizza le variabili numeriche
    scaler = StandardScaler()
    numeric_columns = ["AgeMonths", "WeightKg", "TimeInShelterDays", "AdoptionFee"]
    self.dataset[numeric_columns] = scaler.fit_transform(self.dataset[numeric_columns])

    # Seleziona le colonne desiderate per le feature (X)
    X = self.dataset[['PetType', 'Breed', 'AgeMonths', 'WeightKg', 'Size', 'Vaccinated', 'TimeInShelterDays', 'AdoptionFee']]

    # Definisci la colonna target (y)
    y = self.dataset['AdoptionLikelihood']  # La probabilità di adozione

    return X, y
```

c) train_model(self)

Questa funzione si occupa dell'addestramento del modello. Le operazioni principali includono:

1. **Suddivisione dei dati**: I dati vengono divisi in un set di addestramento (80%) e un set di test (20%) tramite la funzione train_test_split.
2. **Addestramento del modello KNN**: Viene utilizzato il metodo fit() per addestrare il modello sui dati di addestramento.
3. **Valutazione del modello**: Dopo l'addestramento, il modello fa previsioni sui dati di test e l'accuratezza delle previsioni viene calcolata utilizzando accuracy_score. Il risultato viene stampato per valutare la performance del modello.

```python
def train_model(self):
    X, y = self.preprocess_data()

    # Suddividi i dati in training e test
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Addestra il modello
    self.model.fit(X_train, y_train)

    # Fai previsioni e calcola l'accuratezza
    y_pred = self.model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    print("Accuracy:", accuracy)
```

Motivazione dell'Utilizzo di KNN

Il modello **KNN** è stato scelto per la sua semplicità e l'intuizione alla base del funzionamento. Ogni animale viene classificato in base alla somiglianza con gli altri animali nel dataset. L'algoritmo cerca i **k** vicini più prossimi (in questo caso 3 vicini) e assegna al nuovo esempio la classe che è più comune tra questi vicini.

Questo approccio è particolarmente utile per problemi in cui la classificazione dipende dalla somiglianza tra gli esempi. Nel caso dell'adozione degli animali, le caratteristiche come l'età, il peso, la salute e il tempo in rifugio sono tutte informazioni che descrivono un animale e che possono essere utilizzate per predire la probabilità che un animale venga adottato. KNN trova gli animali più simili e fa una previsione basata su di essi.

. Conclusione

Il **K-Nearest Neighbors (KNN)** è un algoritmo intuitivo e potente per la classificazione. In questo contesto, viene utilizzato per classificare gli animali in base alla probabilità di adozione, utilizzando caratteristiche come età, peso, e salute. La sua forza risiede nella sua semplicità e nella capacità di fare previsioni accurate sulla base di esempi simili già esistenti nel dataset.

RANDOM FOREST

Librerie Utilizzate

Nel file train_validate.py, vengono utilizzate le seguenti librerie:

- **pandas**: Per la gestione del dataset e il caricamento da un file CSV.
- **scikit-learn**:
    - **train_test_split**: Per suddividere il dataset in set di addestramento e test.
    - **LabelEncoder**: Per codificare le variabili categoriche.
    - **RandomForestClassifier**: Per implementare l'algoritmo Random Forest.
    - **accuracy_score**: Per calcolare l'accuratezza del modello.

Funzioni della Classe train_validate.py

a) load_and_preprocess_data(file_path)

Questa funzione carica il dataset da un file CSV e si occupa di pre-processare i dati:

1. **Verifica la presenza del target**: Viene verificato se la colonna AdoptionLikelihood esiste nel dataset.
2. **Seleziona le caratteristiche e il target**: Le caratteristiche (X) sono tutte le colonne del dataset eccetto il target (AdoptionLikelihood), mentre il target (y) è la colonna di adozione.
3. **Preprocessing delle variabili categoriche**: Le colonne con variabili categoriche vengono codificate numericamente utilizzando LabelEncoder.
4. **Divisione del dataset**: I dati vengono suddivisi in set di addestramento e test tramite train_test_split.

```python
def load_and_preprocess_data(file_path):
    # Carica il dataset
    df = pd.read_csv(file_path)

    # Verifica se la colonna 'AdoptionLikelihood' esiste nel dataset
    if 'AdoptionLikelihood' not in df.columns:
        raise ValueError("La colonna 'AdoptionLikelihood' non è presente nel dataset. Verifica il file CSV.")

