Progetto per la predizione degli affitti

Gruppo di lavoro

Petruzzella Simone,758426, s.petruzzella8@studenti.uniba.it

Link github:

https://github.com/SimonePet/Progetto_Icon

A.A 2023-2024

Indice:

1. Introduzione
2. Creazione del dataset
3. Ragionamento logico e Prolog
4. Apprendimento non Supervisionato
5. Rete bayesiana
6. Apprendimento non supervisionato

1.Introduzione

Il progetto si propone di sfruttare il dataset degli affitti in Germania per eseguire tre task principali:

1. **Predizione del Prezzo dell'Affitto**: Utilizzando l'apprendimento supervisionato, si sviluppano modelli per prevedere se l'affitto è considerato economico, medio o costoso.
2. **Cluster degli Affitti**: Si applicano tecniche di clustering per raggruppare le proprietà in base a caratteristiche comuni, facilitando la segmentazione del mercato.
3. **Scoperta di Correlazioni con una Rete Bayesiana**: Si utilizza una rete bayesiana per identificare le relazioni tra le diverse caratteristiche degli affitti, individuando i fattori che influenzano maggiormente il prezzo.

Questi task forniscono strumenti per la predizione, la segmentazione e una comprensione approfondita del mercato degli affitti in Germania.

2. Creazione del dataset

Il dataset utilizzato in questo progetto contiene un ampio insieme di dati su proprietà in affitto in Germania, con un totale di 268.850 righe e 49 colonne.

Le feature di input sono:

- regio1: rappresenta lo stato federale in cui è situata la proprietà.
- serviceCharge: spese accessorie come elettricità o internet (in €).
- heatingType: tipo di riscaldamento utilizzato nell'immobile.
- telekomTvOffer: indica se è incluso un servizio di TV a pagamento e quale offerta.
- telekomHybridUploadSpeed: velocità di upload dell'internet ibrido.
- newlyConst: indica se l'edificio è di nuova costruzione.

- balcony: presenza di balcone nell'immobile.
- picturecount: numero di immagini caricate nell'annuncio.
- pricetrend: tendenza dei prezzi, calcolata da Immoscout.
- telekomUploadSpeed: velocità di upload dell'internet.
- totalRent: affitto totale, solitamente comprensivo di affitto base, spese accessorie e riscaldamento.
- yearConstructed: anno di costruzione dell'edificio.
- scoutId: ID dell'immobile su Immoscout.
- noParkSpaces: numero di posti auto disponibili.
- firingTypes: fonti di energia principali, separate da due punti.
- hasKitchen: presenza di cucina nell'immobile.
- geo_bln: stato federale, equivalente a regio1.
- cellar: presenza di cantina nell'immobile.
- yearConstructedRange: anno di costruzione raggruppato in intervalli da 1 a 9.
- baseRent: affitto base, senza elettricità e riscaldamento.
- houseNumber: numero civico dell'immobile.
- livingSpace: superficie abitabile in metri quadrati.
- geo_krs: distretto o circondario, livello superiore rispetto al codice postale.
- condition: stato dell'appartamento.
- interiorQual: qualità degli interni.
- petsAllowed: permesso per animali domestici (può essere sì, no o negoziabile).
- street: nome della strada.
- streetPlain: nome della strada, con formattazione diversa.
- lift: presenza di ascensore nell'edificio.
- baseRentRange: affitto base raggruppato in intervalli da 1 a 9.
- typeOfFlat: tipologia di appartamento.
- geo_plz: codice postale (ZIP code).
- noRooms: numero di stanze nell'appartamento.
- thermalChar: fabbisogno energetico in kWh/(m^2a), utilizzato per determinare la classe di efficienza energetica.
- floor: piano in cui si trova l'appartamento.

- numberOfFloors: numero di piani dell'edificio.
- noRoomsRange: numero di stanze raggruppato in intervalli da 1 a 5.
- garden: presenza di giardino.
- livingSpaceRange: superficie abitabile raggruppata in intervalli da 1 a 7.
- regio2: distretto o circondario (Kreis), equivalente a geo_krs.
- regio3: città o comune in cui si trova la proprietà.
- description: descrizione libera dell'immobile.
- facilities: descrizione libera delle strutture disponibili.
- heatingCosts: costi mensili di riscaldamento (in €).
- energyEfficiencyClass: classe di efficienza energetica (basata su thermalChar, deprecata da febbraio 2020).
- lastRefurbish: anno dell'ultima ristrutturazione dell'immobile.
- electricityBasePrice: prezzo base mensile per l'elettricità (in €, deprecato da febbraio 2020).
- electricityKwhPrice: prezzo dell'elettricità per kWh (deprecato da febbraio 2020).
- date: data di raccolta dei dati.

