Architetture Dati

Progetto - Data Quality

Componenti del gruppo:

Bancora Davide - M.905588
Donato Benedetta - M.905338
Dubini Emanuele - M.904078

Corso di Laurea Magistrale in Informatica - LM 18
Anno Accademico 2022-2023

Indice

Elenco delle figure

Listings

1 Introduzione

1.1 Contesto

La *Data Quality*, nota anche come qualità dei dati, rappresenta un argomento di rilevanza cruciale nell'attuale era dell'informazione digitale.
L'aumento esponenziale del volume e della diversità dei dati disponibili ha reso imprescindibile garantire la qualità dei dati al fine di assicurare l'affidabilità e l'accuratezza delle analisi e dei modelli che ne dipendono.
Lo scopo della *Data Quality* riguarda l'insieme di pratiche mirate a verificare che i dati siano privi di errori, completi, coerenti, affidabili e tempestivi per evitare la presenza di valori mancanti, duplicati, inconsistenze, errori di formattazione e altre fonti di incertezza. Questi errori possono minare la fiducia e l'affidabilità delle informazioni ricavate dai dati stessi e possono condurre a decisioni errate o inefficaci.
Particolarmente rilevante è l'influenza che la qualità dei dati, presenti in un *dataset*, esercita sui modelli di *Machine Learning*; essi sono ampiamente adottati per estrarre informazioni significative dai dati stessi e per prendere decisioni basate su tali informazioni.
Tuttavia, se i dati utilizzati per addestrare gli algoritmi di apprendimento automatico risultano essere di scarsa qualità, si corre il rischio di ottenere risultati distorti o poco affidabili.
Pertanto, la qualità dei dati riveste un ruolo critico durante l'intero processo di addestramento di un modello di *Machine Learning*; durante l'acquisizione dei dati, la pulizia, l'integrazione, la trasformazione e l'addestramento del modello stesso, è necessario assicurare l'accuratezza e la rappresentazione corretta dei dati rispetto alla realtà che si intende

modellare.
Solo attraverso tale attenzione sarà possibile ottenere modelli di *Machine Learning* validi e performanti.
Il presente studio propone di esplorare l'effetto dei dati "sporchi" nel *dataset* sulle prestazioni degli algoritmi di *Machine Learning* scelti modificando i principali fattori che determinano o meno la cancellazione di una prenotazione di un soggiorno in un Hotel.
Attraverso l'introduzione intenzionale di dati "sporchi" nel *dataset*, l'obbiettivo sarà quello di valutare le ripercussioni sulle metriche prestazionali come *precision, recall, F1-measure, accuracy* e *AUC* dei modelli nel classificare correttamente le istanze positive "canceled".
L'intento finale del progetto è quello di comprendere come variano le metriche prestazionali dei modelli in seguito al deterioramento delle istanze presenti nel *dataset* e valutare quale tra i due risulta essere più robusto.

1.1.1 L'importanza dei dati

Come precedentemente citato, la qualità dei dati è un fattore critico per garantire risultati affidabili e decisioni fondate.
Il *dataset Hotel Reservations* offre un'opportunità per esplorare come la presenza di dati sporchi, come dati inconsistenti, valori nulli, *outliers* e inaccuratezze, possano influenzare le prestazioni di un algoritmo di apprendimento automatico come **l'Albero Decisionale** e la **Support Vector Machine**.
Comprendere l'impatto di tali problemi sulla qualità delle previsioni è essenziale per sviluppare strategie di gestione dei dati efficaci e migliorare la precisione delle analisi.

1.1.2 Metodologie utilizzate

Il seguente studio comprende diverse fasi significative: inizialmente è stato analizzato il *dataset* pulito creando grafici a supporto della visualizzazione di correlazioni tra le caratteristiche degli attributi.
Tramite il *dataset* pulito, è stata verificata la presenza di accuratezza, inconsistenza, coerenza e integrità nei dati.
Successivamente, sono stati applicati gli algoritmi Albero Decisionale e Support Vector Machine e sono state misurate le prestazioni utilizzando diverse metriche di valutazione come *precision*, *recall*, *F1-measure*, *accuracy* e *AUC*.
Tali metriche forniscono una misura quantitativa delle prestazioni dell'algoritmo in termini di correttezza delle previsioni e capacità di discriminazione tra le classi *target*.
Una volta salvate le prestazioni ottenute con gli algoritmi di apprendimento applicati al *dataset* pulito, è stato sporcato tale *dataset* tramite lo sviluppo di metodi per il deterioramento delle istanze come: l'aggiunta di valori nulli, *outliers*, inconsistenza, righe duplicate e inaccuratezza.

Questo secondo *dataset* è stato creato per confrontare le *performance* ottenute con i modelli di *Machine Learning* addestrati sul *dataset* sporco, rispetto alle misure ottenute con il *dataset* pulito.
Confrontando le prestazioni tra i due *subset*, siamo stati in grado di valutare se, e in che misura, l'introduzione di sporco influisce sulle prestazioni degli algoritmi di apprendimento.

1.1.3 Requisiti per l'esecuzione del programma

Per potere eseguire correttamente il programma, è necessario scaricare e installare python nella versione 3.10.9 dal link `https://www.python.org/downloads/` e configurare correttamente le variabili d'ambiente per il corretto funzionamento di quest'ultimo.
Il progetto è disponibile tramite il seguente link GitHub:
`https://github.com/dbancora/ProgettoArchitetturaDati_DataQuality`
Una volta scaricato il progetto è necessario configurare correttamente l'interprete python installato all'interno del PC in uso.
Successivamente, è necessario installare le librerie indicate nel file `main.py` tramite i seguenti comandi da inserire nel terminale:

```
1 pip install pandas
2 pip install scikit-learn
3 pip install matplotlib
4 pip install seaborn
```

Listing 1.1: Installazione di librerie

2 Dataset

2.1 Descrizione del dataset

Il *dataset* utilizzato in questo progetto, disponibile al seguente link: `https://www.kaggle.com/datasets/ahsan81/hotel-reservations-classification-dataset`, è composto da 19 attributi riguardanti i principali fattori che possano determinare o meno la cancellazione di una prenotazione di un soggiorno in un Hotel.
La struttura del *dataset* utilizzato è descritta dall'elenco seguente:

1. **Booking_ID**: identificativo univoco di ogni prenotazione.
2. **no_of_adults**: numero di adulti presenti nella prenotazione.
3. **no_of_children**: numero di bambini presenti nella prenotazione.
4. **no_of_weekend_nights**: numero di notti del fine settimana previste dalla prenotazione (sabato - domenica).
5. **no_of_week_nights**: numero di notti settimanali previste dalla prenotazione (lunedì - venerdì).
6. **type_of_meal_plan**: tipologia di piano pasti prenotato dal cliente.
7. **required_car_parking_space**: se è stato richiesto un posto auto durante la prenotazione.

8. **room_type_reserved**: tipologia di camera prenotata dal cliente.

9. **lead_time**: tempo trascorso tra la prenotazione e l'arrivo nella struttura.

10. **arrival_year**: anno di inizio soggiorno.

11. **arrival_month**: mese in cui inizia il soggiorno.

12. **arrival_date**: giorno in cui inizia il soggiorno.

13. **market_segment_type**: il segmento di mercato da cui proviene la prenotazione (Aviation, Complementary, Corporate, Offline, Online).

14. **repeated_guest**: se il cliente è già stato un ospite nella struttura in precedenza o meno.

15. **no_of_previous_cancellations**: numero di prenotazioni precedenti che sono state annullate dal cliente.

16. **no_of_previous_bookings_not_canceled**: numero di prenotazioni precedenti che non sono state annullate dal cliente.

17. **avg_price_per_room**: costo medio giornaliero della camera (espresso in euro).

18. **no_of_special_requests**: numero di richieste "speciali" desiderate dal cliente (es. piano alto, vista dalla camera, ecc.).

19. **booking_status**: indica se la prenotazione è stata annullata o non cancellata.

2.2 Analisi degli attributi

In questa sezione, vengono riportate le analisi esplorative eseguite sul *dataset*.
All'interno del file `graphs_gen.py` è possibile trovare il codice relativo ai grafici che sono stati utilizzati per dedurre le considerazioni di seguito presentate.

La prima analisi effettuata riguarda la variabile *target*; è possibile, infatti, notare che la maggior parte delle prenotazioni non verranno cancellate: 2.1

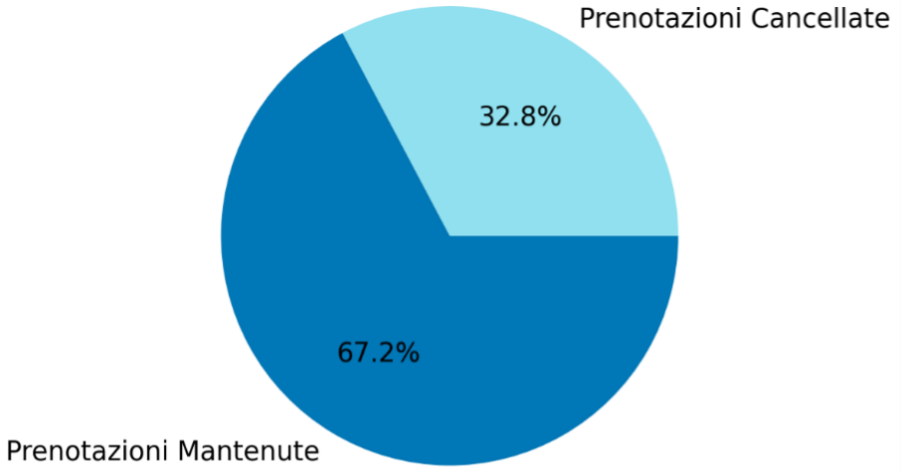


Figura 2.1: Distribuzione dell'attributo target.

La successiva analisi, si è focalizzata sul numero di adulti e di bambini presenti nelle istanze: è stato osservato che la maggior parte delle prenotazioni comprende unicamente una coppia di adulti, come si può notare dal grafico riportato di seguito: 2.2

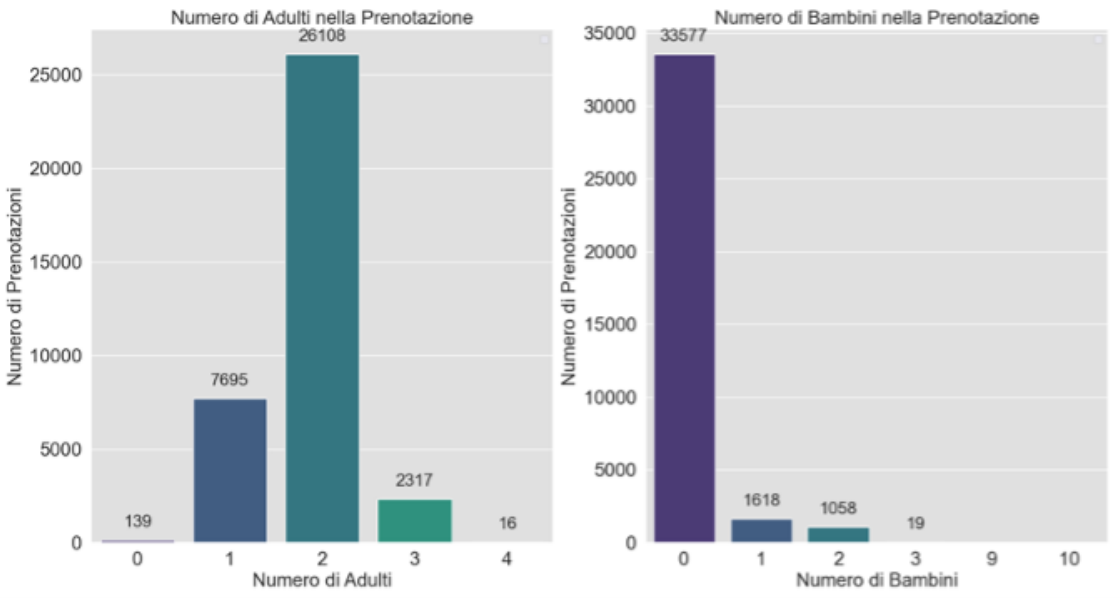


Figura 2.2: Distribuzione degli attributi no_of_children e no_of_adults.

Il grafico successivo è stato creato per determinare se la presenza di bambini nelle prenotazioni avesse un impatto sulla probabilità di cancellazione della stessa.
Tuttavia, dopo l'analisi, è stato osservato che la presenza di bambini non influisce sulla probabilità di cancellazione delle prenotazioni, come rappresentato dalla seguente immagine 2.3:

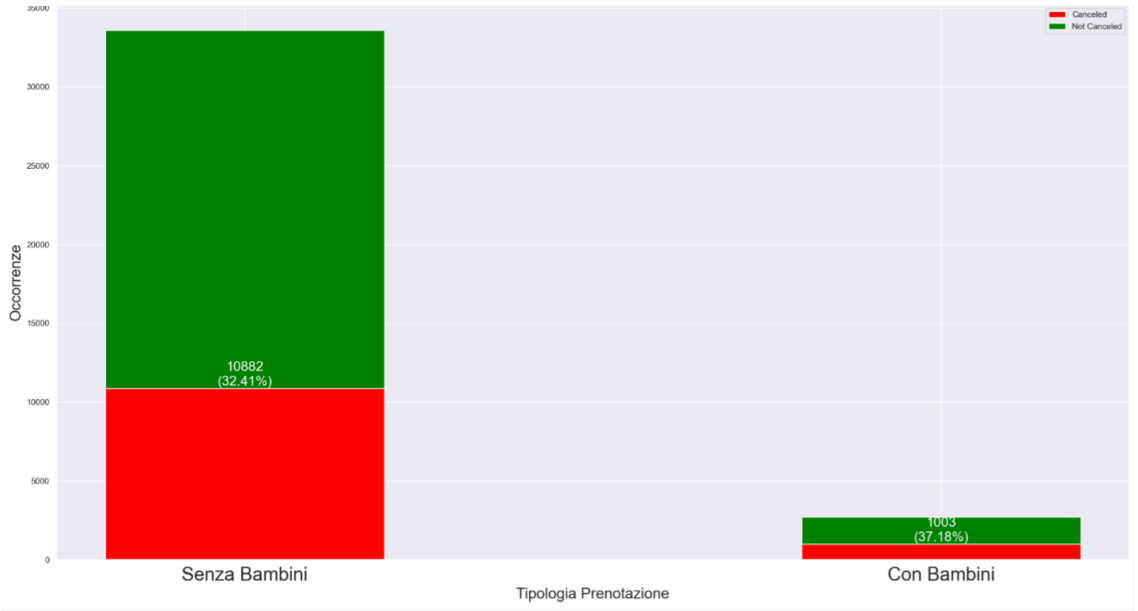


Figura 2.3: Confronto delle cancellazioni e delle prenotazioni in presenza o meno di bambini.

Con le successive analisi è stata osservata una permanenza media superiore a una notte per oltre il 50% delle prenotazioni effettuate.