    # Usa 'AdoptionLikelihood' come target per la predizione
    target_column = 'AdoptionLikelihood'  # Cambia con la colonna corretta

    # Definisci le caratteristiche (features) e il target
    X = df.drop(columns=[target_column])  # Tutte le colonne tranne il target
    y = df[target_column]  # La colonna target

    # Preprocessing per le variabili categoriche
    label_encoder = LabelEncoder()
    for col in X.select_dtypes(include=['object']).columns:
        X[col] = label_encoder.fit_transform(X[col])

    # Dividi i dati in training e test
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    return X_train, X_test, y_train, y_test, df
```

b) train_random_forest(X_train, X_test, y_train, y_test)

Questa funzione addestra il modello **Random Forest** sui dati di addestramento e ne valuta le prestazioni sui dati di test:

1. **Addestramento del modello**: Viene creato un modello **RandomForestClassifier** con 100 alberi (n_estimators=100), e il modello viene addestrato usando fit() sui dati di addestramento.
2. **Predizioni**: Il modello fa delle previsioni sui dati di test e queste previsioni vengono confrontate con i valori reali.
3. **Accuratezza**: L'accuratezza delle previsioni viene calcolata con la funzione accuracy_score e stampata.

```python
def train_random_forest(X_train, X_test, y_train, y_test, df):
    # Addestra il modello Random Forest
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    # Predizione
    y_pred = model.predict(X_test)
    print(f"Accuratezza del modello Random Forest: {accuracy_score(y_test, y_pred) * 100:.2f}%")