Queste sono tutte le features input di partenza. La prima operazione effettuata è stata quella di eliminare alcune colonne superflue che non sarebbero state utili per i vari task prefissati.

Le colonne eliminate sono:

- date
- serviceCharge
- heatingType
- telekomHybridUploadSpeed
- telekomUploadSpeed
- electricityBasePrice
- picturecount
- pricetrend
- scoutId
- firingTypes
- yearConstructedRange
- baseRentRange

- thermalChar
- noRoomsRange
- livingSpaceRange
- electricityKwhPrice
- energyEfficiencyClass
- lastRefurbish
- heatingCosts
- telekomTvOffer€
- baseRent
- geo_bln
- regio3
- geo_plz
- description
- facilities

La seconda operazione è stata l'eliminazione di tutte le righe del dataset contenenti almeno un valore nullo, per garantire la qualità e la coerenza dei dati utilizzati nelle analisi.

Successivamente, è stata applicata una selezione dei dati in base ai prezzi degli affitti, rimuovendo gli immobili con canoni considerati troppo alti o troppo bassi. Dopo aver esaminato i valori massimi e minimi, è stato stabilito un intervallo di prezzo compreso tra 400€ e 10.000€. Solo gli affitti rientranti in questo range sono stati mantenuti, assicurando così che l'analisi si concentri su un mercato realistico e rilevante. Un'altra operazione effettuata è stata l'eliminazione di una riga contenente un valore completamente errato, ovvero una proprietà in affitto in un edificio di 730 piani.

L’ultima operazione è stata quella di filtrare il dataset per considerare solo gli affitti situati in un determinato stato federale, ovvero quello della "Nordrhein-Westfalen". Questo ha permesso di ottenere un dataset più specifico, contenente 5329 righe.

3.Ragionamento logico e Prolog

Nonostante la notevole quantità di dati presenti nel dataset originale riguardante le proprietà in affitto, per i task di predizione che intendiamo affrontare è essenziale introdurre nuove feature più significative.

Le feature aggiunte sono:

- Tipologia_Affitti: può assumere tre valori: economico, medio e costoso, e rappresenta la fascia di prezzo dell'affitto basata sulla media degli affitti nella città corrispondente.
- Tipologia_Proprietà: può assumere quattro valori e classifica le proprietà in monolocale, bilocale, trilocale e appartamenti per famiglie.

Queste nuove informazioni ci permetteranno di migliorare le analisi e le previsioni sui dati degli affitti.

Per creare queste nuove colonne, ho sviluppato una base di conoscenza in Prolog, composta da fatti e regole. I fatti rappresentano informazioni concrete sulle proprietà e sugli affitti, mentre le regole ci permettono di inferire nuove informazioni in base ai dati esistenti. Questo approccio ci consente di arricchire il dataset con feature significative e migliorando le nostre capacità analitiche e predittive.

Per aggiungere i fatti alla nostra base di conoscenza, è stata utilizzata una funzione che genera un singolo fatto per ogni proprietà in affitto. Questa funzione estrae le informazioni rilevanti dal dataset e le organizza in una forma che Prolog può interpretare. In questo modo, ogni proprietà è rappresentata come un fatto, permettendo di sfruttare le regole della base di conoscenza per inferire nuove informazioni e classificare gli affitti e le tipologie di proprietà in modo efficiente.

Le regole sono state definite nel seguente modo:

```
regole = """
    media_affitti_citta(Citta, Media) :-
        findall(PrezzoAffitto, proprieta_affitto(_, _, PrezzoAffitto, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, Citta), ListaAffitti),
        length(ListaAffitti, NumeroAffitti),
        NumeroAffitti > 0,
        sum_list(ListaAffitti, SommaAffitti),
        Media is SommaAffitti / NumeroAffitti.

    affitto_economico(ID,Citta, PrezzoAffitto,Media) :-
        proprieta_affitto(ID, _, PrezzoAffitto, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, Citta),
        media_affitti_citta(Citta, Media),
        PrezzoAffitto < Media - 150.

    affitto_costose(ID,Citta, PrezzoAffitto,Media) :-
        proprieta_affitto(ID, _, PrezzoAffitto, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, Citta),
        media_affitti_citta(Citta, Media),
        PrezzoAffitto > Media + 100.

    affitto_medie(ID,Citta, PrezzoAffitto,Media) :-
        proprieta_affitto(ID, _, PrezzoAffitto, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, Citta),
        media_affitti_citta(Citta, Media),
        PrezzoAffitto < Media + 100,
        PrezzoAffitto > Media - 150.

    monolocale(ID) :-
        proprieta_affitto(ID, _, _, Living_space, No_rooms, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _),
        No_rooms =:= 1,
        Living_space < 50.

    bilocale(ID) :-
        proprieta_affitto(ID, _, _, Living_space, No_rooms, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _),
        No_rooms =:= 2,
        Living_space >= 50,
        Living_space < 70.

    trilocale(ID) :-
        proprieta_affitto(ID, _, _, Living_space, No_rooms, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _),
        No_rooms =:= 3,
        Living_space >= 70,
        Living_space < 100.

    proprieta_per_famiglie(ID) :-
        proprieta_affitto(ID, _, _, Living_space, No_rooms, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _),
        No_rooms > 4,
        Living_space >= 100.
```