Proseguendo con l'analisi degli attributi, é stato preso in considerazione l'attributo required_car_parking_space che indica la richiesta di avere o meno il parcheggio riservato alla vettura dell'ospite: la maggior parte delle istanze assume valore pari a zero.
Questo valore indica che il cliente non ha richiesto il posto auto durante la fase di prenotazione del soggiorno oppure, possiamo supporre che alcune strutture alberghiere non dispongano di parcheggi riservati agli ospiti e di conseguenza il valore rimane pari a zero.
Questo risultato é visibile tramite il seguente grafico: 2.4

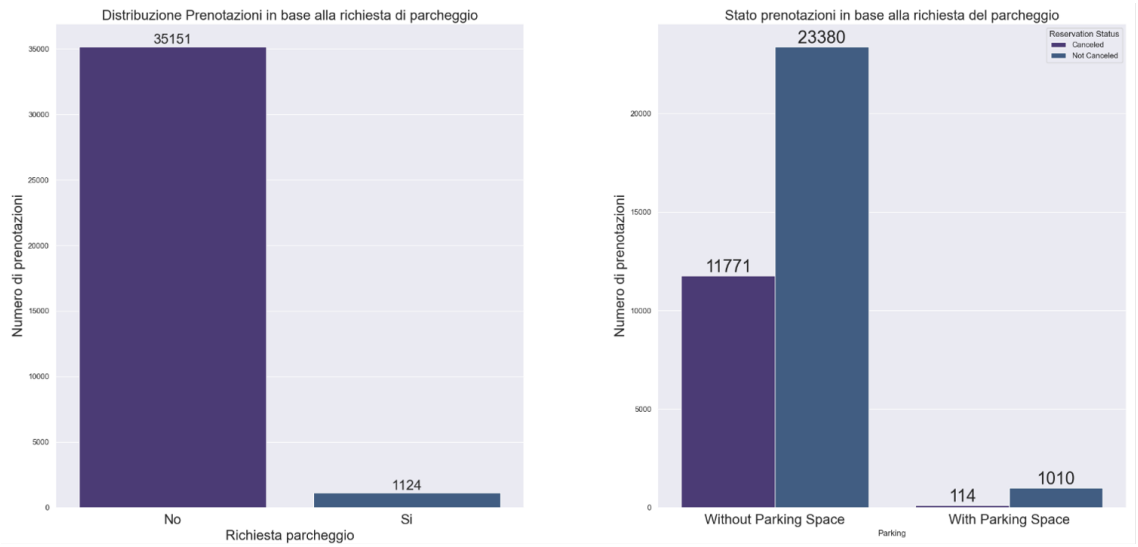


Figura 2.4: Confronto delle prenotazioni che presentano o meno la richiesta di parcheggio.

Il grafico successivo mostra che le prenotazioni di tipo *Complementary* hanno una cancellazione pari allo 0%. Questo avviene perché sono prenotazioni offerte gratuitamente al cliente da parte della struttura alberghiera per promuoverla o per premiare i clienti fedeli e, difficilmente, il cliente le cancellerà. Anche le cancellazioni delle prenotazioni *Corporate*, ossia quelle effettuate dalle aziende per i propri dipendenti, hanno un tasso di cancellazione molto basso, come si evince dalla figura 2.5:

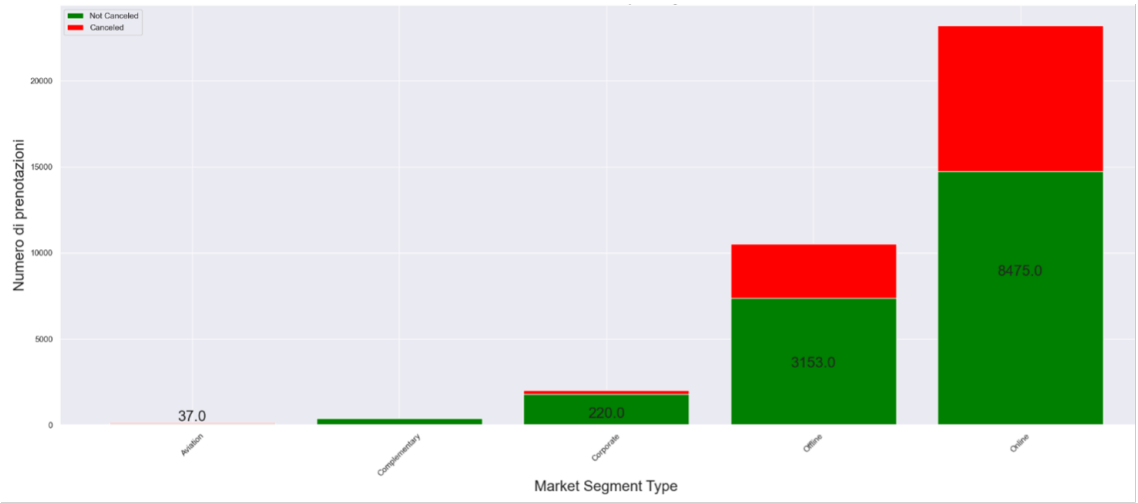


Figura 2.5: Numero delle prenotazioni cancellate in base al market_segment_type.

Un'importante osservazione ottenuta analizzando la relazione tra la variabile no_of_special_request e l'attributo *target*, riguarda la percentuale di cancellazione delle prenotazioni: al crescere delle richieste speciali diminuisce la probabilità che il cliente cancelli la prenotazione (pari a 0% quando il cliente inserisce 3 o piu richieste speciali); concludiamo quindi che i due attributi sono inversamente proporzionali tra loro, come rappresentato dalla figura2.5:

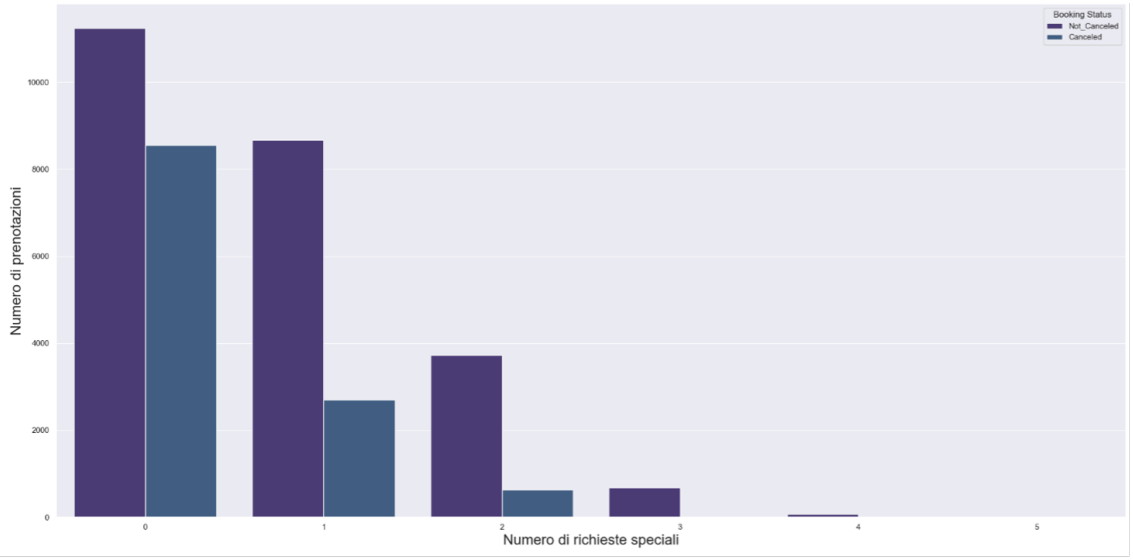


Figura 2.6: Prenotazioni cancellate/non cancellate in base al numero di richieste speciali.

All'interno della relazione sono riportati solo alcuni grafici; in particolare quelli che sono risultati più significativi da una prima analisi esplorativa.
Nel codice è comunque presente l'opportunità di generare ulteriori grafici che analizzano altri attributi.

3
Data Quality

3.1 Metriche della Data Quality

Nel contesto di una gestione efficace dei dati, un aspetto fondamentale è la valutazione della qualità dei dati attraverso l'uso di apposite metriche.
L'utilizzo di queste metriche fornisce un quadro completo della qualità dei dati presenti nel *dataset* e consente di identificare aree di miglioramento per garantire l'affidabilità dei modelli di *Machine Learning*.
Di seguito verranno illustrate le principali misure da noi implementate:

- **Consistenza**: si riferisce alla coerenza dei dati all'interno di un *dataset*.
  La consistenza dei dati implica che le informazioni siano in accordo tra loro e non contengano contraddizioni o incongruenze.
  Nel momento in cui i dati presentano discrepanze o contraddizioni, si potrebbero ottenere risultati inaccurati o incoerenti nel trarre conclusioni o prendere decisioni basate su tali dati.
  Inoltre, la consistenza dei dati è essenziale per garantire l'integrità delle relazioni e delle dipendenze tra le diverse entità o attributi presenti nel *dataset*.

- **Coerenza**: si riferisce alla misura in cui i dati sono coerenti e allineati tra diverse fonti o all'interno di un singolo *dataset*.
  Valutare la coerenza dei dati è cruciale per garantire l'integrità delle informazioni e per evitare discrepanze o incongruenze che potrebbero influenzare negativamente sull'addestramento dei modelli di *Machine Learning*.

Le misure di coerenza possono includere la verifica dell'allineamento dei dati nei vari attributi del *dataset* e la conformità alle regole di integrità stabilite.
L'identificazione di incoerenze e la loro correzione tempestiva sono essenziali per mantenere la coerenza dei dati e preservare la qualità complessiva del *dataset*.

- **Accuratezza**: si riferisce alla misura di quanto i dati riflettono con precisione la realtà che intendono rappresentare.
  È fondamentale verificare l'accuratezza dei dati presenti in un *dataset* poiché l'utilizzo di dati inaccurati può condurre a decisioni errate.
  Nel momento in cui le informazioni contenute nel *dataset* sono inaffidabili o imprecise, qualsiasi analisi o modello basato su di esse produrrà risultati distorti e non rappresenterà correttamente la realtà.
  L'accuratezza dei dati è cruciale per addestrare modelli di *Machine Learning* validi e performanti in quanto essi rispecchieranno la qualità dei dati utilizzati per l'addestramento.
  Con valori inaffidabili o poco precisi, i risultati ottenuti dal modello saranno altrettanto imprecisi.

- **Integrità**: l'integrità dei dati in un dataset si riferisce alla loro qualità e coerenza. Indica la condizione in cui i dati sono accurati, completi e affidabili, senza errori o inconsistenze.
  Questa misura è fondamentale per garantire che le informazioni contenute nel dataset siano corrette e rappresentino in modo fedele la realtà che i dati stessi intendono rappresentare.

La scelta di verificare queste metriche di qualità dipende dal contesto e dagli obiettivi o dell'analisi che si sta affrontando.
Le metriche sopra riportate saranno quelle da noi considerate per determinare la qualità dei dati utilizzati per l'addestramento dei modelli di *Machine Learning* durante lo svolgimento delle varie prove nel progetto.

3.2 Implementazione di ciascuna metrica

In questa sezione, verranno elencate le metriche verificate e i relativi metodi implementati nel progetto.
Per favorire una maggior chiarezza e semplicità, sono elencati tutti i metodi implementati e, per quelli più rilevanti ai fini del progetto, è presente il codice in relazione.
Il codice completo dei metodi utilizzati per la valutazione delle metriche di *Data Quality* è contenuto all'interno del file `DataQualityEvaluation.py` reperibile al seguente link di GitHub: `https://github.com/dbancora/ProgettoArchitetturaDati_DataQuality/blob/main/dataQualityEvaluation.py` ed inoltre, i risultati ottenuti applicati al dataset sono

disponibili all'interno del file `DataQuality_DatasetPulito.txt` reperibile al link `https://github.com/dbancora/ProgettoArchitetturaDati_DataQuality/blob/main/Data%20Quality/DataQuality_DatasetPulito.txt`

3.2.1 Consistenza

La funzione `check_adults_children_consistency` verifica la coerenza degli attributi `no_of_adults` e `no_of_children` all'interno del *dataset*.
L'esecuzione di questo codice permette di identificare e gestire le incongruenze negli attributi specificati, contribuendo a verificare la qualità e all'accuratezza complessiva dell'analisi dei dati. Il codice verifica che i valori degli attributi `no_of_adults` e `no_of_children` siano non negativi, in caso contrario verrà visualizzato un messaggio di notifica.
Inoltre, questa funzione verifica la relazione logica presente tra il numero di adulti e il numero di bambini.
In particolare, identifica le istanze del *dataset* in cui il numero di adulti è pari a zero mentre il numero di bambini è maggiore di zero.
Tale combinazione, ai fini del progetto, è considerata inconsistente, poiché la presenza di soli bambini senza adulti non dovrebbe essere resa possibile dal sistema di prenotazioni presso le strutture alberghiere.
Testando questa funzione, applicata al *dataset* pulito, abbiamo verificato la presenza di una minima parte di istanze appartenenti al *dataset* in cui era presente questo tipo di inconsistenza.

```
def check_adults_children_consistency(data):
    array1 = ["no_of_adults", "no_of_children"]
    if (data['no_of_adults']>=0).any() or (data['no_of_children']>=0).any():
        print("I valori di no_of_adults e no_of_children sono
            non negativi.")
    else:
        print("Errore: I valori di no_of_adults e no_of_children devono
            essere non negativi.")
    column_array = ["no_of_adults", "no_of_children"]
    values_array = [0, 2]
    invalid_relation = (data['no_of_adults'] == 0) &
        (data['no_of_children'] > 0)
    if invalid_relation.any():
        print("Errore: Ci sono righe con no_of_adults pari a 0 e
            no_of_children maggiore di 0.")
        first_invalid_row = (data[invalid_relation].iloc[0])
        no_of_adults_value = first_invalid_row['no_of_adults']
        no_of_children_value = first_invalid_row['no_of_children']
        values_array = [no_of_adults_value, no_of_children_value]
    return column_array, values_array
```

Listing 3.1: Funzione check_adults_children_consistency

La funzione `check_nights_consistency` esegue le verifiche sulla coerenza dei dati relativi al numero di notti trascorse durante un soggiorno controllando la presenza di valori negativi all'interno degli attributi `no_of_weekend_nights` e `no_of_week_nights`.
Nel caso in cui venisse trovato almeno un valore negativo in uno di questi, viene stampato un messaggio di notifica.

```
def check_nights_consistency(data):
    if (data['no_of_weekend_nights'] < 0).any() or
        (data['no_of_week_nights'] < 0).any():
        print("Errore: I valori di no_of_weekend_nights e
            no_of_week_nights devono essere non negativi.")
```

Listing 3.2: Funzione check_nights_consistency

La funzione `check_car_parking_space_consistency` verifica la coerenza dei valori presenti nell'attributo `required_car_parking_space` controllando che siano presenti solo valori pari a 0 oppure 1.
Nel caso in cui si manifestino valori anomali, viene stampato un messaggio di notifica stampando a console le righe non valide.
In presenza di soli valori accettabili verrà visualizzato un messaggio che conferma la non presenza di incongruenze nell'attributo relativo alla richiesta di parcheggio.

```
def check_car_parking_space_consistency(data):
    invalid_values = (data['required_car_parking_space'] != 0) &
        (data['required_car_parking_space'] != 1)
    if invalid_values.any():
        print("Errore: I valori di required_car_parking_space devono essere
            0 o 1.")
        print("Righe non valide:")
        print(data[invalid_values])
    else:
        print("[ATTRIBUTO REQUIRED_PARKING_SPACE]: Non presenta
            inconsistenza.")
```

Listing 3.3: Funzione check_car_parking_space_consistency

`check_lead_time_consistency` è una funzione che verifica la presenza di valori negativi nella colonna `lead_time` del *dataset* utilizzato nel progetto e in caso affermativo stampa un messaggio di notifica.
Senza la presenza di valori nulli viene stampato un messaggio che indica l'assenza di inconsistenze nell'attributo.
Questo controllo è utile per garantire che tali valori siano coerenti e rispettino le regole o le condizioni di validità desiderate.

La funzione check_special_requests_consistency verifica se ci sono valori negativi nella colonna `no_of_special_requests` del *dataset* considerato.
Se anche solo uno dei valori è negativo, viene stampato un messaggio di errore che indica che i valori di `no_of_special_requests` devono essere positivi.
In caso contrario, viene stampato un messaggio che indica l'assenza di incongruenze nell'attributo `no_of_special_requests`.

check_booking_status_consistency è la funzione che controlla la presenza di valori diversi da "Canceled" o "Not_Canceled" nella colonna `booking_status` del *dataset*.
Nel caso in cui venisse rilevato almeno un valore non valido, la funzione stampa un messaggio di notifica indicando che i valori dell'attributo non corrispondono a quelli previsti, stampando le istanze del *dataset* che non rispettano il vincolo specificato.
Se non viene rilevata alcuna anomalia nei valori dell'attributo viene stampato un messaggio che indica l'assenza di valori non accettabili.