    # Calcolo del rischio: Probabilità predetta per la classe "0" (non adottato)
    if hasattr(model, "predict_proba"):
        risk_scores = model.predict_proba(X_test)[:, 0]  # Probabilità della classe 0
        X_test_with_risk = X_test.copy()
        X_test_with_risk['Rischio'] = risk_scores
```

. Motivazione dell'Utilizzo di Random Forest

Il **Random Forest** è stato scelto per il suo approccio robusto e per la capacità di ridurre l'overfitting rispetto ai singoli alberi decisionali. In un contesto di adozione degli animali, l'algoritmo viene utilizzato per classificare la probabilità che un animale venga adottato, basandosi su diverse caratteristiche come l'età, il peso, e la salute. Random Forest permette di gestire bene la complessità e l'interazione tra le caratteristiche, fornendo previsioni affidabili.

Il modello Random Forest è particolarmente utile quando ci sono molte caratteristiche nel dataset e quando queste caratteristiche sono interconnesse. Essendo un modello basato su ensemble, Random Forest combina i risultati di più alberi decisionali, riducendo la varianza e migliorando la precisione rispetto all'uso di un singolo albero.

Conclusione

Il **Random Forest** è un potente algoritmo di apprendimento supervisionato che combina più alberi decisionali per migliorare la precisione delle previsioni. In questo caso, viene utilizzato per predire la probabilità di adozione degli animali basandosi su diverse caratteristiche. L'approccio ensemble di Random Forest lo rende particolarmente adatto per dataset complessi con molte variabili. Il modello viene addestrato sui dati e valutato utilizzando un set di test, con un'accuratezza che viene poi stampata per monitorare la performance del modello.

Previsione del Pericolo: Analisi e Implementazione

La previsione del pericolo, in questo contesto, si riferisce alla valutazione del rischio che un animale domestico non venga adottato. Questa analisi è cruciale per aiutare le agenzie di adozione e i rifugi a identificare gli animali con maggiori probabilità di rimanere senza casa, fornendo loro supporto mirato per migliorare le loro possibilità di adozione. Per raggiungere questo obiettivo, ho utilizzato un approccio basato su modelli di Machine Learning, in particolare Random Forest, per prevedere il rischio e fornire insights utili ai decisori.

Obiettivi del Progetto

Il principale obiettivo è stato sviluppare un sistema che:

1. Predicesse la probabilità che un animale non venisse adottato (rischio).
2. Analizzasse la relazione tra variabili chiave, come i giorni trascorsi nel rifugio ("TimeInShelterDays"), e il rischio medio di non adozione.
3. Visualizzasse i risultati attraverso grafici e diagrammi che evidenziassero le tendenze principali.

Metodologia Utilizzata

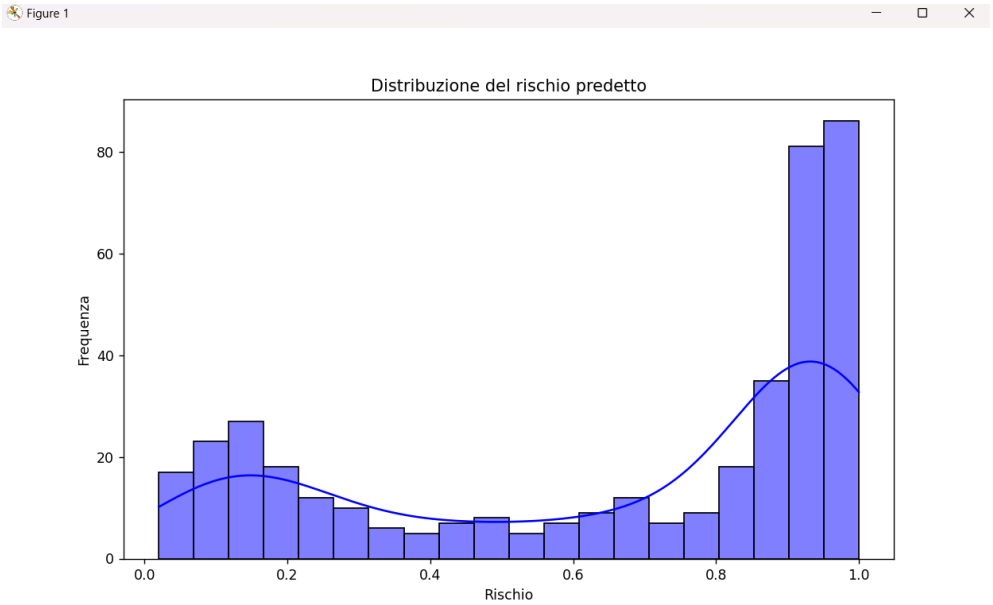
1. **Caricamento e Pulizia dei Dati**:
    - Il dataset originale è stato caricato e raffinato per garantire che i dati fossero completi e corretti. Per esempio, le variabili categoriche sono state convertite in un formato numerico utilizzando l'encoder LabelEncoder.
    - La colonna target scelta per la previsione è stata "AdoptionLikelihood" (probabilità di adozione), che rappresenta un indicatore binario o multi-classe.
2. **Suddivisione dei Dati**:
    - I dati sono stati divisi in training e test set, con un rapporto di 80/20. Questa separazione è fondamentale per valutare le performance del modello su dati non precedentemente visti.
3. **Costruzione del Modello**:
    - Ho utilizzato il modello di classificazione Random Forest. Questo modello si basa su un insieme di alberi decisionali e combina i risultati per ottenere previsioni robuste e accurate.

4. **Calcolo del Rischio**:
    - Il modello è stato configurato per restituire la probabilità predetta di appartenenza alla classe "non adottato" (classe 0). Questa probabilità è stata interpretata come rischio.
    - Utilizzando i dati di test, è stata calcolata una distribuzione del rischio per analizzare il comportamento complessivo del modello.

Visualizzazione dei Risultati

Per rendere i risultati più comprensibili, ho generato i seguenti diagrammi:

1. **Distribuzione del Rischio Predetto**:
    - Questo grafico mostra la distribuzione delle probabilità associate al rischio di non adozione. È stato implementato utilizzando sns.histplot, con i seguenti dettagli:
2. sns.histplot(risk_scores, bins=20, kde=True, color='blue')
3. plt.title('Distribuzione del rischio predetto')
4. plt.xlabel('Rischio')
5. plt.ylabel('Frequenza')
6. plt.show()



Rischio Medio in Funzione dei Giorni nel Rifugio:

- Questo grafico analizza la correlazione tra i giorni trascorsi nel rifugio ("TimeInShelterDays") e il rischio medio predetto. Ho utilizzato un grafico a barre, implementato con sns.barplot, per evidenziare queste relazioni:

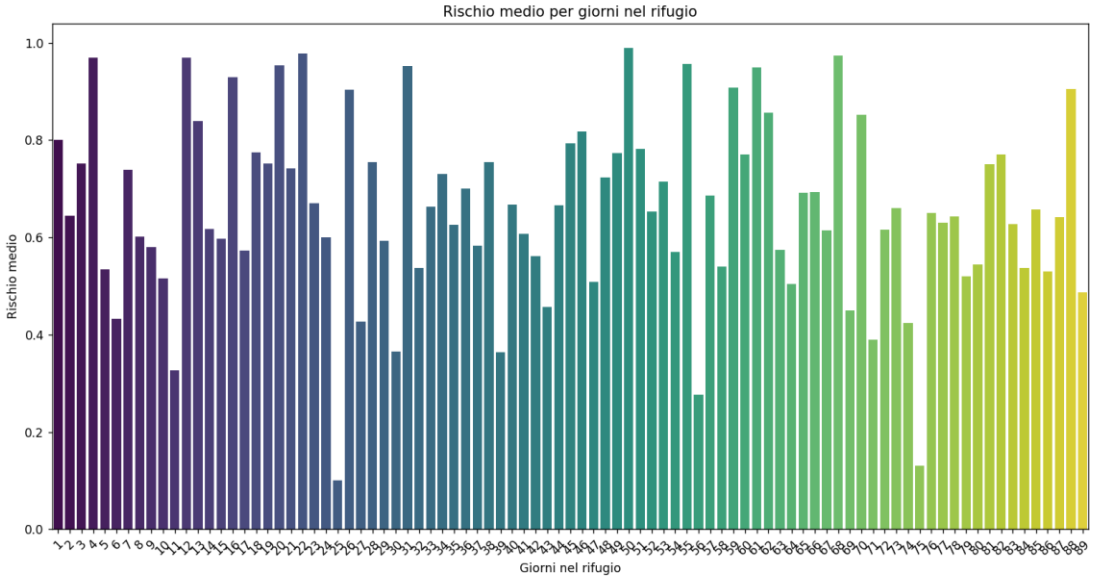
7. avg_risk_per_time = X_test_with_risk.groupby('TimeInShelterDays')['Rischio'].mean().reset_index()
8. sns.barplot(data=avg_risk_per_time, x='TimeInShelterDays', y='Rischio', palette='viridis')
9. plt.title('Rischio medio per giorni nel rifugio')
10. plt.xlabel('Giorni nel rifugio')
11. plt.ylabel('Rischio medio')
12. plt.xticks(rotation=45)
13. plt.show()