Utilizzando queste regole e due dizionari, siamo in grado di arricchire il dataset con nuove informazioni.

Il “dizionario_propieta” mantiene la correlazione tra gli ID delle proprietà e le stringhe “economico”, “medio” e “costoso”, a seconda dei risultati ottenuti dalle query. Allo stesso modo, il “dizionario_dimensione” stabilisce la correlazione tra gli ID delle proprietà e le stringhe “monolocale”, “bilocale”, “trilocale” e “per famiglie”, in base ai risultati delle stesse query.

Grazie a questi due dizionari, sono state aggiunte due nuove colonne al dataset originale, e il risultato finale è stato salvato in un file chiamato “nuovo_dataset.csv”. Questo processo ci consente di avere a disposizione un dataset più completo e utile per le analisi e le predizioni future.

4.Apprendimento non Supervisionato

Il clustering è una tecnica di apprendimento automatico non supervisionato utilizzata per raggruppare un insieme di oggetti in modo che gli oggetti all'interno di un gruppo siano più simili tra loro rispetto agli oggetti appartenenti a gruppi diversi. Lo scopo del clustering è scoprire strutture e pattern nascosti nei dati.

K-means è uno degli algoritmi di clustering più utilizzati. Funziona suddividendo un dataset in k gruppi (o cluster). L'algoritmo opera in modo iterativo per raggiungere la soluzione ottimale:

1. Inizializzazione: Si scelgono casualmente k punti dal dataset, che fungono da centri iniziali dei cluster.
2. Assegnazione: Ogni punto dati nel dataset viene assegnato al cluster il cui centro è il più vicino, secondo una misura di distanza (ad esempio, distanza euclidea).
3. Aggiornamento: Si calcola il nuovo centro di ogni cluster come la media dei punti assegnati a quel cluster.
4. Ripetizione: I passaggi di assegnazione e aggiornamento vengono ripetuti fino a quando i centri dei cluster non cambiano più o fino a quando viene raggiunto un numero massimo di iterazioni.

Utilizzeremo il K-means per raggruppare proprietà in affitto simili secondo alcune caratteristiche specifiche:

- yearConstructed
- balcony
- condition
- interiorQual
- hasKitchen
- lift
- garden
- livingSpace
- noRooms
- typeOfFlat
- floor
- numberOfFloors