La funzione check_avg_price_consistency verifica la coerenza dei valori presenti nella colonna `avg_price_per_room`.
Il controllo consiste nel verificare l'esistenza di valori negativi all'interno della colonna considerata e nel caso in cui venga rilevato almeno un valore negativo, sarà emesso un messaggio di notifica.
In caso contrario, viene stampata una conferma che indica l'assenza di valori negativi verificando l'assenza di incongruenze nell'attributo `avg_price_per_room`.

La funzione check_arrival_date_consistency svolge un controllo sulla coerenza dei valori assunti dagli attributi `arrival_year`, `arrival_month` e `arrival_date`.
L'obiettivo è garantire che i valori di tali attributi siano nel formato corretto e rappresentino date valide.
Questo controllo viene effettuato utilizzando espressioni regolari per confrontare i valori letti con *pattern* specifici.
Se almeno uno dei valori non rispetta il formato corretto, viene stampato un messaggio di notifica insieme alle righe non valide.
Inoltre, viene effettuato un controllo per verificare se i valori degli attributi corrispondano a date valide. E nel caso in cui venga lanciata l'eccezione "ValueError" l'indice della riga viene aggiunto a una lista che contiene le date non valide.
Successivamente, verrà stampato un messaggio di notifica che contiene gli indici delle righe non valide con le annesse date errate.

```
1 def check_arrival_date_consistency(data):
2     invalid_format=~data['arrival_year'].astype(str).str.match(r'^\d{4}$')|
3                    ~data['arrival_month'].astype(str).str.match(r'^\d{2}$')|
4                    ~data['arrival_date'].astype(str).str.match(r'^\d{2}$')
```

```
5
6       if invalid_format.any():
7           print(
8               "Errore: I valori di arrival_year, arrival_month e arrival_date
9                   devono essere nel formato corretto.")
10          print("Righe non valide:")
11          print(data[invalid_format])
12
13      invalid_dates = []
14      for idx, row in data.iterrows():
15          try:
16              year = int(row['arrival_year'])
17              month = int(row['arrival_month'])
18              day = int(row['arrival_date'])
19              datetime.datetime(year, month, day)
20          except ValueError:
21              invalid_dates.append(idx)
22      if len(invalid_dates) > 0:
23          print(
24              "Errore: Alcuni valori di arrival_year, arrival_month e
25                  arrival_date non corrispondono a date valide.")
26          print("Righe non valide:")
27          print(data.loc[invalid_dates])
```

Listing 3.4: Funzione check_arrival_date_consistency

Testando questa funzione, applicata al *dataset* pulito, abbiamo verificato la presenza di 37 istanze del *dataset* in cui era presente la data della prenotazione non valida in quanto corrispondeva al 29/02/2018 oppure 29/02/2017 (anni non bisestili).

3.2.2 Coerenza

La metrica riguardante la coerenza, viene verificata tramite il codice riportato di seguito sfruttando una serie di stampe che analizzano i valori *unique* presenti in determinate colonne di un *dataset*.
I controlli vengono effettuate per i seguenti attributi:

- **no_of_adults**: Stampa i valori unici presenti nella colonna ”no_of_adults”;
- **no_of_children**: Stampa i valori unici presenti nella colonna ”no_of_children”;
- **arrival_year**: Stampa i valori unici presenti nella colonna ”arrival_year”;
- **arrival_month**: Stampa i valori unici presenti nella colonna ”arrival_month”;
- **arrival_date**: Stampa i valori unici presenti nella colonna ”arrival_date”;
- **booking_status**: Stampa i valori unici presenti nella colonna ”booking_status”.

Tale codice è utile per fornire una panoramica dei valori distinti presenti in questi attributi aiutando a identificare eventuali valori anomali, valori mancanti o variazioni nei dati.

```
def unique_value(df):
    # Verifica i valori nelle colonne no_of_adults e no_of_children
    print_to_console_and_file("VALORI NELL'ATTRIBUTO no_of_adults:")
    print_to_console_and_file(df['no_of_adults'].unique())

    print_to_console_and_file("VALORI NELL'ATTRIBUTO no_of_children:")
    print_to_console_and_file(df['no_of_children'].unique())

    # Verifica i valori nelle colonne arrival_year, arrival_month e
    arrival_date
    print_to_console_and_file("VALORI NELL'ATTRIBUTO arrival_year:")
    print_to_console_and_file(df['arrival_year'].unique())

    print_to_console_and_file("VALORI NELL'ATTRIBUTO arrival_month:")
    print_to_console_and_file(df['arrival_month'].unique())

    print_to_console_and_file("VALORI NELL'ATTRIBUTO arrival_date:")
    print_to_console_and_file(df['arrival_date'].unique())

    # Verifica i valori nella colonna booking_status
    print_to_console_and_file("VALORI NELL'ATTRIBUTO booking_status:")
    print_to_console_and_file(df['booking_status'].unique())
```

Listing 3.5: Funzione unique_value

3.2.3 Accuratezza

Per la metrica di accuratezza viene definita una funzione chiamata `check_column_types` che riceve in input il set di dati da analizzare.
La funzione esegue le seguenti operazioni:

- Itera su tutte le colonne del *dataset* e, per ciascuna di queste, ricava il nome della colonna (column_name) e i suoi dati (column_data).
- Verifica il tipo di dati della colonna utilizzando l'attributo dtype; a seconda del valore ottenuto, esegue le seguenti azioni:
  - Se i dati sono di tipo `object`, la funzione verifica se tutti i valori nella colonna sono stringhe usando la funzione `apply()` e la funzione `isinstance(x, str)`. Se tutti i valori sono corretti, viene stampato un messaggio che indica la correttezza di tutti i valori. In caso contrario, viene stampato un messaggio di errore.

- Con dati di tipo `int64`, la funzione verifica se tutti i valori nella colonna sono interi usando la funzione `apply()` insieme alla funzione `isinstance(x, int)` e stampa un messaggio di conferma nel caso in cui tutti i valori siano corretti. Altrimenti, viene visualizzato nella console un messaggio di errore.
- Se i dati sono di tipo `float64`, la funzione verifica che tutti i valori presenti nell'attributo siano float usando la funzione `apply()` insieme alla funzione `isinstance(x, float)` e, in caso affermativo, viene stampato un messaggio di conferma. Altrimenti, viene stampato un messaggio di errore indicando la presenza contemporanea di diversi tipi di dati nella colonna.
- Se il tipo di dati restituito non rientra in nessuna delle precedenti categorie, viene stampato un messaggio di errore generico.

Al termine delle iterazioni, nel caso in cui non si siano verificati errori precedentemente, viene stampato un messaggio indicante che tutte le colonne contengono solo valori del tipo corretto restituendo il valore booleano `True`. Di seguito riportiamo il codice:

```
def check_column_types(df):
    for column_name, column_data in df.items():
        if column_data.dtype == 'object':
            if column_data.apply(lambda x: isinstance(x, str)).all():
                print(f"Tutti i valori nella colonna {column_name} sono
                    di tipo stringa.")
            else:
                print(f"Errore: La colonna {column_name} contiene diversi
                    tipi di dati.")
        elif column_data.dtype == 'int64':
            if column_data.apply(lambda x: isinstance(x, int)).all():
                print(f"Tutti i valori nella colonna {column_name} sono di
                    tipo intero.")
            else:
                print(f"Errore: La colonna {column_name} contiene diversi
                    tipi di dati.")
        elif column_data.dtype == 'float64':
            if column_data.apply(lambda x: isinstance(x, float)).all():
                print(f"Tutti i valori nella colonna {column_name} sono di
                    tipo float.")
            else:
                print(f"Errore: La colonna {column_name} contiene diversi
                    tipi di dati.")
        else:
            print(f"Errore: Tipo di dati non gestito per la colonna
                {column_name}.")
    print("Tutte le colonne contengono solo valori del tipo corretto.")
    return True
```

Listing 3.6: Funzione check_column_types

3.2.4 Integrità

La metrica di integrità viene verificata utilizzando la funzione `check_null_values` che esegue un'analisi dei valori mancanti (o nulli) nel *dataset*.

```
def check_null_values(df):
    missing_values = df.isnull().sum()

    missing_percentage = (missing_values / len(df)) * 100

    print_to_console_and_file("\n--- Valori mancanti per colonna:")
    print_to_console_and_file(missing_values)
    print_to_console_and_file("\n--- Percentuale di valori mancanti per
        colonna:")
    print_to_console_and_file(missing_percentage)
```

Listing 3.7: Funzione check_null_values

Tale funzione, inoltre, calcola il numero e la percentuale di valori mancanti per ciascuna colonna che vengono stampati sulla *console*, mostrando il nome della colonna seguito dalla percentuale di valori mancanti corrispondente.
Questo controllo può fornire una panoramica della completezza dei dati, permettendo di decidere come gestire i valori mancanti durante l'elaborazione o l'analisi dei dati.

La funzione `check_duplicates` identifica il numero di righe duplicate.
Questo può essere utile per verificare la presenza di duplicati nel *dataset* per poi decidere come gestirli, ad esempio, rimuovendo le righe duplicate o elaborando i dati in modo appropriato.

```
def check_duplicates(df):
    duplicated_rows = df.duplicated().sum()
    print_to_console_and_file("\n--- Numero di righe duplicate:")
    print_to_console_and_file(duplicated_rows)
```

Listing 3.8: Funzione check_duplicates

Successivamente è stata verificata l'integrità per l'attributo `avg_price_per_room`.
Tale attributo rappresenta il costo medio giornaliero delle camere e può essere considerato un fattore cruciale nell'analisi del settore alberghiero.
Per valutare l'integrità dei dati, è stata utilizzata la deviazione standard come misura di dispersione dei valori rispetto alla media, consentendo di identificare la variabilità dei prezzi delle camere all'interno del *dataset*.
È stata inoltre definita una soglia per identificare i valori *outliers*: la soglia è stata fissata a 3 volte la deviazione standard al di sopra e al di sotto della media (non andando sotto il valore zero).

Questa scelta si basa sul fatto che valori significativamente discostati dalla media, potrebbero essere considerati come possibili *outliers*.
Tramite l'applicazione delle misure prima citate, sono stati identificati i valori "anomali" nella colonna `avg_price_per_room`, che potrebbero causare situazioni particolari o errori nei dati.
Di seguito riportiamo il codice:

```
std_dev = df['avg_price_per_room'].std()
upper_threshold = df['avg_price_per_room'].mean() + (3 * std_dev)
mean = df['no_of_children'].mean()
lower_threshold = max(mean - (3 * std_dev), 0)
anomalous_values = df[(df['avg_price_per_room'] > upper_threshold) |
    (df['avg_price_per_room'] < lower_threshold)]
print_to_console_and_file("\n--- Valori outlier per l'attributo
    avg_price_per_room:")
print_to_console_and_file(anomalous_values)
```

Listing 3.9: Verifica dell'integrità per la colonna avg_price_per_room all'interno della funzione def check_integrity_attributes

Successivamente, è stata valutata l'integrità della colonna `market_segment_type`.
L'attributo rappresenta il segmento di mercato da cui proviene una prenotazione, fattore critico nell'analisi e nel comportamento dei clienti nonché determinante per la probabilità di cancellare o meno una prenotazione, come precedentemente illustrato. 2.2
Per valutare l'integrità dei dati, è stata calcolata la frequenza dei diversi valori presenti nell'attributo. La frequenza fornisce una visione delle distribuzioni relative dei diversi segmenti di mercato all'interno del *dataset*.
Successivamente, è stata definita una soglia per identificare i valori *outliers* nell'attributo: questa è stata stabilita minore dell'1% delle occorrenze totali nel *dataset*.
La scelta è basata sull'idea che i valori con una frequenza inferiore alla soglia, possono essere considerati rari o insoliti all'interno del contesto del *dataset* in esame.
Attraverso l'applicazione di queste misure, sono stati identificati i valori anomali nell'attributo `market_segment_type`. Di seguito, è riportato il codice:

```
segment_counts = df['market_segment_type'].value_counts()
threshold = len(df) * 0.01
print_to_console_and_file(threshold)
anomalous_values = segment_counts[segment_counts < threshold]
anomalous_rows = df[df['market_segment_type'].isin(anomalous_values.index)]
print_to_console_and_file("Righe con valori anomali nell'attributo
    market_segment_type:")
print_to_console_and_file(anomalous_rows['market_segment_type'])
```

Listing 3.10: Verifica dell'integrità per la colonna market_segment_type all'interno della funzione def check_integrity_attributes

Per valutare l'integrità dell'attributo `no_of_children`, è stata utilizzata la deviazione standard come misura di dispersione dei valori rispetto alla media, che permette di comprendere la variabilità dei numeri di bambini presenti nelle prenotazioni all'interno del *dataset*.
Successivamente, è stata calcolata una soglia per identificare i valori anomali nell'attributo `no_of_children`, stabilita come 5 volte la deviazione standard sopra la media.
Questa scelta è basata sul fatto che, impostando una soglia di 3 volte superiore alla deviazione standard, i valori restituiti come *"Upper" outliers* erano pari a 2: questo valore, in tale dominio, sembra essere un valore del tutto normale e lecito, infatti, la maggior parte delle prenotazioni sono composte da un numero pari a 2 di bambini.
La soglia pari al valore 5 era l'unica che permetteva di aumentare leggermente tale valore.
Inoltre, viene calcolata una soglia *"Lower"* per assicurare che non vi siano valori negativi o non validi per il numero di bambini.
Infine, vengono visualizzati i valori *outliers* nell'attributo `no_of_children`.
Il codice di seguito riporta quanto descritto:

```
std_dev = df['no_of_children'].std()
upper_threshold = df['no_of_children'].mean() + (5 * std_dev)
mean = df['no_of_children'].mean()
lower_threshold = max(mean - (5 * std_dev), 0)
anomalous_values = df[(df['no_of_children'] > upper_threshold) |
    (df['no_of_children'] < lower_threshold)]
print_to_console_and_file("\n--- Valori outlier per l'attributo
    no_of_children:")
print_to_console_and_file(anomalous_values)
```

Listing 3.11: Verifica dell'integrità per la colonna no_of_children all'interno della funzione def check_integrity_attributes

Lo stesso procedimento è stato implementato per valutare l'integrità dell'attributo `no_of_adults`.
La differenza rispetto al codice fornito in precedenza risiede nel fatto che la soglia è stata impostata come 3 volte la deviazione standard sopra la media.

In conclusione, l'analisi basata sulla deviazione standard e l'identificazione di valori *outlier* ci ha permesso di valutare anche l'integrità dell'attributo `no_of_special_requests` del *dataset*.
La soglia per identificare gli *outliers* è stata impostata, come per la media, con meno 3 volte la deviazione standard e un valore minimo di 0 per evitare valori negativi.

```
std_dev = df['no_of_special_requests'].std()
upper_threshold = df['no_of_special_requests'].mean() + (3 * std_dev)
mean = df['no_of_special_requests'].mean()
lower_threshold = max(mean - (3 * std_dev), 0)
```

```
anomalous_values = df[
    (df['no_of_special_requests'] > upper_threshold) |
        (df['no_of_special_requests'] < lower_threshold)]
print_to_console_and_file("\n--- Valori outlier per l'attributo
    no_of_special_requests:")
print_to_console_and_file(anomalous_values)
```

Listing 3.12: Verifica dell'integrità per la colonna no_of_special_requests all'interno della funzione def check_integrity_attributes

4
Modelli di Classificazione

4.1 Descrizione dei modelli di Machine Learning scelti

In questo studio sono stati implementati due differenti modelli di *Machine Learning*, per la classificazione delle istanze, con lo scopo di descriverne il comportamento assunto in seguito al processo di addestramento utilizzando le istanze di un *trainset* "sporco".

4.1.1 Decision Tree

Il primo modello implementato è rappresentato degli alberi decisionali, è un modello predittivo ampiamente utilizzato nell'apprendimento automatico per problemi di classificazione e regressione. Offrono una rappresentazione intuitiva e comprensibile dei risultati e delle decisioni intraprese dal modello stesso.
La loro natura flessibile permette di elaborare sia dati numerici che categorici (dopo averli espressamente gestiti tramite *one-hot-encoding*) presenti in abbondanza nel *dataset*.
Inoltre, gli alberi decisionali sono in grado di individuare relazioni tra i vari attributi e il target. Per farlo creano una gerarchia di decisioni che facilita la previsione dell'appartenenza di ciascuna istanza a una classe del target, altrimenti difficilmente osservabile.
Inoltre, i Decision Tree aiutano ad identificare l'importanza di ciascun attributo ai fini della classificazione delle prenotazioni in "cancellate" o "non cancellate": tale verifica può essere utile per capire quali fattori influenzano maggiormente la decisione di cancellare una prenotazione.