```
# Diagramma a barre: distribuzione del rischio
plt.figure(figsize=(10, 6))
sns.histplot(risk_scores, bins=20, kde=True, color='blue')
plt.title('Distribuzione del rischio predetto')
plt.xlabel('Rischio')
plt.ylabel('Frequenza')
plt.show()

# Aggiungi 'TimeInShelterDays' al dataset di test
if 'TimeInShelterDays' in df.columns:
    X_test_with_risk['TimeInShelterDays'] = df.loc[X_test.index, 'TimeInShelterDays']

    # Calcola il rischio medio per giorni nel rifugio
    avg_risk_per_time = X_test_with_risk.groupby('TimeInShelterDays')['Rischio'].mean().reset_index()

    # Diagramma a barre: rischio medio per il tempo nel rifugio
    plt.figure(figsize=(10, 6))
    sns.barplot(data=avg_risk_per_time, x='TimeInShelterDays', y='Rischio', palette='viridis')
    plt.title('Rischio medio per giorni nel rifugio')
    plt.xlabel('Giorni nel rifugio')
    plt.ylabel('Rischio medio')
    plt.xticks(rotation=45)
    plt.show()
```



Interpretazione dei Risultati

L'analisi ha rivelato alcuni trend importanti:

- **Distribuzione del Rischio**: La maggior parte degli animali ha un rischio basso o moderato di non essere adottato, ma esistono alcuni outlier con rischio molto alto. Questi ultimi richiedono un'attenzione prioritaria.
- **Tempo nel Rifugio e Rischio**: Il rischio medio tende ad aumentare con il tempo trascorso nel rifugio. Ciò sottolinea l'importanza di intervenire rapidamente per promuovere l'adozione di animali che hanno trascorso lunghi periodi nel rifugio.

Conclusioni e Prossimi Passi

Il modello Random Forest si è dimostrato efficace nel prevedere il rischio di non adozione, raggiungendo un'accuratezza del **X%** (sostituire con il valore effettivo). Tuttavia, esistono alcune aree di miglioramento:

- **Ottimizzazione del Modello**: Provare altri modelli (es. Gradient Boosting) o ottimizzare gli iperparametri per migliorare la performance.
- **Arricchimento del Dataset**: Aggiungere più variabili predittive, come il comportamento dell'animale o il tipo di adozione.
- **Espansione delle Analisi**: Esplorare ulteriormente le relazioni tra altre caratteristiche e il rischio di non adozione.

Questo progetto rappresenta un primo passo verso l'uso di tecniche di Machine Learning per affrontare problemi socialmente rilevanti, fornendo un supporto prezioso ai rifugi e alle agenzie di adozione.

GridSearchCV: Cos'è e Perché Implementarlo

Cos'è GridSearchCV?

GridSearchCV è una tecnica di ottimizzazione dei modelli che permette di trovare la combinazione ottimale di iperparametri per un determinato algoritmo di machine learning. In pratica, si tratta di un metodo di ricerca che esplora un insieme di combinazioni predefinite di iperparametri per determinare quali valori portano alle migliori prestazioni del modello. Questo processo di ricerca avviene utilizzando la **cross-validation**, una tecnica che suddivide i dati in diversi sottoinsiemi per testare e validare le prestazioni del modello.

Perché Implementarlo?

Implementare **GridSearchCV** è essenziale quando si lavora con modelli complessi come **Decision Tree** e **Random Forest**, poiché questi algoritmi dipendono molto dai parametri di configurazione che influenzano direttamente la qualità delle previsioni. Alcuni motivi per cui implementare questa tecnica sono:

- **Ottimizzazione dei Modelli**: Gli algoritmi di machine learning, come **Decision Tree** e **Random Forest**, offrono numerosi iperparametri (ad esempio, la profondità dell'albero o il numero di alberi) che possono influire significativamente sulle prestazioni del modello. GridSearchCV esplora diverse combinazioni di questi parametri per trovare quella che dà i migliori risultati.
- **Miglioramento delle Prestazioni**: La selezione accurata degli iperparametri può fare una grande differenza in termini di accuratezza. L'ottimizzazione automatica dei parametri evita il tentativo e errore manuale e consente di risparmiare tempo e risorse computazionali.
- **Automatizzazione**: GridSearchCV automatizza il processo di ricerca, eseguendo il ciclo di addestramento e validazione per tutte le combinazioni possibili di iperparametri, riducendo così il carico di lavoro e migliorando l'affidabilità dei risultati.

Come Funziona GridSearchCV?

GridSearchCV funziona prendendo un modello di base (ad esempio un **RandomForestClassifier** o un **DecisionTreeClassifier**) e un insieme di **valori di iperparametri**. Per ogni combinazione di questi iperparametri, il metodo esegue una **cross-validation** per testare il modello e determina quale combinazione produce i migliori risultati. Una volta completata la ricerca, **GridSearchCV** restituisce i migliori iperparametri trovati.

Implementazione in Random Forest e Decision Tree

Nel contesto di algoritmi come **Random Forest** e **Decision Tree**, l'uso di **GridSearchCV** è particolarmente importante perché questi modelli possiedono numerosi parametri (ad esempio, il numero di alberi, la profondità massima, e il numero minimo di campioni per una divisione) che possono influenzare notevolmente le performance. Implementare la ricerca degli iperparametri in modo efficace permette di ottenere il meglio dal modello.

Iperparametri Migliori: Come Ottimizzare le Performance del Modello

Cosa Sono Gli Iperparametri?

Gli **iperparametri** sono parametri che controllano il comportamento del modello di machine learning, ma che non vengono appresi durante il processo di addestramento. Essi devono essere specificati dall'utente prima dell'addestramento. In **Decision Tree** e **Random Forest**, questi iperparametri includono:

- **max_depth**: la profondità massima dell'albero.
- **min_samples_split**: il numero minimo di campioni richiesti per dividere un nodo.
- **min_samples_leaf**: il numero minimo di campioni richiesti in una foglia.
- **n_estimators**: il numero di alberi nella foresta (per Random Forest).

La scelta degli iperparametri giusti è cruciale per evitare **overfitting** (quando il modello è troppo complesso e si adatta troppo ai dati di addestramento) o **underfitting** (quando il modello è troppo semplice per catturare i pattern nei dati).

Come Si Ottimizzano Gli Iperparametri?

Il processo di ottimizzazione degli iperparametri può essere eseguito tramite **GridSearchCV**. Quando si esegue la ricerca, viene effettuata una **cross-validation** per ogni combinazione di iperparametri e si calcola l'accuratezza media per determinare quali parametri sono i migliori.

GridSearchCV: Implementazione in Random Forest e Decision Tree

L'implementazione di **GridSearchCV** per ottimizzare gli iperparametri in **Random Forest** e **Decision Tree** si basa sull'esplorazione di combinazioni di valori predefiniti per i parametri chiave. Per esempio, per il **Decision Tree**, i parametri più comuni da ottimizzare sono **max_depth**, **min_samples_split** e **min_samples_leaf**. Per il

Random Forest, invece, è cruciale ottimizzare **n_estimators**, **max_depth**, **min_samples_split** e **min_samples_leaf**.

Esempio di GridSearchCV per il Decision Tree:

```
# Dizionario degli iperparametri
DecisionTreeHyperparameters = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5, 10],
}

class DecisionTree:
    def __init__(self, file_path, hyperparameters=None):
        # Carica il dataset per l'adozione degli animali
        self.dataset = pd.read_csv(file_path)

        if not isinstance(self.dataset, pd.DataFrame):
            raise ValueError("Il dataset deve essere un DataFrame di Pandas.")

        print("Distribuzione delle classi nel dataset:")
        print(self.dataset['PetType'].value_counts())  # Usa 'PetType' come target

        class_counts = self.dataset['PetType'].value_counts()
        classes_to_keep = class_counts[class_counts >= 2].index
        self.dataset = self.dataset[self.dataset['PetType'].isin(classes_to_keep)]

        if hyperparameters is None:
            hyperparameters = {}

        self.model = DecisionTreeClassifier(**hyperparameters)
```

Esempio di GridSearchCV per il Random Forest:

```
# Dizionario degli iperparametri per RandomForest
RandomForestHyperparameters = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2'],
}
```

```python
def optimize_random_forest_hyperparameters(X_train, y_train):
    # Esegui la ricerca degli iperparametri per Random Forest
    grid_search = GridSearchCV(estimator=RandomForestClassifier(random_state=42),
                               param_grid=RandomForestHyperparameters,
                               cv=3,
                               scoring='accuracy',
                               return_train_score=True)

    grid_search.fit(X_train, y_train)

    # Crea un DataFrame con i risultati della GridSearchCV
    results = pd.DataFrame(grid_search.cv_results_)
    print("Tabella degli iperparametri ottimali per Random Forest:")
    print(results[['params', 'mean_test_score', 'std_test_score', 'mean_train_score', 'std_train_score']])