Nella seguente analisi, il modello degli Alberi Decisionali è stato implementato utilizzando la classe `DecisionTreeClassifier` della libreria `scikit-learn`, molto diffusa ed ampiamente utilizzata per l'apprendimento automatico in Python.
Osservando il codice sorgente, è possibile notare alcuni parametri scelti durante la definizione del modello stesso:

- `max_depth` è settato a 4, limitando la profondità massima dell'albero per evitare l'eccessiva complessità e il rischio di ottenere il fenomeno di *overfitting*.
- `random_state`, assume lo stesso valore del parametro precedente, è utilizzato per generare numeri casuali in modo da garantire la riproducibilità nell'addestramento del modello ad ogni esecuzione del programma, consentendo di ottenere gli stessi risultati anche in seguito a diverse esecuzioni.

Di seguito é riportata l'implementazione completa del modello di Decision Tree, con il codice relativo all'addestramento e la previsione del target confrontato con il *testset* presente all'interno della funzione `modelling` nel file `utils.py` del progetto.

```
decision_tree_model = DecisionTreeClassifier(max_depth=4, random_state=0)
    decision_tree_model.fit(X_train, y_train)
    '''
    calcola e stampa un report di classificazione per valutare le
    prestazioni del modello Decision Tree addestrato utilizzando il set di
    addestramento (X_train, y_train). Tale report include diverse metriche
    di valutazione per ogni classe nel dataset. Stampando il report di
    classificazione per il set di addestramento, si puo ottenere una
    panoramica delle prestazioni del modello sui dati che sono stati
    utilizzati per addestrarlo. Questo puo dare un'idea di come il modello
    si comporta sulla stessa distribuzione di dati con cui e' stato
    addestrato.
    '''
    print("\n--- Prestazioni del modello Decision Tree applicato al
        set di Test: \n")
    print("\n--- Prestazioni del modello Decision Tree applicato al
        set di Test: \n", file=file)
    y_pred = decision_tree_model.predict(X_test)

    print_confusion_matrix(y_test, y_pred)
    print(classification_report(y_test, y_pred))
    print(classification_report(y_test, y_pred), file=file)
    decision_tree_accuracy = accuracy_score(y_test, y_pred)
    print("--- Accuratezza del modello Decision Tree:",
        round(decision_tree_accuracy, 5))
    print("--- Accuratezza del modello Decision Tree:",
        round(decision_tree_accuracy, 5), file=file)
```

Listing 4.1: Implementazione del modello Decision Tree

Per valutare le prestazioni del modello, sono state utilizzate diverse metriche di valutazione.
In particolare, durante l'esecuzione del codice, vengono collezionate le metriche di ***precision, recall, F1-score, accuracy, ROC*** e ***AUC***, offrendo una valutazione dettagliata delle capacità predittive del modello Decision Tree.
Di seguito è riportato il grafo rappresentante il modello di classificazione Decision Tree da noi implementato:

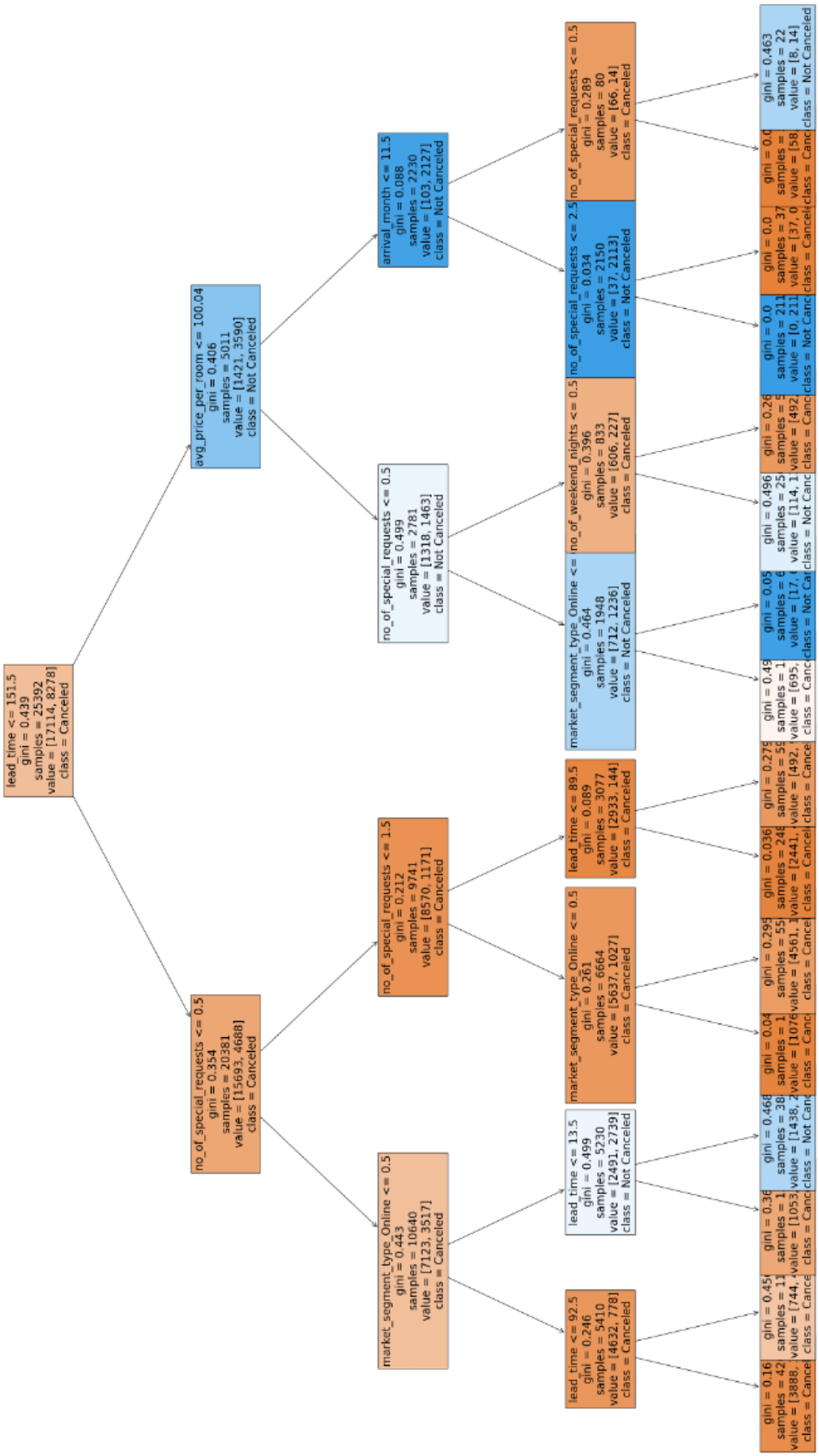


Figura 4.1: Grafo del modello Decision Tree.

Inoltre, inseriamo di seguito le metriche prestazionali e il grafico relativo alla curva ROC ottenuti dal modello da noi addestrato e testato sul set di istanze privo di errori:

```
---------- DECISION TREE Dataset Pulito ----------

--- Prestazioni del modello Decision Tree applicato al set di Test:

              precision    recall  f1-score   support

           0       0.83      0.91      0.87      7276
           1       0.78      0.63      0.70      3607

    accuracy                           0.82     10883
   macro avg       0.81      0.77      0.78     10883
weighted avg       0.82      0.82      0.81     10883

--- Accuratezza del modello Decision Tree: 0.81898
--- AUC del modello Decision Tree: 0.86528
```

Figura 4.2: Prestazioni del modello Decision Tree applicato al testset pulito.

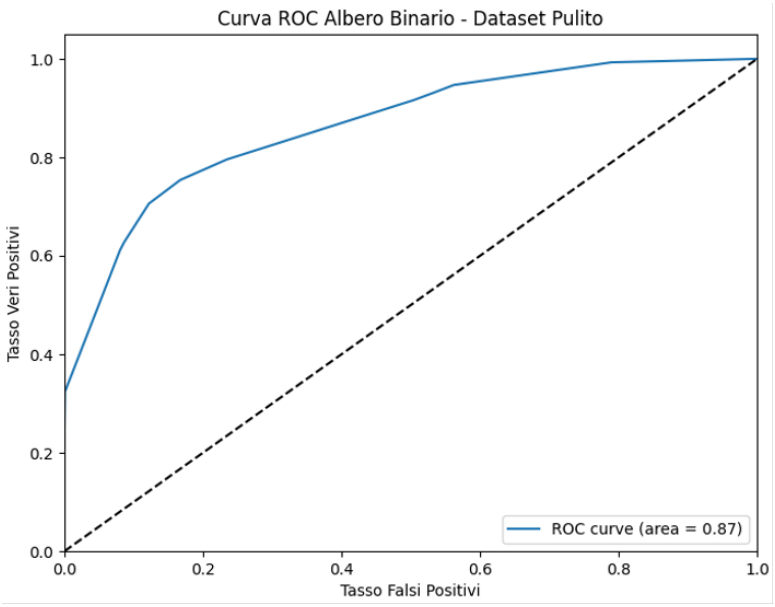


Figura 4.3: Curva ROC - Decision Tree

Una volta addestrato il modello tramite un *trainset* privo di errori e testato usando

un *testset* (anch'esso privo di errori) contenente 10883 istanze, le previsioni del modello sono state confrontate con le etichette corrette nel set di test per calcolare le metriche di valutazione ottenendo i seguenti risultati:

- La ***Precision*** per la **classe 0 è 0.83** e per la **classe 1 è 0.78**.
  Essa misura la capacità del modello di classificare correttamente le istanze positive. Si calcola come il rapporto tra il numero di istanze correttamente classificate come positive (veri positivi) e il numero totale di istanze classificate positive (veri positivi + falsi positivi).
  Un valore di *precision* elevato indica che il modello è in grado di identificare correttamente la classe positiva "canceled".

- Il ***Recall***, anche chiamato sensibilità o *true positive rate*, misura la capacità del modello di identificare correttamente le istanze positive.
  Si calcola come il rapporto tra il numero di istanze correttamente classificate positive (veri positivi) e il numero totale di istanze positive presenti nel set di dati.
  Nel nostro caso, il valore di *recall* per la **classe 0 è pari a 0.91**, mentre la **classe 1 è di 0.63**.

- L'***F1-score*** rappresenta una media ponderata delle metriche di *precision* e *recall*, ed è una misura complessiva delle prestazioni del modello calcolata come il rapporto tra il prodotto tra *precision* e *recall* e la loro somma. L'*F1-score* varia da 0 a 1, dove 1 rappresenta il valore ottimale.
  Nel nostro caso, per la **classe 0 è pari a 0.87** e per la **classe 1 è di 0.70**.
  Un *F1-score* elevato indica un buon bilanciamento tra *precision* e *recall*.

- L'***Accuracy*** indica la proporzione di istanze correttamente classificate rispetto al numero totale di istanze nel set di dati.
  Nel nostro caso è pari a **0.82**.
  Questa metrica fornisce una misura generale delle prestazioni del modello, ma può essere influenzata da una distribuzione non uniforme delle classi target nelle istanze del *dataset*.

- L'***AUC*** rappresenta l'area sottostante alla curva *ROC* (*Receiver Operating Characteristic*) e fornisce una misura della capacità del modello di classificare correttamente le istanze in base alla loro probabilità di appartenere alla classe positiva.
  Questa misura varia da 0 a 1, dove un valore più vicino a 1 indica una maggiore capacità predittiva del modello.
  Nel nostro caso, corrisponde a **0.87** suggerendo una buona capacità discriminante del modello.

Osservando le prestazioni ottenute dal modello Decision Tree è possibile notare risultati promettenti, con un'*Accuracy* del **82%** e un *AUC* del **86%** con una buona capacità di prevedere la cancellazione di una prenotazione.

4.1.2 Support Vector Machine - SVM

Il secondo modello di *Machine Learning* considerato sono le Support Vector Machine (SVM).
Questo algoritmo di apprendimento automatico si basa sulla teoria dei vettori di supporto, e consiste nella creazione di un iperpiano ottimale (una superficie decisionale) nello spazio delle caratteristiche per separare le diverse classi target, che nel *dataset* preso in esame corrispondono a "canceled" e "not_canceled".
La scelta di utilizzare le SVM è giustificata dalla loro capacità di elaborare *dataset* complessi con un elevato numero di attributi.

Nella nostra analisi, il modello delle Support Vector Machine (SVM) è stato implementato utilizzando la classe `SVC` della libreria `scikit-learn`.
Durante la loro implementazione sono stati definiti alcuni parametri:

- `kernel` è stato settato in modo da utilizzare un kernel lineare, che rappresenta una scelta comune per i modelli SVM, in quanto stabilisce una funzione di similarità lineare tra le istanze nel *dataset*.
  Di conseguenza, il modello SVM cerca di trovare un iperpiano lineare ottimale per separare le istanze nelle diverse classi target.

- `random_state` che, come illustrato per il modello precedente, assume lo stesso valore ed è utilizzato per generare numeri casuali in modo da garantire la riproducibilità nell'addestramento del modello ad ogni esecuzione del programma.

Di seguito riportiamo l'implementazione completa del modello SVM, con il codice relativo all'addestramento e la previsione del target confrontato con il *testset* presente all'interno della funzione `modelling` nel file `utils.py` del progetto.

```
'''Il kernel lineare e' una scelta comune per i modelli SVM e rappresenta
una funzione di similarita' lineare tra le istanze nel dataset. Questo
significa che il modello SVM cerchera' di trovare un iperpiano lineare
ottimale per separare le diverse classi nel problema di classificazione.'''

svm_model = SVC(kernel='linear', random_state=0)
    svm_model.fit(X_train, y_train)

    y_pred = svm_model.predict(X_test)
    print_confusion_matrix(y_test, y_pred)

    print(classification_report(y_test, y_pred))
```

```
    print(classification_report(y_test, y_pred), file=file)

    # Accuratezza modello
    svm_accuracy = accuracy_score(y_test, y_pred)
    print("--- Accuratezza del modello SVM:", round(svm_accuracy, 5))
    print("--- Accuratezza del modello SVM:", round(svm_accuracy, 5),
        file=file)

    # #ROC Curve
    y_prob = svm_model.decision_function(X_test)
    fpr, tpr, thresholds = roc_curve(y_test, y_prob)

    # Calcolo dell'AUC
    roc_auc = auc(fpr, tpr)
```

Listing 4.2: Implementazione del modello SVM

Anche per il modello SVM, al fine di valutarne le prestazioni, sono state utilizzate diverse metriche di valutazione.
In particolare, durante l'esecuzione del codice, vengono collezionate le metriche di ***precision, recall, F1-score, accuracy, ROC*** e ***AUC*** offrendo una valutazione dettagliata delle capacità predittive del modello Decision Tree.

```
---------- SUPPORT VECTOR MACHINE Dataset Pulito ----------

              precision    recall  f1-score   support

           0       0.84      0.88      0.86      7276
           1       0.74      0.65      0.69      3607

    accuracy                           0.81     10883
   macro avg       0.79      0.77      0.77     10883
weighted avg       0.80      0.81      0.80     10883

--- Accuratezza del modello SVM: 0.8063
--- AUC del modello SVM: 0.85572
```

Figura 4.4: Prestazioni del modello SVM applicato al testset pulito.