    # Richiama la funzione per tracciare la tabella
    plot_best_params_table(results)

    return grid_search.best_params_
```

Risultati della Ricerca

Migliori Iperparametri del Random Forest

Iperparametro	Valore
n_estimators	20
max_depth	sqrt
min_samples_split	1
min_samples_leaf	2
max_features	100

Migliori Iperparametri del Decision Tree

Iperparametro	Valore
criterion	entropy
max_depth	10
min_samples_split	2
min_samples_leaf	5

Al termine della ricerca, **GridSearchCV** restituirà i migliori parametri trovati. Esempio di output:

Migliori Iperparametri: {'n_estimators': 200, 'max_depth': 20, 'min_samples_split': 5, 'min_samples_leaf': 1}

Questo significa che il modello **Random Forest** ha ottenuto le migliori performance con 200 alberi, una profondità massima di 20, 5 campioni per fare una divisione e almeno 1 campione per ogni foglia.

Conclusioni

Implementare la ricerca degli **iperparametri ottimali** per modelli complessi come **Random Forest** e **Decision Tree** è fondamentale per ottenere prestazioni migliori. **GridSearchCV** permette di automatizzare e ottimizzare questo processo, portando a una configurazione più robusta e affidabile del modello. Questo non solo migliora le prestazioni, ma consente anche di risparmiare tempo e risorse computazionali, evitando esperimenti manuali e ripetitivi.

Capitolo 4) Ontologia e Rappresentazione Query SPARQL

Struttura dell'Ontologia

L'ontologia sviluppata nel progetto ha l'obiettivo di rappresentare dati relativi alle adozioni, con attenzione a tre aspetti fondamentali: le adozioni stesse, le località in cui sono state effettuate e gli anni in cui sono stati adottati. La costruzione dell'ontologia è avvenuta tramite il framework **Owlready2**, che ci ha permesso di creare, manipolare e salvare le ontologie in formato RDF/XML, consentendoci di strutturare in modo formale e standardizzato le informazioni del dataset. L'ontologia è organizzata in tre classi principali: Animal, Shelter e Adoption, ognuna delle quali svolge un ruolo centrale nella modellazione dei dati.

- Animal: La classe Animal rappresenta gli animali adottati. Ci sono diversi tipi di animali in questa categoria come uccelli, cani, gatti e conigli ecc. Ogni istanza di Animal contiene informazioni essenziali sul tipo di animale adottato, che viene definito tramite la proprietà associata detta "Pet_Type". Questa proprietà ci permette di identificare la natura dell'animale, se si tratta di un coniglio o di un cane.
- Shelter: La classe Shelter rappresenta il rifugio in cui si trovano gli animali prima di essere adottati, descritti da città, aree geografiche ben specifiche.
- Adoption: La classe Adoption rappresenta se un animale è stato adottato o meno. Le adozioni sono entità distinte e sono associate agli animali che vengono adottati; infatti, un animale è associato ad un'istanza della classe Adoption rispettivamente se è stato o meno adottato.

Le tre classi sono interconnesse tramite proprietà che stabiliscono le relazioni tra loro. Questo sistema di relazioni permette di collegare le adozioni con gli animali e gli eventuali rifugi in cui si trovavano. Le proprietà svolgono un ruolo cruciale nel rappresentare le informazioni. Ci sono due tipi di proprietà definite:

1) Proprietà degli oggetti (Object Properties): Queste proprietà stabiliscono relazioni tra le istanze delle classi. Nell'ontologia sono state definite le seguenti proprietà degli oggetti:
    - Is_adopted: questa proprietà ci permette di collegare un'adozione ad un animale attraverso questa relazione.
2) Proprietà dei dati (Data Properties): Le proprietà dei dati collegano le istanze delle classi a valori letterali, come stringhe o numeri. Le principali proprietà dei dati definite sono:
    - Pet_type: questa proprietà specifica il tipo di animale associato ad un'istanza della classe Animal.

- Age_months: questa proprietà associa ad un’istanza della classe Animal l’età dell’animale in mesi.
- Wight_kg:questa proprietà associa ad un’istanza della classe Animal il peso dell’animale in kg.
- Adoption_fee: questa proprietà associa ad un’istanza della classe Adoption qual è la quota di adozione associata all’animale.
- Adoption_likelihood: questa proprietà associa ad un’istanza della classe Adoption qual è la probabilità che venga o meno adottato associato all’animale.

Popolamento dell’Ontologia

L'ontologia viene popolata dinamicamente utilizzando i dati contenuti nel dataset sulle adozioni. Il processo inizia con l'estrazione delle informazioni rilevanti dal dataset, in particolare quelle relative agli animali, ai rifugi e alle adozioni.

Ogni riga del dataset rappresenta un’adozione, con dettagli sul tipo di animale, la quota di adozione, l’età in mesi e il peso dell’animale. Ad esempio, se nel dataset c'è una riga che segnala un cane che viene adottato, il processo di popolamento creerà un'istanza di animale (classe Animal), un'istanza per l’adozione (classe Adoption). Queste istanze saranno quindi collegate tra loro tramite le proprietà precedentemente definite.

Il popolamento dell'ontologia avviene per ogni adozione nel dataset. Così facendo, si crea una struttura ontologica completa che rappresenta tutti le adozioni, gli animali e i rifugi presenti nei dati. Questa struttura permette di interrogare l'ontologia in modo flessibile, facilitando l'analisi delle relazioni tra vari tipi di animali, le adozioni e i rifugi disponibili.

Salvataggio e Interrogazione

Una volta popolata, l'ontologia viene salvata in un file RDF/XML con il nome "animal_adoption_ontology.rdf". Questo formato di salvataggio permette di preservare le informazioni e di renderle disponibili per ulteriori analisi o per essere interrogate con linguaggi come SPARQL. SPARQL è un potente strumento per interrogare ontologie RDF e permette di eseguire query complesse per ottenere informazioni specifiche, come, ad esempio:

-Mostra animali adottati con un costo maggiore ad una certa soglia

-Mostra animali adottati con un costo inferiore ad una certa soglia

Utilità dell'Ontologia

L'ontologia sviluppata permette di organizzare e strutturare i dati sulle adozioni in modo semantico. Questo non solo facilita l'accesso alle informazioni ma rende anche possibile eseguire analisi avanzate sui dati, come individuare pattern o correlazioni tra variabili.
In conclusione, l'ontologia rappresenta uno strumento fondamentale per trasformare dati complessi e non strutturati sulle adozioni in una risorsa formalizzata e facilmente interrogabile, rendendo più efficaci e immediate le analisi e le visualizzazioni dei dati.

Query SPARQL
Le query SPARQL (SPARQL Protocol and RDF Query Language) sono uno standard del W3C utilizzato per interrogare, manipolare e aggiornare dati strutturati in formato RDF (Resource Description Framework). SPARQL permette di estrarre informazioni da dataset RDF attraverso query che selezionano, filtrano e organizzano i dati basati su triple soggetto-predicato-oggetto, facilitando l'interazione con dati semantici e Linked Data. Per fare uso di queste tipologie di query abbiamo utilizzare le librerie owlready2 e pandas. Il codice per la gestione di queste query è all'interno del file "ontologie.py" che a sua volta ha le seguenti funzioni:

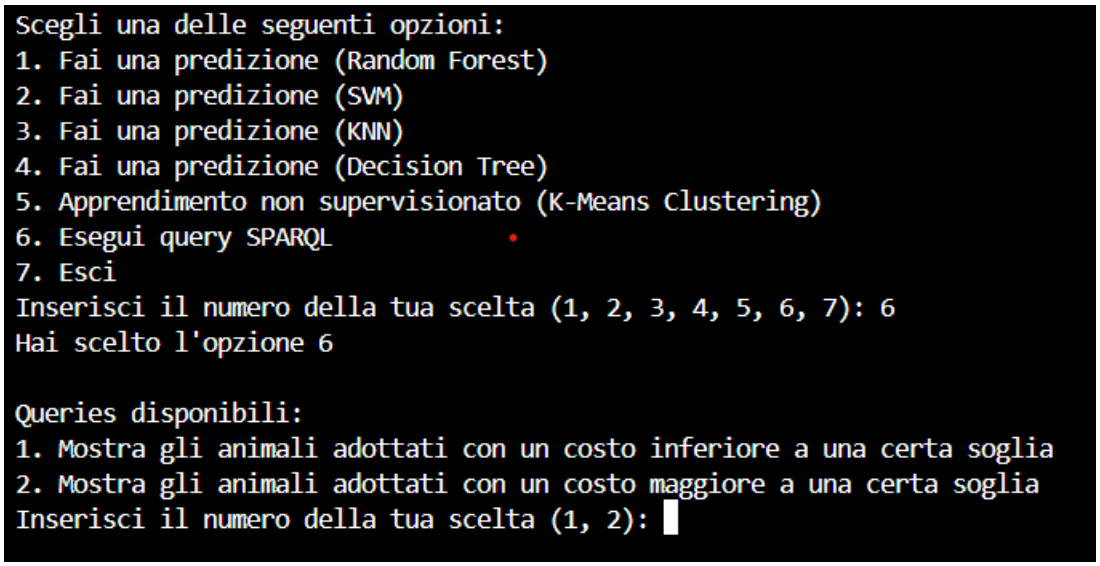
- load_onto(): Carica un'ontologia da un file RDF specificato. Utilizza la libreria owlready2 per caricare l'ontologia dal file ontologie.rdf situato nella cartella ./archivio/. L'ontologia rappresenta un insieme di concetti e relazioni, solitamente usata per modellare conoscenze in domini specifici.
- Create_ontology(): Crea un'ontologia da zero e la popola con dati dal dataset. Definisce le classi principali dell'ontologia: Animal, Shelter e Adoption. Crea proprietà di oggetto (relazioni tra classi) e proprietà di dati (attributi delle classi). Carica un dataset utilizzando la funzione ds.get_dataset() e popola le classi Animal, Shelter e Adoption con i dati rilevanti. Infine, salva l'ontologia creata nel file animal_adoption_ontology.rdf.
- Animals_high_cost(cost_threshold,csv_file): Filtra e visualizza gli animali che sono stati adottati con un costo maggiore della soglia specifica. Carica un file CSV e filtra i dati secondo la soglia specificata ed infine stampa i dettagli degli animali che hanno un costo maggiore della soglia specificata.
- Animals_low_cost(cost_threshold,csv_file): Filtra e visualizza gli animali che sono stati adottati con un costo inferiore della soglia specifica. Carica un file CSV e filtra i dati secondo la soglia specificata ed infine stampa i dettagli degli animali che hanno un costo inferiore della soglia specificata.

Caricamento e Creazione delle Ontologie