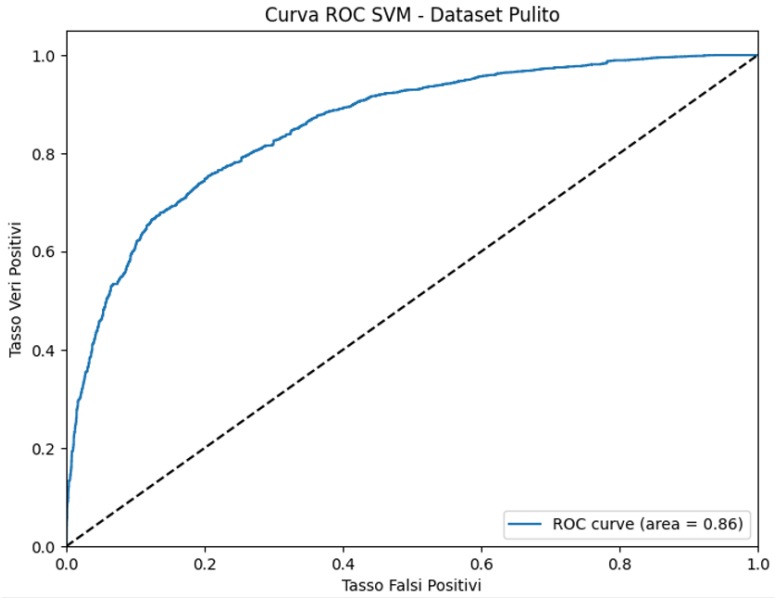


Figura 4.5: Curva ROC - SVM

Anche in questo caso, per valutare le prestazioni del modello SVM, é stato utilizzato un *trainset* (pulito) e un *testset* (pulito) che conteniene 10.883 istanze, suddivise in due classi: 0 e 1.
Come per l'Albero Decisionale, le previsioni del modello sono state confrontate con le etichette corrette del *testset* per calcolare le metriche prestazionali:

- La ***Precision*** per la **classe 0 è 0.84** e per la **classe 1 è 0.74**.
- Il ***Recall***, per la **classe 0 è pari a 0.88** e per la **classe 1 è di 0.65**.
- L' ***F1-score*** varia da 0 a 1, dove 1 rappresenta il valore ottimale. Nel nostro caso, l'F1-score per la **classe 0 è 0.86** e per la **classe 1 è 0.69**.
- L'***Accuracy*** complessiva del modello SVM è **0.81**.
- L'***AUC*** del modello SVM è **0.86**, il che suggerisce una buona capacità discriminante del modello.

Osservando le metriche prestazionali ottenute, possiamo affermare che: il modello SVM presenta risultati promettenti applicato al *dataset* pulito, con un valore di *Accuracy* pari all'**81%** e un *AUC* del **86%**.

4.2 Funzioni per il deterioramento delle istanze del dataset

Una volta ottenute le metriche prestazionali dei modelli e i risultati delle misure di *data quality* è stato deciso di sporcare il *dataset*, andando ad implementare alcuni metodi per introdurre valori nulli, *outliers*, inconsistenza, duplicati e inaccuratezza con lo scopo successivo di monitorare le capacità predittive dei due modelli di classificazione nel classificare le istanze con target pari a 1 ('canceled') all'aumentare del degradamento dei dati del *dataset*.

Di seguito sono presenti i metodi implementati per andare a sporcare le istanze del *dataset* e il codice completo è disponibile all'interno del file `utils.py` reperibile al seguente link GitHub:
`https://github.com/dbancora/ProgettoArchitetturaDati_DataQuality/blob/main/utils.py`

La prima funzione realizzata è `introduce_null_values`, progettata per aggiungere valori nulli (o valori mancanti) ad una determinata colonna del *dataset*; essa prende in input il nome della colonna, la percentuale di valori nulli da inserire e il set di dati su cui operare.
Inizialmente, la funzione verifica il tipo di dati della colonna passata come argomento.
Nel caso in cui sia di tipo `int64`, crea un numero di valori nulli, equivalente alla percentuale fornita, sostituendo la rispettiva quantità di valori specificata con il numero 999 (Consideriamo il valore 999 come un valore nullo per gli interi).
Se la colonna è di tipo `object`, la funzione genera valori nulli sostituendo i valori da modificare con stringhe vuote ("").
In presenza di una colonna di tipo `float`, genera valori nulli rimpiazzando i valori corrispondenti a 999.99.
Successivamente, gli indici dei valori nulli vengono generati casualmente, senza sostituzione, utilizzando la funzione `np.random.choice`.
Questi indici vengono utilizzati per assegnare i valori nulli alla colonna corrispondente nel *dataset*.

```
def introduce_null_values(column, percentage, df):
    if column.dtype == 'int64':
        num_null_values = int(len(column) * (percentage / 100))
        null_indices = np.random.choice(len(column), num_null_values,
            replace=False)
        column.loc[null_indices] = 999
    elif column.dtype == 'object':
        num_null_values = int(len(column) * (percentage / 100))
        null_indices = np.random.choice(len(column), num_null_values,
            replace=False)
        column.loc[null_indices] = ""
```

```
    elif column.dtype == 'float':
        num_null_values = int(len(column) * (percentage / 100))
        null_indices = np.random.choice(len(column), num_null_values,
            replace=False)
        column.loc[null_indices] = 999.99
    df[column.name] = column
    return df
```

Listing 4.3: Funzione introduce_null_values

Infine, il set di dati viene aggiornato con la colonna contenente i valori nulli e viene restituito come risultato dell'esecuzione della funzione.

Le funzioni implementate successivamente sono `add_outliers_int` e `add_categorical_outliers` progettate per aggiungere *outliers* a un *dataset* in base al tipo di dati della colonna specificata come argomento:

- `add_outliers_int` aggiunge *outlier* a una colonna di tipo intero, calcolando i limiti degli *outlier* basati sulla media e la deviazione standard dei dati presenti nella colonna stessa.
  In seguito, genera casualmente un insieme di indici delle righe da modificare e crea valori *outlier* casuali all'interno dei limiti calcolati.
  Infine, assegna gli *outlier* alle righe selezionate nella colonna specificata nel set di dati.

- `add_categorical_outliers` aggiunge *outlier* a una colonna di tipo categorico calcolando il valore *outlier* come il valore meno frequente nella colonna specificata.
  Successivamente, seleziona casualmente un insieme di righe da modificare e assegna il valore *outlier* a tali righe nella colonna specificata nel *dataset*.

Entrambe le funzioni restituiscono il *dataset* modificato come risultato dell'operazione.

```
def add_outliers_int(column, percentage, df):
    mean = df[column].mean()
    std = df[column].std()

    lower_limit = mean - (z_score * std)
    upper_limit = mean + (z_score * std)

    percentage = percentage / 100

    n_rows = int(len(df) * percentage)

    outlier_indices = np.random.choice(df.index, size=n_rows, replace=False)
    outliers = np.random.uniform(lower_limit, upper_limit, size=n_rows)
    outliers = np.abs(outliers)  # Modulo per evitare valori nulli
```

```
    outliers = np.round(outliers).astype(int)
    df.loc[outlier_indices, column] = outliers
    return df
```

Listing 4.4: Funzione add_outliers_int

```
def add_categorical_outliers(column, percentage, dataset):
    # Calcola il valore outlier per la colonna specificata
    outlier_value = dataset[column].value_counts().idxmin()
    print(outlier_value)

    # Determina il numero di righe da modificare
    n_rows = int(len(dataset) * (percentage / 100))

    # Inserisci il valore outlier nella percentuale di righe specificata
    outlier_indices = dataset.sample(n_rows).index
    dataset.loc[outlier_indices, column] = outlier_value

    return dataset
```

Listing 4.5: Funzione add_categorical_outliers

Per aggiungere inconsistenza è stata sviluppata la funzione `introduce_inconsistencies`. Essa prende in input: un array contenente i nomi degli attributi da modificare, un array di valori da assegnare a tali colonne, il *dataset* su cui operare e la percentuale di righe da modificare.
Inizialmente, viene creata una copia del *dataset* originale chiamata `modified_dataset` e successivamente viene calcolato il numero totale di righe presenti nel *dataset* e il numero di righe da modificare, basato sulla percentuale fornita come parametro.
Utilizzando la funzione `random.sample` vengono selezionati casualmente gli indici di riga corrispondenti alle istanze da modificare.
In seguito, per ciascuna riga selezionata, vengono assegnati specifici valori alle colonne indicate: il primo valore viene assegnato alla prima colonna specificata nell'array `column_array` e il secondo valore viene assegnato alla seconda colonna specificata nello stesso array.
In questo modo, si introduce un'inconsistenza tra i valori delle due colonne selezionate.
Infine, viene stampato il *dataset* modificato e viene restituito come risultato dell'esecuzione della funzione.

```
def introduce_inconsistencies(column_array, values_array, dataset,
        percentage):
    modified_dataset = dataset.copy()

    num_rows = len(modified_dataset)
    num_rows_to_modify = int(num_rows * (percentage / 100))
    rows_to_modify = random.sample(range(num_rows), num_rows_to_modify)
```

```
8
9       for row_index in rows_to_modify:
10          row = modified_dataset.loc[row_index, column_array]
11          modified_dataset.loc[row_index, column_array[0]] = values_array[0]
12          modified_dataset.loc[row_index, column_array[1]] = values_array[1]
13
14      print(modified_dataset)
15      return modified_dataset
```

Listing 4.6: Funzione introduce_inconsistencies

Con lo scopo di duplicare le righe, è stato deciso di creare la funzione `duplicate_rows` che è progettata per duplicare una determinata percentuale di righe all'interno di un *dataset* dato in input.
Viene calcolato il numero di righe da duplicare, basato sulla percentuale fornita rispetto alla lunghezza totale del *dataset* selezionando successivamente in modo casuale gli indici delle righe da duplicare tramite la funzione `np.random.choice`.
La selezione delle righe avviene con possibilità di sostituzione, consentendo di duplicare la stessa riga più volte.
L'estrazione dal *dataset* delle righe selezionate avviene utilizzando l'operatore di indicizzazione `loc` per poi essere memorizzate in un nuovo set di dati chiamato `duplicated_data`.
Per concludere, il DataFrame `duplicated_data` viene concatenato al *dataset* originale utilizzando la funzione `pd.concat` con il parametro `ignore_index` impostato su True in modo da generare nuovi indici per le righe duplicate.
Il *dataset* modificato sarà poi restituito come risultato dell'esecuzione della funzione.

```
1  def duplicate_rows(dataset, percent):
2      num_duplicates = int(len(dataset) * percent / 100)
3      duplicated_rows = np.random.choice(dataset.index, size=num_duplicates,
4          replace=True)
5      duplicated_data = dataset.loc[duplicated_rows]
6      dataset = pd.concat([dataset, duplicated_data], ignore_index=True)
7      return dataset
```

Listing 4.7: Funzione duplicate_rows

Per quanto rigurda l'aggiunta di inaccuratezza al *dataset*, viene utilizzata la funzione `remove_first_letter` progettata per rimuovere la prima lettera di un certo numero di righe all'interno di una determinata colonna passata come parametro insieme al *dataset* su cui operare e alla percentuale di righe da modificare.
Viene calcolato il numero di righe da modificare, basato sulla percentuale fornita rispetto alla lunghezza totale del *dataset*. Inoltre, vengono selezionati casualmente gli indici delle righe da modificare utilizzando la funzione `random.sample`.
Successivamente, la funzione verifica il tipo di dati della colonna:

- Se la colonna è di tipo `object` (stringa), la funzione itera su ogni valore nella colonna.
  Se l'indice della riga corrente non è presente negli indici delle righe da modificare, il ciclo continua con la successiva iterazione.
  Nel caso in cui il valore non è nullo e ha una lunghezza maggiore di 0, viene selezionato casualmente un indice compreso tra 0 e la lunghezza del valore sottratto di un'unità (-1).
  Viene quindi creato un nuovo valore modificato rimuovendo la lettera corrispondente all'indice calcolato e il *dataset* viene aggiornato assegnando il nuovo valore alla riga corrispondente.

- Se la colonna è di tipo intero (`int`) o `float` (virgola mobile), la funzione itera su ogni valore nella colonna.
  Se l'indice della riga corrente non è presente negli indici delle righe da modificare, il ciclo continua con la successiva iterazione.
  Nel caso in cui il valore non è nullo, viene convertito in una stringa:

  – Per la colonna `int`, viene rimossa l'ultima cifra (carattere) della stringa e il valore viene convertito nuovamente in un intero;

  – Per la colonna `float`, viene individuata la posizione del punto decimale nella stringa, e poi viene rimossa la cifra corrispondente a quella posizione. Il valore viene quindi convertito nuovamente in un `float`.

  Il *dataset* viene aggiornato assegnando il nuovo valore alla riga corrispondente.

Infine, il *dataset* modificato viene restituito come risultato dell'esecuzione della funzione

```
def remove_first_letter(column, dataset, percentage):
    num_rows = int(len(dataset) * (percentage / 100))
    rows_to_modify = random.sample(range(len(dataset)), num_rows)
    rows_modified = 0

    if column.dtype == 'object':
        for i, value in enumerate(column):
            if i not in rows_to_modify:
                continue
            if value is not None and len(value) > 0:
                index = random.randint(0, len(value) - 1)
                modified_value = value[:index] + value[index + 1:]
                dataset.loc[i, column.name] = modified_value
                rows_modified += 1

    elif pd.api.types.is_integer_dtype(column):
        for i, value in enumerate(column):
            if i not in rows_to_modify:
```

```
                continue
            if pd.notnull(value):
                value_str = str(value)
                modified_value = value_str[:-1]
                if modified_value:
                    dataset.loc[i, column.name] = int(modified_value)
                    rows_modified += 1

    elif pd.api.types.is_float_dtype(column):
        for i, value in enumerate(column):
            if i not in rows_to_modify:
                continue
            if pd.notnull(value):
                value_str = str(value)
                decimal_index = value_str.index('.')
                modified_value = value_str[:decimal_index]
                    + value_str[decimal_index + 1:]
                if modified_value:
                    dataset.loc[i, column.name] = float(modified_value)
                    rows_modified += 1

    return dataset
```

Listing 4.8: Funzione remove_first_letter

4.3 Addestramento dei modelli su trainset "sporco"

Per lo svolgimento del progetto sono stati implementati due differenti modelli di *Machine Learning*, ossia Alberi Decisionali e Support Vector Machine (SVM), permettendo la classificazione delle istanze del *dataset*.
All'interno di questa sezione della relazione si farà riferimento alle metriche di prestazione ottenute dai vari modelli che, ai fini di una miglior impaginazione e leggibilità del documento, sono stati salvati all'interno di specifici *file* disponibili sul link GitHub del progetto: `https://github.com/dbancora/ProgettoArchitetturaDati_DataQuality/tree/main/Performance%20Modelli%20Decisonali`
Come precedentemente descritto, inizialmente i modelli sono stati addestrati e testati utilizzando un *train-set* e *test-set* generati a partire dal *dataset* "pulito".
Di conseguenza, si farà riferimento alle metriche prestazionali presentate nelle sezioni precedenti.
Per raggiungere l'obbiettivo del progetto, saranno utilizzate le funzioni di deterioramento delle istanze presentate nella sezione precedente della relazione.
Inizialmente verranno monitorate le metriche prestazionali dei due modelli presi in considerazione applicando, in quantità crescente, ciascuna funzione su un unico attributo del

trainset.
L'attributo sporcato sarà `lead_time`, di tipo `float` e rappresenta il numero di giorni che intercorrono dal momento in cui viene effettuata una prenotazione, all'effettivo soggiorno presso una struttura alberghiera.
In base al grafico generato dalla *feature importance*, è possibile notare come questo sia il più importante per gli algoritmi di decisione.
Di fatto la funzione `feature_importances`, definita all'interno della libreria `scikit-learn`, è in grado di associare, a ciascun attributo del *dataset*, un punteggio che descrive la quantità d'informazione fornita dall'attributo stesso tramite un modello di classificazione Decision Tree.
Di seguito riportiamo il codice da noi implementato per generare il grafico di *feature importance* e il risultato ottenuto:

```
def feature_importance(x_train, y_train, title):
    dt = DecisionTreeClassifier()
    dt.fit(x_train, y_train)

    sort = dt.feature_importances_.argsort()
    features_sorted = np.array(x_train.columns.to_list())
    features_sorted = features_sorted[sort]
    feature_importances = pd.DataFrame({'features': features_sorted,
        'importance': dt.feature_importances_[sort]})

    plt.figure(figsize=(26, 20))
    sns.barplot(data=feature_importances, y="features", x="importance",
            order=feature_importances.sort_values('importance').features)

    plt.xlabel("Importanza delle feature " + title)
    plt.title("Importanza di ciascuna feature " + title)

    plt.show()
```

Listing 4.9: Funzione feature_importance

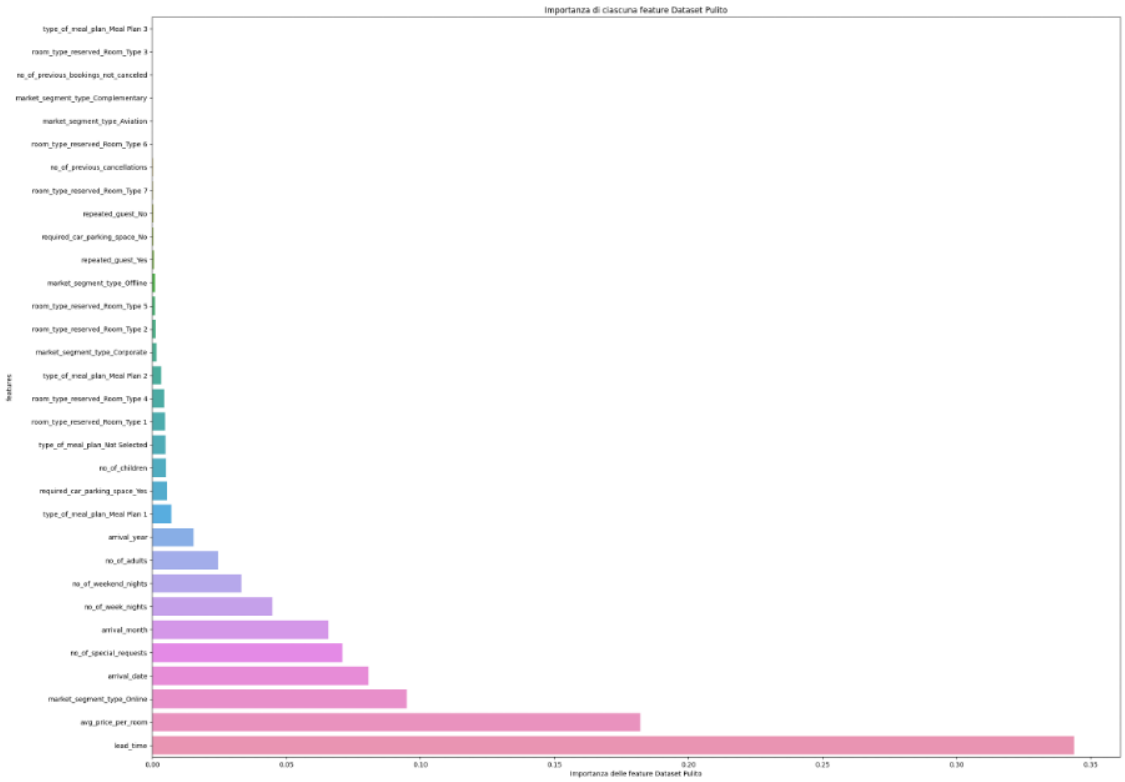
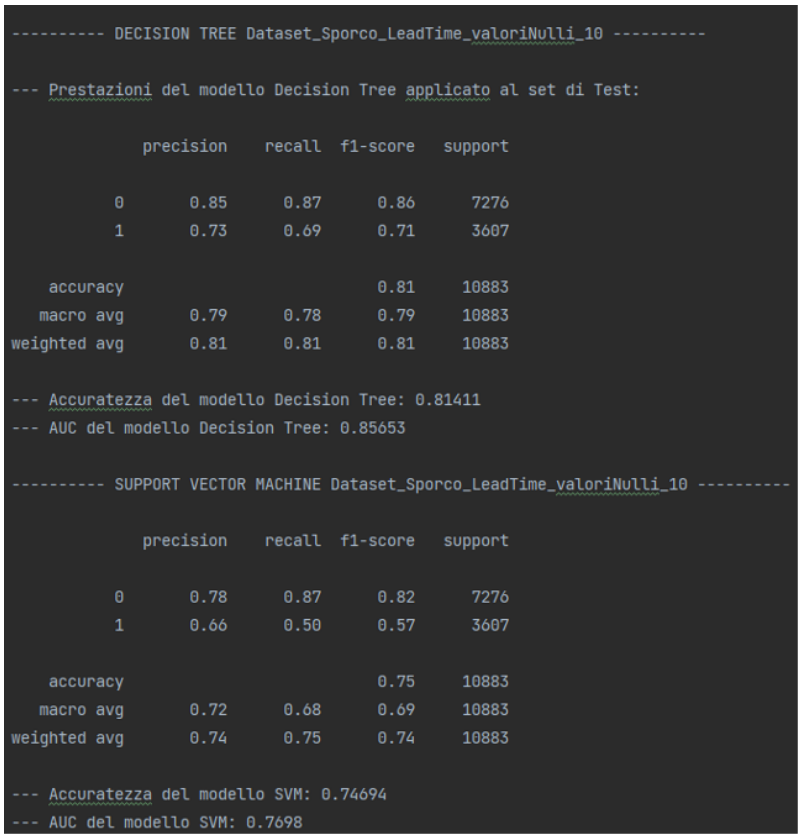


Figura 4.6: Grafico feature importance - Dataset pulito

Scegliere di sporcare l’attributo che restituisce la maggior informazione di classificazione delle istanze, consente di verificare quanto il modello di classificazione sia robusto. Come ultimo punto dell’analisi verranno mostrati i risultati ottenuti applicando le varie funzioni di deterioramento del *train-set* combinate tra loro e applicate su attributi differenti.

4.3.1 Aggiunta dei valori nulli

La prima funzione utilizzata per sporcare il *dataset* `introduce_null_values` ha il compito di sostituire quantità crescenti di valori contenuti nella colonna `lead_time` con valori nulli. Con una percentuale di valori nulli inseriti pari al 10% si ottengono le seguenti metriche prestazionali:



```
---------- DECISION TREE Dataset_Sporco_LeadTime_valoriNulli_10 ----------

--- Prestazioni del modello Decision Tree applicato al set di Test:

              precision    recall  f1-score   support

           0       0.85      0.87      0.86      7276
           1       0.73      0.69      0.71      3607

    accuracy                           0.81     10883
   macro avg       0.79      0.78      0.79     10883
weighted avg       0.81      0.81      0.81     10883

--- Accuratezza del modello Decision Tree: 0.81411
--- AUC del modello Decision Tree: 0.85653

---------- SUPPORT VECTOR MACHINE Dataset_Sporco_LeadTime_valoriNulli_10 ----------

              precision    recall  f1-score   support

           0       0.78      0.87      0.82      7276
           1       0.66      0.50      0.57      3607

    accuracy                           0.75     10883
   macro avg       0.72      0.68      0.69     10883
weighted avg       0.74      0.75      0.74     10883

--- Accuratezza del modello SVM: 0.74694
--- AUC del modello SVM: 0.7698
```

Figura 4.7: Metriche prestazionali attributo lead_time 10% di valori nulli.

Rieseguendo il programma ed andando a sporcare ulteriormente l'attributo `lead_time`, aggiungendo prima il 20% per poi raggiungere il 40% dei valori nulli, nel caso degli Alberi Decisionali otteniamo delle *performance* lievemente inferiori, percepibili nelle metriche di *precision* e *accuracy* del modello, con relativo cambiamento delle soglie di classificazione all'interno dei nodi dell'Albero Decisionale.
Ciò suggerisce che la *feature* `lead_time` è influenzata dai dati sporchi nel *dataset* e potrebbe essere un fattore critico per identificare le prenotazioni con alta probabilità di cancellazione in tali circostanze.
Nel momento in cui si raggiunge il 40% di valori nulli, osservando la struttura dell'Albero Decisionale si nota che viene sostituito l'attributo `lead_time`, precedentemente presente nella condizione del nodo radice per prendere la prima decisione di classificazione delle istanze, inserendo come soglia per la classificazione l'attributo `no_of_special_requests` ≤ 0.5.
Di seguito è riportato il grafo del modello di classificazione Decision Tree con il 40% di valori nulli nell'attributo `lead_time`:

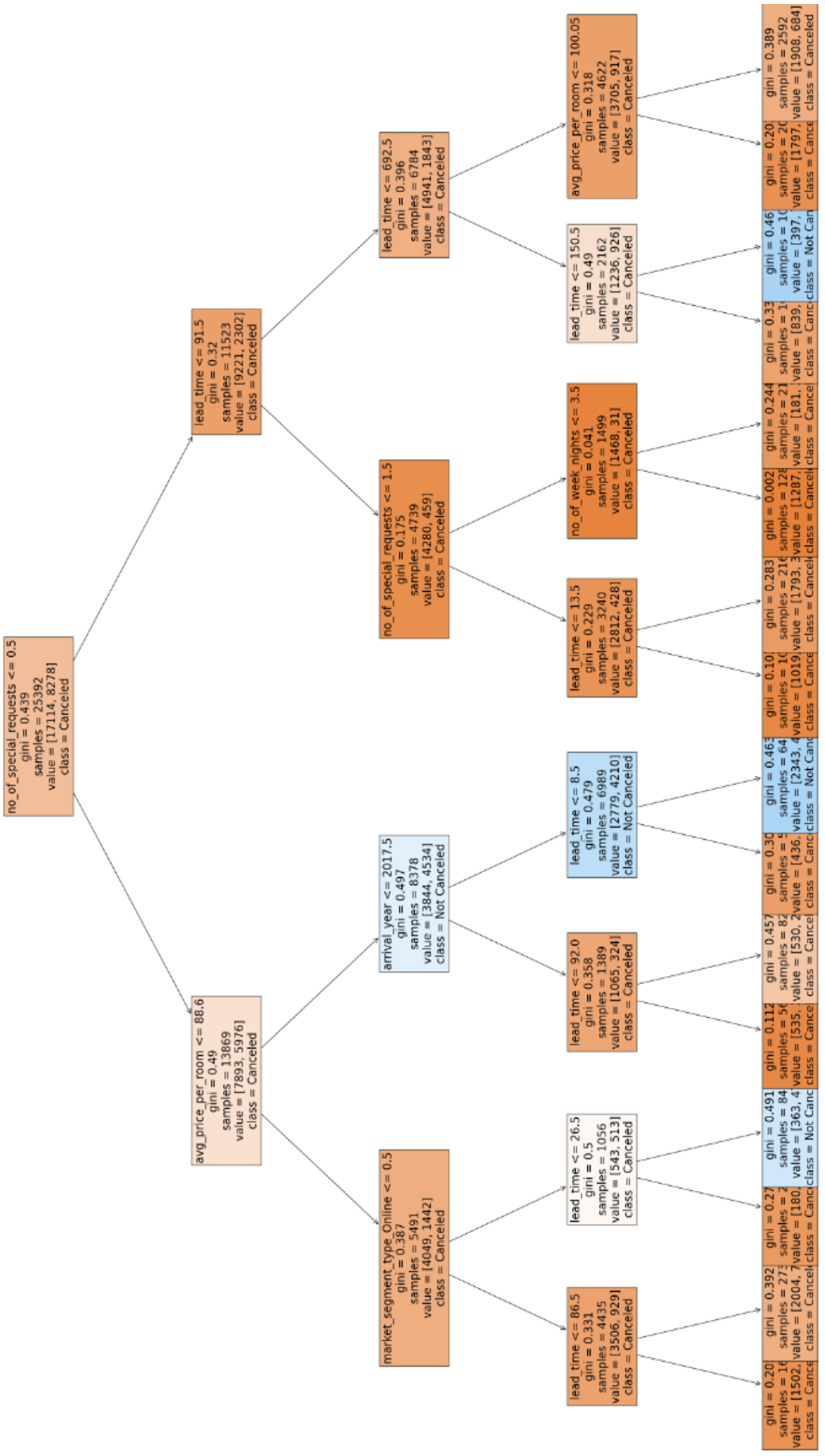


Figura 4.8: Albero Decisionale con 40% di valori nulli nell'attributo lead_time.

Per quanto riguarda il modello delle SVM, si percepisce un decremento più significativo di tutte le metriche rispetto al modello addestrato utilizzando il *dataset* "pulito" di partenza.
Con il raggiungimento del 50% di valori nulli nell'attributo `lead_time` si nota che: all'aumentare dei valori nulli l'attributo presenta una minor *feature importance* e, di conseguenza, il modello Decision Tree sfrutta altri attributi per effettuare la classificazione delle istanze.
Infatti, osservando il grafico della *feature importance* ora l'attributo con maggior importanza è `avg_price_per_room`.

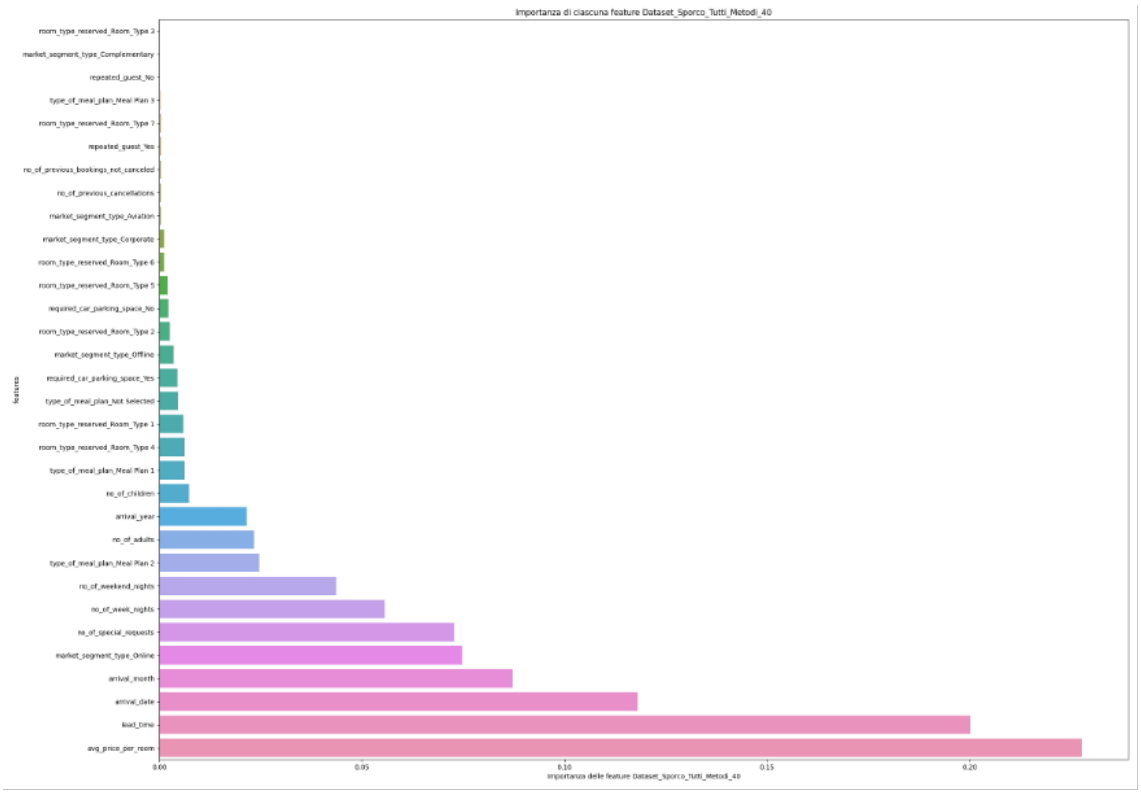


Figura 4.9: Grafico feature importance con 50% di valori nulli nell'attributo lead_time.