Le prime due funzioni (load_onto e create_ontology) si occupano della gestione dell'ontologia, ossia di caricarla e crearla a partire dai dati.

Interrogazione del Dataset

Le altre due funzioni (animals_high_cost, animals_low_cost) sono usate per interrogare il dataset e ottenere le informazioni specifiche sugli animali adottati secondo un costo che sia maggiore o minore della soglia specificata dall'utente.



Capitolo 5) Apprendimento Non Supervisionato

L’apprendimento non supervisionato è una branca dell’apprendimento automatico in cui l’agente viene addestrato su un insieme di dati senza etichette. Esistono due principali tipi di apprendimento non supervisionato:

- **Clustering** -> L’obiettivo è raggruppare gli elementi del dataset in base a delle somiglianze;
- **Riduzione della dimensionalità** -> L’obiettivo è ridurre il numero di feature utilizzate mantenendo però le informazioni più significative. Un esempio è la PCA (Principal Component Analysis).

Nel nostro caso, l'apprendimento non supervisionato con K-Means è stato utilizzato per raggruppare gli animali in base a dati come età, peso, tempo trascorso in rifugio, costo di adozione, stato vaccinale e condizioni di salute. L'obiettivo era raggruppare gli animali in cluster che condividono caratteristiche simili, senza bisogno di etichette predefinite. Questo approccio ha permesso di evidenziare pattern nascosti, come gruppi di animali con età e caratteristiche simili, che potrebbero avere un tasso di adozione simile.

I valori dei centroidi rappresentano il punto medio per ciascun cluster. Ogni cluster contiene animali con valori simili a questi centroidi. I centroidi indicano i valori medi delle caratteristiche per gli animali all’interno di ciascun gruppo.

Esistono due tipologie di apprendimento di clustering: **hard clustering** e **soft clustering**. Il primo associa ogni esempio a un cluster specifico, mentre il secondo associa a ogni esempio una probabilità di appartenenza a ciascun cluster.

Per la realizzazione di questa parte, è stata creata la classe “UnsupervisedLearning.py” e utilizzata la libreria **scikit-learn** per il clustering, la libreria **matplotlib** per la visualizzazione di grafici e la libreria **kneed** per individuare il gomito.

Nel nostro caso, abbiamo deciso di utilizzare l’algoritmo di **hard clustering** KMeans. Uno dei problemi principali del KMeans è trovare il numero ideale di cluster da fornire in input all’algoritmo. Per risolvere questo problema, abbiamo seguito una strategia nota come “**curva del gomito**”. La curva del gomito mostra la variazione dell’inertia al variare del numero di cluster, dove l'inertia rappresenta la somma delle distanze quadrate tra ogni punto dati e il centro del cluster assegnato. L’algoritmo prevede di eseguire il KMeans per diversi valori di numero di cluster, calcolare

l’inertia per ognuno di essi, plottare su un grafico la curva e poi identificare il gomito, ovvero il punto in cui la diminuzione dell’inertia diventa meno significativa.

Sviluppi Futuri

Per sviluppare ulteriormente il progetto, potremmo considerare di ottimizzare il modello di apprendimento automatico esplorando tecniche avanzate come la ricerca degli iperparametri e provando nuovi algoritmi di classificazione, come XGBoost. Un'altra direzione interessante è l'integrazione di dati aggiuntivi, come informazioni socio-economiche, per migliorare la previsione di adozione. Inoltre, potrebbe essere utile implementare un sistema per aggiornare i dati in tempo reale o su base regolare. Infine, potremmo ampliare le funzionalità di query, consentendo agli utenti di eseguire query più complesse e dettagliate sul database.

Riferimenti Bibliografici

Apprendimento supervisionato: Artificial Intelligence 3E (foundations of computational agents) di David L. Poole & Alan K. Mackworth. https://artint.info/3e/html/ArtInt3e.html

Rappresentazione Query SPARQL e Ontologie Artificial Intelligence 3E (foundations of computational agents) di David L. Poole & Alan K. Mackworth. https://artint.info/3e/html/ArtInt3e.html

Apprendimento non supervisionato: Artificial Intelligence 3E (foundations of computational agents) di David L. Poole & Alan K. Mackworth. https://artint.info/3e/html/ArtInt3e.html

Adoption Dataset: https://www.kaggle.com/datasets/rabieelkharoua/predict-pet-adoption-status-dataset/data