Di seguito sono presenti le *performance* dei due modelli con il 50% di valori nulli:

```
---------- DECISION TREE Dataset_Sporco_LeadTime_valoriNulli_50 ----------

--- Prestazioni del modello Decision Tree applicato al set di Test:

              precision    recall  f1-score   support

           0       0.77      0.93      0.84      7276
           1       0.74      0.44      0.55      3607

    accuracy                           0.76     10883
   macro avg       0.76      0.68      0.69     10883
weighted avg       0.76      0.76      0.74     10883

--- Accuratezza del modello Decision Tree: 0.76284
--- AUC del modello Decision Tree: 0.83669

---------- SUPPORT VECTOR MACHINE Dataset_Sporco_LeadTime_valoriNulli_50 ----------

              precision    recall  f1-score   support

           0       0.75      0.89      0.82      7276
           1       0.65      0.41      0.51      3607

    accuracy                           0.73     10883
   macro avg       0.70      0.65      0.66     10883
weighted avg       0.72      0.73      0.71     10883

--- Accuratezza del modello SVM: 0.73206
--- AUC del modello SVM: 0.74323
```

Figura 4.10: Performance dei due modelli di classificazione con il 50% di valori nulli nell'attributo lead_time.

In conclusione, possiamo notare che all'aumentare dei valori nulli presenti nella colonna `lead_time` questa perde d'importanza ai fini della classificazione.
Il modello dell'Albero Decisionale risulta quindi abbastanza resistente fino al raggiungimento del 40% di valori nulli, senza notare un peggioramento marcato ed improvviso delle prestazioni mostrando però un valore di *precision* sempre minore (di poco) rispetto alle prestazioni del modello ottenuto con il *dataset* originale.
Questo può essere dovuto al fatto che: l'aggiunta di valori nulli all'interno dei valori di uno specifico attributo, potrebbe influenzare la distribuzione dei dati, rendendo più facile al modello distinguere correttamente gli esempi negativi (“not_canceled”) da quelli positivi (“canceled”).

Superato il 50% di valori nulli dell'attributo `lead_time` si ottengono delle prestazioni del modello non soddisfacenti per l'obbiettivo iniziale di classificazione con relativo degrado delle *performance*.
Anche se le varie prestazioni rimangono costanti all'aumentare dei valori nulli (oltre il 50%) il modello fatica notevolmente ad eseguire la discriminazione delle varie istanze tra le due classi target non riuscendo a prevedere in modo affidabile la cancellazione di una prenotazione sfruttando l'attributo `lead_time`.
Da notare come l'attributo stesso smette di essere più utilizzato all'interno delle varie condizioni di classificazione nei nodi dell'albero all'aumentare del numero di valori nulli dell'attributo.
Per quanto riguarda il modello Support Vector Machine (SVM), rispetto al modello degli Alberi Decisionali, può essere più sensibile alla presenza di valori nulli a causa della sua dipendenza dalla separabilità lineare dei dati risultando meno robusto.
Ciò è dovuto al fatto che la SVM si basa sulla costruzione di un iperpiano per separare le classi target, e i valori nulli possono influenzare negativamente la determinazione del piano di separazione ottimale.
Si nota che, all'aumentare dei valori nulli nell'attributo `lead_time`, si percepisce un continuo peggioramento delle prestazioni del modello.
Anche per quanto riguarda il modello delle SVM, come nei Decision Tree, superato il 50% di valori nulli presenti nell'attributo `lead_time` le prestazioni non peggiorano ulteriormente pur mantenendo prestazioni del modello non soddisfacenti per l'obbiettivo di classificazione iniziale.

4.3.2 Aggiunta di Outlier

Successivamente abbiamo analizzato l'effetto dell'aumento dei valori *outlier* nell'attributo `lead_time` sulle prestazioni del modello degli Alberi Decisionali e Support Vector Machine.
La presenza di *outlier* nell'attributo `lead_time` può derivare da errori di registrazione delle prenotazioni, prenotazioni anticipate o ritardi imprevisti nel soggiorno.
Un *outlier* è un valore che si discosta significativamente dalla media dei valori assunti da un attributo nel *dataset*.
In questa sezione sono state condotte diverse sperimentazioni utilizzando un *dataset* pulito come punto di riferimento per poi sostituire progressivamente una percentuale crescente di valori dell'attributo `lead_time` con valori *outlier*.
Le prestazioni di entrambi i modelli sono state valutate utilizzando metriche di valutazione come *precision, recall, f1-score, accuracy* e *AUC*.
Per quanto riguarda il modello **Decision Tree** i risultati hanno mostrato che inizialmente, con il 10% di valori *outlier*, si ha un lieve incremento delle prestazioni e il modello riesce a classificare leggermente meglio le varie istanze nelle classi target "canceled" o "not canceled".

Questo aumento, ottenuto con il 10% di *outlier*, potrebbe essere attribuito a diversi fattori: gli alberi decisionali sono generalmente considerati modelli robusti in presenza di valori anomali.
Di conseguenza, in presenza di una bassa percentuale di *outlier*, il modello potrebbe ancora essere in grado di apprendere determinati pattern presenti nei dati non influenzati dagli *outlier*, migliorando così le prestazioni.
Tuttavia, gli *outlier* eccessivi o mal posizionati possono avere un impatto negativo sulle prestazioni del modello.
Infatti si osserva che, superato il 20% di valori *outlier*, la capacità di discriminare correttamente le istanze positive del modello diminuisce, ottenendo metriche prestazionali simili a quelle riscontrate dal modello con il 30%, o più, di valori nulli all'interno dell'attributo `lead_time`.
Questo indica che l'aumento degli *outlier* può influenzare negativamente la capacità del modello di identificare correttamente gli esempi positivi.

Riguardo al modello della **Support Vector Machine**, i risultati ottenuti indicano un deterioramento delle prestazioni all'aumentare della percentuale di *outlier* nell'attributo `lead_time`: L'accuratezza diminuisce gradualmente, mentre le metriche come *precision, recall* e *F1-score* mostrano un calo significativo per la classe "canceled".
Ciò suggerisce che il modello SVM fatica a prevedere eventuali prenotazioni cancellate quando la presenza di *outlier* nell'attributo `lead_time` è elevata.
L'impatto negativo sulle prestazioni del modello, dovuto alla presenza crescente di *outlier* nell'attributo, può essere spiegato da vari fattori.
Innanzitutto, gli *outlier* rappresentano valori estremi che possono distorcere la relazione tra l'attributo `lead_time` e le variabili predittive. Ciò può compromettere la capacità del modello SVM di individuare i pattern e le relazioni significative nei dati di addestramento, riducendo la sua precisione e la capacità di generalizzazione.
Inoltre, gli *outlier* possono influenzare la fase di apprendimento del modello SVM, in quanto durante questa fase il modello cerca di identificare un iperpiano ottimale che sia in grado di separare in modo ottimale le istanze nelle due classi target, complicandone la ricerca.

4.3.3 Aggiunta di inaccuratezza

Un ulteriore funzione utilizzata per studiare e testare le prestazioni dei modelli di *Machine Learning* addestrati su un *dataset* sporco, è `remove_first_letter` che permette l'aggiunta di inaccuratezze all'interno dei valori dell'attributo `lead_time`.
Un'inaccuratezza è data da errori di registrazione o battitura dei dati e si verifica quando mancano uno o più caratteri (oppure numeri) nel valore di un attributo.
Come nei casi precedenti, anche in questa sezione, l'attributo `lead_time` è stato manipolato con l'obbiettivo di introdurre progressivamente un numero di inconsistenze crescente

(partendo da un 10% con il raggiungimento dell'80% dei valori dell'attributo) e valutando, ad ogni passaggio, le relative metriche prestazionali di entrambi i modelli.
Considerando gli **Alberi Decisionali**, in seguito all'aggiunta di inaccuratezze nei valori dell'attributo `lead_time`, le prestazioni rimangono generalmente stabili fino al 30% di inconsistenze con un degradamento lieve delle metriche visibili soprattutto nell'accuratezza e *AUC* del modello.
Al raggiungimento del 50% di inaccuratezze, iniziano a verificarsi leggeri cali nelle metriche prestazionali come *precision, accuracy* e *AUC*.
Con il 60% di inconsistenze si nota un peggioramento marcato di *precision* e *f1-score*, con un valore di *recall* minore per la classe 0 e maggiore per la classe 1, nonché un peggioramento lieve di *accuracy* e *AUC*.
Inoltre, si notano modifiche nella struttura dell'Albero Decisionale, sostituendo la condizione soglia del nodo radice e anche quelle di altri nodi a livelli inferiori.
Risulta anche visibile la diminuzione dell'importanza dell'attributo `lead_time` nel grafico *feature importance* rimanendo comunque la feature con una maggiore importanza per la classificazione.
Un notevole peggioramento nelle prestazioni si verifica all'aggiunta dell'80% di inaccuratezza, dove l'Albero Decisionale ha un valore di *precision* inferiore al 50%, il che equivale a scegliere in modo randomico la classe target al quale appartiene l'istanza considerata.
Anche in questo caso si ha un peggioramento di *f1-score* e del valore di *recall* che, come con il 60% di inconsistenze, diminuisce per la classe 0 e aumenta per la classe 1.
Inoltre, diminuiscono in modo significativo le metriche di *accuracy* e *AUC*.

Per quanto riguarda il modello **Support Vector Machine**, si osserva una tendenza simile.
Fino al 30% di inconsistenza, le prestazioni rimangono relativamente stabili, con lievi variazioni nelle metriche prestazionali.
Con il raggiungimento del 50% di inconsistenze nei valori dell'attributo, si riscontra un peggioramento delle metriche prestazionali simile a quanto riscontrato negli Alberi Decisionali.
Al raggiungimento del 60% di inconsistenze invece, la metrica di *precision* diminuisce, anche se i valori di *AUC* e *accuracy* decrescono meno significativamente rispetto al modello degli Alberi Decisionali.
Anche in questo caso, il valore di *recall* aumenta per la classe target 1 e diminuisce per la classe 0.
Con l'aggiunta dell'80% di inaccuratezze, anche il modello SVM non è esente da peggioramenti, anche se in questo caso sembrerebbe reagire meglio rispetto al modello degli Alberi Decisionali.
Nelle metriche si nota un peggioramento marcato delle prestazioni per quanto riguarda i valori di *precision, accuracy* e *AUC*, ma meno drastico rispetto a quanto osservato con i

Decision Tree.
Infatti, in queste circostanze il valore di *precision* si attesta attorno al 60% a differenza del 50% associato al modello degli Alberi Decisionali.
In generale, possiamo concludere che l'inserimento di inconsistenze nell'attributo `lead_time` ha un impatto progressivamente negativo sulle prestazioni dei modelli di Decision Tree e SVM.
Fino al 30%, l'impatto è relativamente limitato, ma oltre questa soglia si verifica un deterioramento più significativo delle prestazioni.
È importante notare che l'entità dell'inconsistenza e la sua distribuzione nel *dataset* possono influenzare il grado di deterioramento.
Con bassi livelli di inconsistenze, i modelli riescono ancora a distinguere le istanze corrette da quelle errate, mantenendo buone prestazioni.
Tuttavia, con un aumento delle inconsistenze, diventano meno affidabili, tendendo a fornire previsioni errate.
Confrontando i risultati appena ottenuti, si nota che l'introduzione di inconsistenze nei valori dell'attributo `lead_time` non influisce significativamente rispetto all'aggiunta di valori nulli o di *outlier* nel *dataset*.
Di conseguenza, possiamo concludere che i modelli Decision Tree e SVM sono abbastanza robusti rispetto all'inaccuratezza.

4.3.4 Aggiunta di righe duplicate

Dopo aver testato le prestazioni dei modelli, inserendo nei valori dell'attributo `lead_time` delle inconsistenze, sono state misurare le performance inserendo all'interno del *dataset* una quantità di righe duplicate sempre più crescente partendo da un 10% delle istanze fino alla loro completa duplicazione (100%).
Si sottolinea il fatto che, duplicando le varie righe del *dataset*, quello che andrà ad impattare il modello analizzato sarà solamente la presenza di istanze duplicate all'interno del *training set* utilizzato per l'addestramento.
Sarà invece successivamente utilizzato lo stesso *test set* per la valutazione delle metriche prestazionali, che non presenta duplicati.
Analizzando le prestazioni relative all'**Albero Decisionale** applicato a diversi livelli di righe duplicate nel *trainset*, i risultati suggeriscono che si ha un impatto limitato sulle misure di *performance* del modello di Decision Tree.
Non sembra esserci un miglioramento significativo nella *precision*, mentre il *recall* e l'*F1-score* per la classe 1 mostrano un aumento apprezzabile con il 10% di istanze duplicate e rimangono successivamente stabili negli altri livelli di duplicazione.
Di contro, l'accuratezza complessiva del modello sembra migliorare leggermente all'aumentare delle righe duplicate.
L'*AUC* (*Area Under the Curve*), invece, rimane invariata per tutti i livelli di duplicazione.

Questo può essere dovuto dal fatto che l'aggiunta di righe duplicate non impatta significativamente sulla capacità del modello di classificare correttamente le diverse classi.

Per quanto riguarda il modello della **Support Vector Machine**, l'analisi dei risultati rivela un'interessante tendenza: l'aumento della percentuale di righe duplicate nel *trainset*, ha un impatto limitato sulle prestazioni del modello.
Nonostante alcune fluttuazioni, le metriche di *precision, accuracy* e *AUC* rimangono sostanzialmente stabili.
Una possibile spiegazione di questo fenomeno è legata al funzionamento interno delle SVM stesse: questi modelli cercano di massimizzare il margine presente tra i punti di due classi diverse.
Aggiungere righe duplicate nel *trainset* non introduce informazioni aggiuntive, poiché tali duplicati saranno identici alle istanze già presenti.
Di conseguenza, il modello SVM non viene influenzato significativamente dalla presenza di righe duplicate, in quanto non forniscono informazioni aggiuntive per il migliorare la separazione delle classi.
Di contro, un'eccessiva presenza di righe duplicate può portare ad un aumento della complessità computazionale, poiché il modello dovrà elaborare un maggior numero di istanze.
Pertanto, nonostante le prestazioni del modello SVM rimangano relativamente stabili, l'aggiunta di molte righe duplicate porterà ad un aumento dei requisiti computazionali e un'eventuale riduzione dell'efficienza dell'addestramento.

4.3.5 Aggiunta di inconsistenze

Le metriche di qualità del *dataset* pulito indicano che esso presenta principalmente valori incoraggianti come nessun valore nullo e la limitata presenza di *outlier*.
Tuttavia, sono state individuate inconsistenze tra il numero di bambini e il numero di adulti presenti nelle prenotazioni.
In particolare, è stato stabilito che un caso di inconsistenza si verifica quando una prenotazione ha un numero di adulti pari a 0 e un numero di bambini maggiore o uguale a 1.
Questa situazione è considerata come una discrepanza nei dati poiché è difficile pensare che un bambino soggiornerà da solo non accompagnato da un adulto.
Anche in questo caso le inconsistenze verranno aggiunte solamente all'interno delle istanze del *trainset* utilizzato per addestrare i modelli di classificazione sfruttando la funzione `introduce_inconsistencies` spiegata precedentemente.
Con successiva predizione dei risultati sottoponendo un *testset* che non presenta alcuna forma di inconsistenza (lo stesso utilizzato per testare i modelli con il *dataset* pulito).
Per quanto riguarda entrambi i modelli decisionali analizzati, anche aggiungendo il 100% di inconsistenze, le metriche prestazionali non cambiano rispetto a quelle ottenute con il *dataset* pulito.
Nel caso degli Alberi Decisionali, questo risultato può essere spiegato dal fatto che le

colonne `no_of_adults` e `no_of_children` non sono particolarmente rilevanti ai fini della classificazione delle istanze all'interno del *dataset*.
Infatti, osservando il grafo che raffigura la struttura del Decision Tree, si nota che i valori degli attributi modificati con l'aggiunta di inconsistenze non sono mai presenti all'interno delle soglie di classificazione dei vari nodi dell'albero.
L'unico cambiamento osservato è dato dall'importanza assunta dai vari attributi.
Si nota che applicando il 30% di inconsistenze all'interno del grafico di *feature importance* i due attributi `no_of_adults` e `no_of_children` acquisiscono maggior spessore per la classificazione delle varie istanze.
Però, all'aumentare della quantità di inconsistenze all'interno del *dataset* l'importanza di questi due attributi diminuisce.
Fino al raggiungimento del 100% di inconsistenze dove i due attributi perdono totalmente importanza.

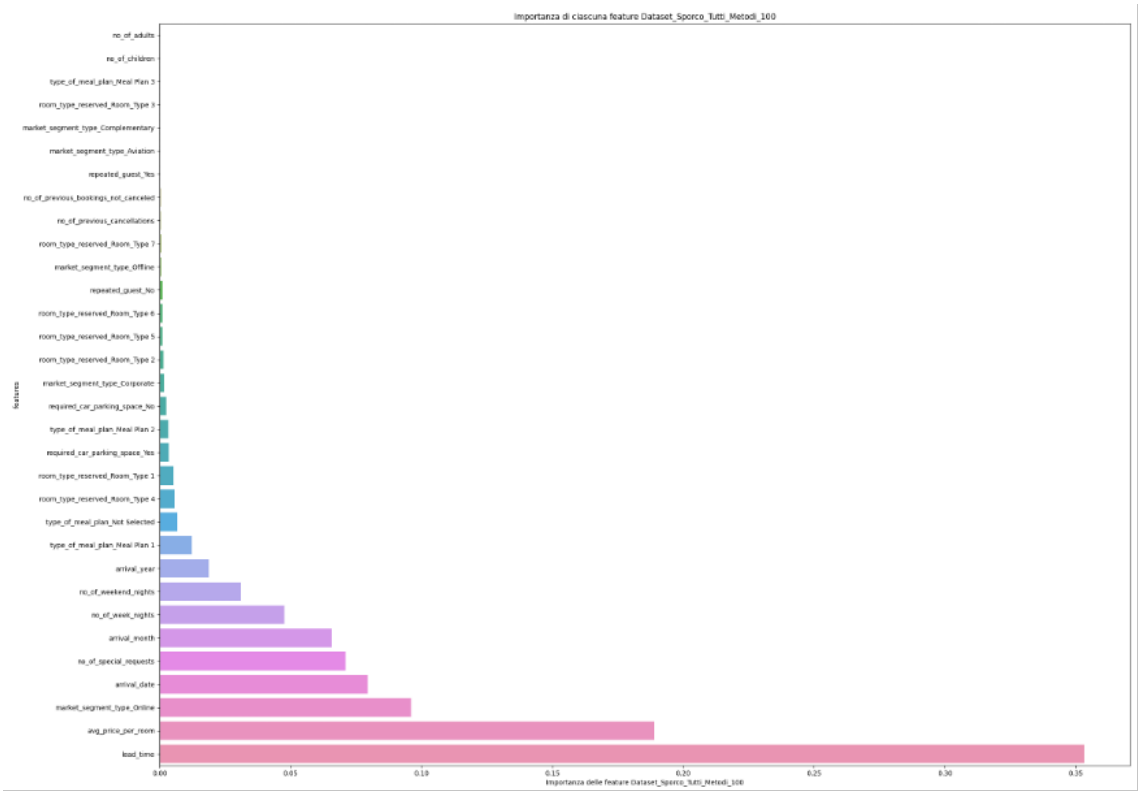


Figura 4.11: Grafico di feature importance con 100% di inconsistenze.

4.3.6 Modifica combinata del dataset con vari metodi

Fino ad ora, sono stati analizzati gli effetti di vari metodi da applicare al *dataset* eseguiti separatamente come l'introduzione di valori nulli, *outliers*, inaccuratezze, righe duplicate e inconsistenze, con l'obbiettivo di analizzare le *performance* in presenza di dati sporchi relative ai due modelli decisionali presi in considerazione.
Tuttavia, per ottenere una visione completa, è necessario studiare l'effetto combinato di tali metodi applicati contemporaneamente su diversi attributi del *dataset*.
Di fatto, i dati reali spesso contengono rumore, errori di misurazione e altre forme di deterioramento che possono compromettere le prestazioni dei modelli.
Pertanto, è importante comprendere come l'aumento del livello di deterioramento dei dati nel set di addestramento influenzi i modelli di classificazione adottati: Albero Decisionale (Decision Tree) e Support Vector Machine (SVM).
Per garantire una simulazione accurata nel deterioramento delle istanze del *dataset*, simulando un *dataset* reale, è stata adottata una strategia basata sull'importanza di ciascun attributo, come evidenziato dal grafico di *feature importance*, derivante da un modello di Albero Decisionale.
Ogni funzione di deterioramento dei dati implementata nel progetto è stata applicata selettivamente agli attributi con maggiore importanza per la classificazione, nonché ad alcuni attributi di importanza secondaria.
L'idea alla base di questa scelta è che, nel mondo reale, gli attributi più rilevanti per la classificazione di solito presentano un impatto significativo sulla capacità predittiva del modello: sporcando sia gli attributi di grande importanza che quelli di minore importanza, si è cercato di simulare uno scenario realistico in cui non solo gli attributi chiave vengono danneggiati, ma anche quelli meno influenti per la classificazione.
Osserveremo le misure di *precision, recall, F1-score, accuracy* e *AUC* confrontando successivamente le metriche prestazionali ottenute rispetto a quelle misurate con i modelli addestrati sul *dataset* pulito.
I risultati sperimentali hanno mostrato che entrambi i modelli sono influenzati dall'aumento del livello di sporco nel set di addestramento, sebbene con andamenti leggermente diversi.
Per quanto riguarda l'Albero Decisionale, si osservano lievi oscillazioni nelle prestazioni all'aumentare del deterioramento dei dati.
Le diminuzioni più consistenti si riscontrano nelle metriche di *precision, recall* e *F1-score* per la classe 1.
Le metriche *accuracy* e *AUC* tendono a diminuire leggermente all'aumentare del deterioramento dei dati.
I risultati ottenuti suggeriscono che l'Albero Decisionale è sensibile al rumore presente nei dati di addestramento, con un impatto che riguarda maggiormente la classe target 1.
Con oltre il 40% di dati deteriorati nel set di addestramento, i valori di *precision, recall*

e *F1-score* si stabilizzano attorno al valore di 0,65, mantenendo un andamento simile fino al raggiungimento del 75% dei dati sporchi.
Tuttavia, con l'aggiunta dell'85% dei valori deteriorati, si osserva un netto peggioramento delle metriche di *recall, F1-score*, nonché di *accuracuy* e *AUC*.
Per la SVM, si è osservato un declino più consistente delle prestazioni all'aumentare della quantità di dati deteriorati.
L'andamento della metrica di *precision* tende al deterioramento per entrambe le classi del target all'aumentare graduale del livello di sporco nel *dataset*.
In particolare, quando i dati deteriorati vengono gradualmente aggiunti, si osserva una diminuzione progressiva del valore di *precision* fino al raggiungimento del 25% di valori sporchi.
A partire dal 30% in poi, la *precision* per la classe 1 si stabilizza attorno al valore di 0,60, mantenendosi costante con lievi fluttuazioni: questo significa che al crescere del deterioramento delle istanze, il modello raggiunge una sorta di stabilità nell'identificare correttamente gli esempi positivi.
La metrica di *recall* presenta un andamento variabile per la classe 0 e una diminuzione costante per la classe 1.
Il diminuire della metrica di *recall*, indica che il modello sta classificando in modo errato un numero crescente di casi positivi appartenenti alla classe 1 relativamente a tutte le istanze del *testset*.
Anche il valore di *F1-score* segue un andamento simile alla *recall*.
Infine, ad ogni aumento della percentuale di dati deteriorarti, accuratezza e area sottesa alla curva (AUC) tendono a diminuire leggermente.
Questi risultati indicano che la SVM è influenzata dal rumore nei dati di addestramento, con un impatto significativo sulle prestazioni del modello che si dimostrano più suscettibili al deterioramento dei dati rispetto all'Albero Decisionale rendendo più difficile la corretta classificazione delle istanze.
In generale, i risultati suggeriscono che l'aumento del livello di sporco nel set di addestramento può influenzare negativamente le prestazioni di entrambi i modelli, senza riscontrare un andamento prevedibile o lineare.

5 Conclusioni

5.1 Descrizione dei risultati ottenuti

Dall'analisi condotta sui modelli degli Alberi Decisionali e Support Vector Machine, in presenza di un deterioramento delle istanze del *trainset* utilizzato per addestrare i modelli di *Machine Learning*, è emerso che questi sono influenzati negativamente dalla presenza di valori nulli, *outlier*, inaccuratezze, righe duplicate e inconsistenze.
Tuttavia, l'impatto e il grado di deterioramento delle metriche prestazionali variano a seconda del tipo di deterioramento introdotto e della quantità con cui si manifesta.
L'aggiunta di valori nulli nell'attributo `lead_time` ha un impatto significativo sul comportamento e sulle prestazioni dei modelli di *Machine Learning* utilizzati.
Dalle prove effettuate si nota che, sia l'Albero Decisionale che la SVM, sono abbastanza robusti fino al raggiungimento del 40% di valori nulli nell'attributo `lead_time`.
Oltre questa soglia, le prestazioni iniziano a deteriorarsi notevolmente, con un peggioramento localizzato nelle metriche di *precision, accuracy* e *AUC*.
Entrambi i modelli mostrano un degrado significativo delle prestazioni e faticano nell'eseguire la discriminazione tra le classi target, non riuscendo a prevedere in modo affidabile la cancellazione delle prenotazioni.
Inoltre, l'attributo `lead_time` perde importanza ai fini della classificazione all'aumentare dei valori nulli, di conseguenza, in risposta a questo cambiamento, l'Albero Decisionale modifica la sua struttura sostituendo la soglia di classificazione presente nel nodo radice con un altro attributo per effettuare le previsioni delle istanze.
La SVM si dimostra meno resistente alla presenza di valori nulli, mostrando un deteriora-

mento più significativo delle prestazioni rispetto all'Albero Decisionale; infatti, si osserva un peggioramento più significativo di tutte le metriche rispetto al modello addestrato utilizzando il *dataset* pulito.
L'efficacia delle SVM è fortemente influenzata dalla presenza di valori nulli, poiché la loro dipendenza dalla separabilità lineare dei dati li rende meno robusti.
Di conseguenza, al crescere dei valori nulli nell'attributo `lead_time`, si osserva un continuo deterioramento delle prestazioni del modello SVM.
In merito all'aggiunta di *outlier*, l'Albero Decisionale sembra essere più robusto rispetto alla SVM.
Per quanto riguarda il modello degli Alberi Decisionali, inizialmente si è osservato un leggero miglioramento delle prestazioni con l'introduzione del 10% di valori anomali.
Ciò può essere attribuito alla robustezza intrinseca degli Alberi Decisionali rispetto agli *outlier*.
Tuttavia, superato il 20% di valori atipici, si verifica un deterioramento delle prestazioni, con una riduzione della capacità del modello di discriminare correttamente le istanze positive.
Ciò indica che un aumento eccessivo o mal posizionato degli *outlier* può influenzare negativamente la capacità del modello di identificare correttamente gli esempi positivi.
La SVM mostra un deterioramento più marcato delle prestazioni all'aumentare della percentuale di *outlier* nell'attributo `lead_time`: l'accuratezza è diminuita gradualmente, mentre le metriche come *precision, recall* e *F1-score* hanno mostrato un calo significativo per la classe "canceled".
Questo indica che il modello SVM ha faticato a prevedere correttamente le prenotazioni cancellate quando la presenza di *outlier* era elevata.
Quanto riportato suggerisce che l'aggiunta di valori *outlier* nell'attributo `lead_time` ha un impatto negativo sulle prestazioni di entrambi i modelli, tuttavia, gli Alberi Decisionali si sono dimostrati più robusti alla presenza di *outlier* rispetto alle SVM.
Per le inaccuratezze, sia l'Albero Decisionale che la SVM mostrano una certa resistenza fino al 30% di inconsistenze.
Oltre questa soglia, si osserva un peggioramento più marcato delle metriche prestazionali, con un calo significativo delle metriche di *precision, accuracy* e *AUC*.
Il modello Decision Tree ha mostrato una maggiore difficoltà nel distinguere correttamente le istanze positive con conseguenti modifiche nella struttura dell'Albero Decisionale tramite la sostituzione delle condizioni soglia dei nodi.
Anche il modello SVM ha mostrato una tendenza simile.
Fino al 30% di inaccuratezze, le prestazioni sono rimaste relativamente stabili, con lievi variazioni delle metriche prestazionali.
Tuttavia, superato il 50% di inconsistenze, si è verificato un peggioramento delle prestazioni, seppur meno significativo rispetto al modello degli Alberi Decisionali.

Con l'aggiunta dell'80% di inaccuratezze, sia il modello degli Alberi Decisionali che il modello SVM hanno mostrato un notevole peggioramento delle prestazioni, con una riduzione significativa di *precision, accuracy* e *AUC*.
Tuttavia, il modello SVM sembra aver reagito leggermente meglio rispetto al modello degli Alberi Decisionali, mantenendo un valore di *precision* superiore.
Per quanto riguarda le righe duplicate, entrambi i modelli sembrano essere abbastanza stabili e poco influenzati dalla presenza di duplicati nel set di addestramento.
Sebbene si osservi un leggero miglioramento delle metriche prestazionali per il modello degli Alberi Decisionali con il 10% di righe duplicate, le prestazioni rimangono generalmente stabili al variare della percentuale di duplicazione.
Allo stesso modo, il modello SVM ha mostrato prestazioni stabili, con alcune fluttuazioni, indipendentemente dalla presenza di righe duplicate.
Tuttavia, è importante constatare che un'eccessiva presenza di righe ripetute porta ad un aumento della complessità computazionale durante l'addestramento dei modelli, in seguito alla necessità elaborare un maggior numero di istanze comportando ad un aumento dei requisiti computazionali in termini di tempo e memoria.
In generale, anche con un aumento delle righe ripetute non vengono fornite informazioni deterioranti per la capacità di classificazione dei modelli.
Nel caso dell'aggiunta di inconsistenze nel numero di adulti e bambini presenti nelle prenotazioni, le metriche prestazionali dei modelli decisionali non sono cambiate rispetto al *dataset* pulito.
Questo risultato può essere dovuto dal fatto che gli attributi `no_of_adults` e `no_of_children` non sono rilevanti ai fini della classificazione delle istanze.
L'analisi della struttura dell'Albero Decisionale ha mostrato che i valori delle colonne modificate con l'aggiunta di inconsistenze non sono mai stati utilizzati come soglie di classificazione all'interno dei nodi dell'albero.
L'unico cambiamento rilevato è stata l'importanza attribuita a questi due attributi nel grafico di *feature importance*.
All'aumentare della percentuale di inconsistenze nel *dataset*, l'importanza dei due attributi diminuisce, fino al raggiungimento del 100% di inconsistenze dove perdono totalmente importanza.
Infine, l'applicazione combinata di diverse funzioni per il deterioramento dei dati, ha evidenziato un progressivo peggioramento delle prestazioni dei modelli, con una diminuzione delle metriche di *precision, recall, F1-score, accuracy* e *AUC*.
L'Albero Decisionale risulta sensibile all'aggiunta di rumore nei dati di addestramento, con un impatto più significativo sulla classe target 1.
Per quanto riguarda la SVM è ancora più suscettibile al deterioramento dei dati rispetto al modello Decision Tree, con una marcata diminuzione delle prestazioni al crescere della quantità di dati deteriorati.

In generale, l'Albero Decisionale sembra essere più resistente alla presenza di dati sporchi rispetto alla SVM.
Tuttavia, entrambi i modelli presentano un deterioramento delle prestazioni all'aumentare della percentuale di dati deteriorati nel set di addestramento anche se, la perdita di prestazioni è legata al tipo e alla quantità di sporco introdotta. In conclusione, questi risultati sottolineano l'importanza di avere un *dataset* pulito e affidabile per addestrare modelli di *Machine Learning*.
È fondamentale prestare attenzione alla qualità dei dati e utilizzare metodi adeguati per la pulizia nonché per la gestione dei dati sporchi al fine di ottenere prestazioni ottimali dai modelli di classificazione adottati.