- totalRent
- regio2

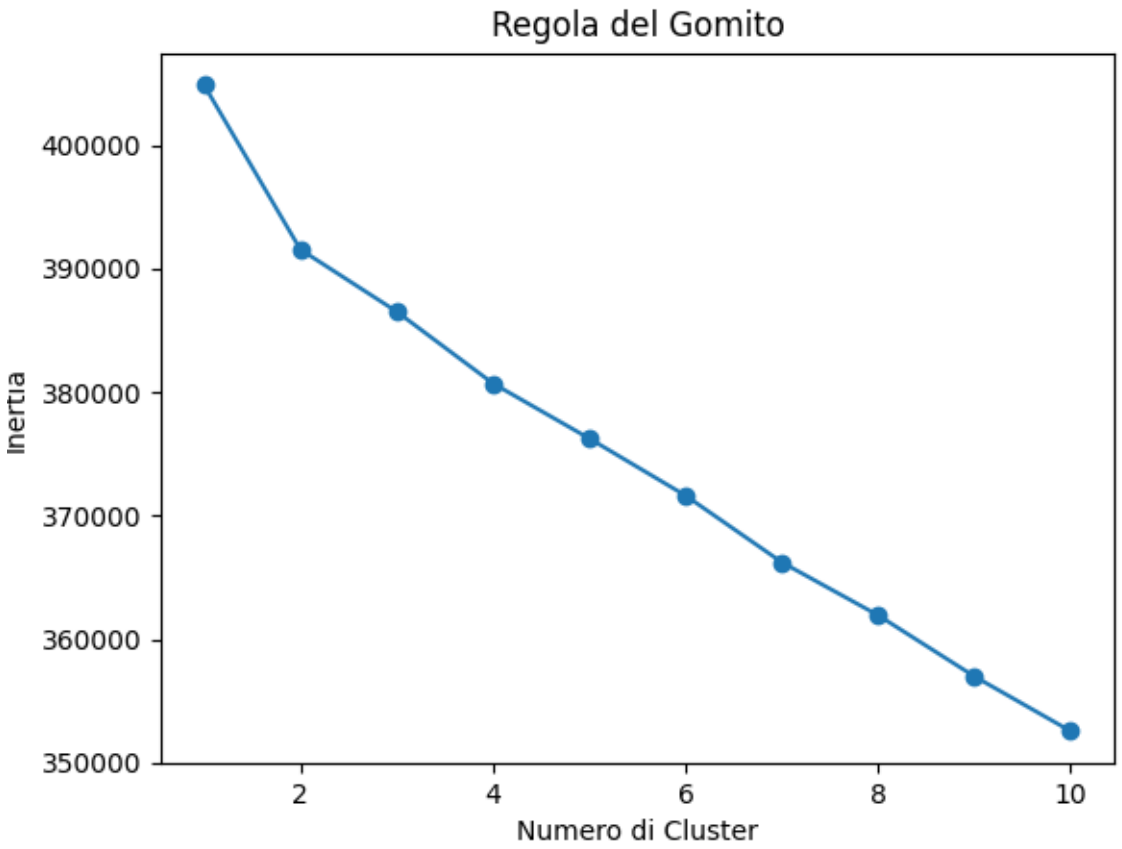
Le feature di carattere categorico vengono codificate sotto forma di interi.

Per trovare il numero ottimale di cluster (k) in un algoritmo di clustering K-means, viene applicata la tecnica del gomito. Questa tecnica consiste nell'eseguire K-means per una serie di valori di k e nel tracciare un grafico della somma delle distanze al quadrato (inerzia) tra i punti e i centroidi dei cluster a cui appartengono. Si misura l'inerzia per ogni valore di k e si osserva il grafico risultante. Il punto in cui il grafico mostra una riduzione evidente ma poi si appiattisce è noto come "gomito". Questo punto rappresenta il numero di cluster ideale, suggerendo un compromesso ottimale tra la riduzione dell'inerzia e la semplicità del modello.

Regola del gomito:

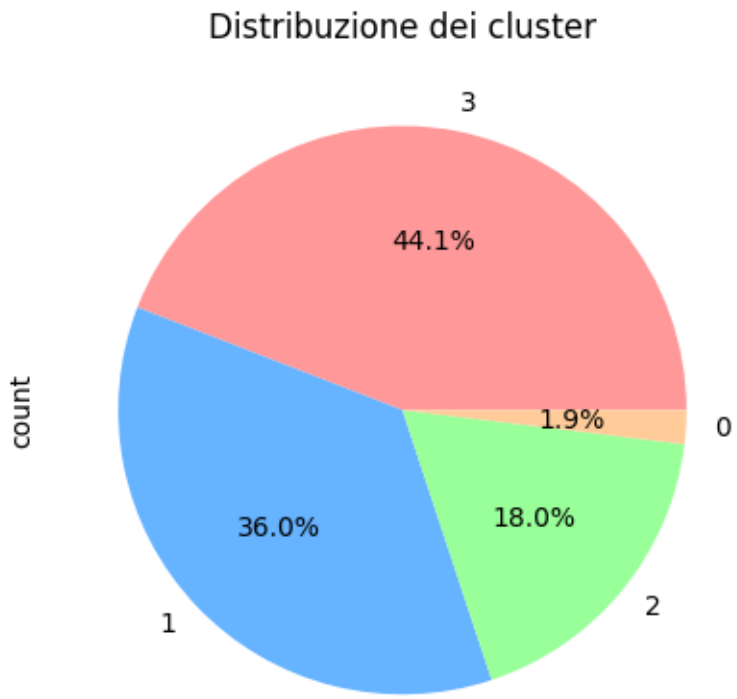
```python
def regola_del_gomito(data):
    inertia = []
    k_range = range(1, 11)

    for k in k_range:
        kmeans = KMeans(n_clusters=k, init='random', n_init=10)
        kmeans.fit(data)
        inertia.append(kmeans.inertia_)

    kl = KneeLocator(k_range, inertia, curve='convex', direction='decreasing')
    plt.plot(k_range, inertia, marker='o')
    plt.xlabel('Numero di Cluster')
    plt.ylabel('Inertia')
    plt.title('Regola del Gomito')
    plt.show()

    return kl.elbow
```

Grafico generato dalla regola del gomito



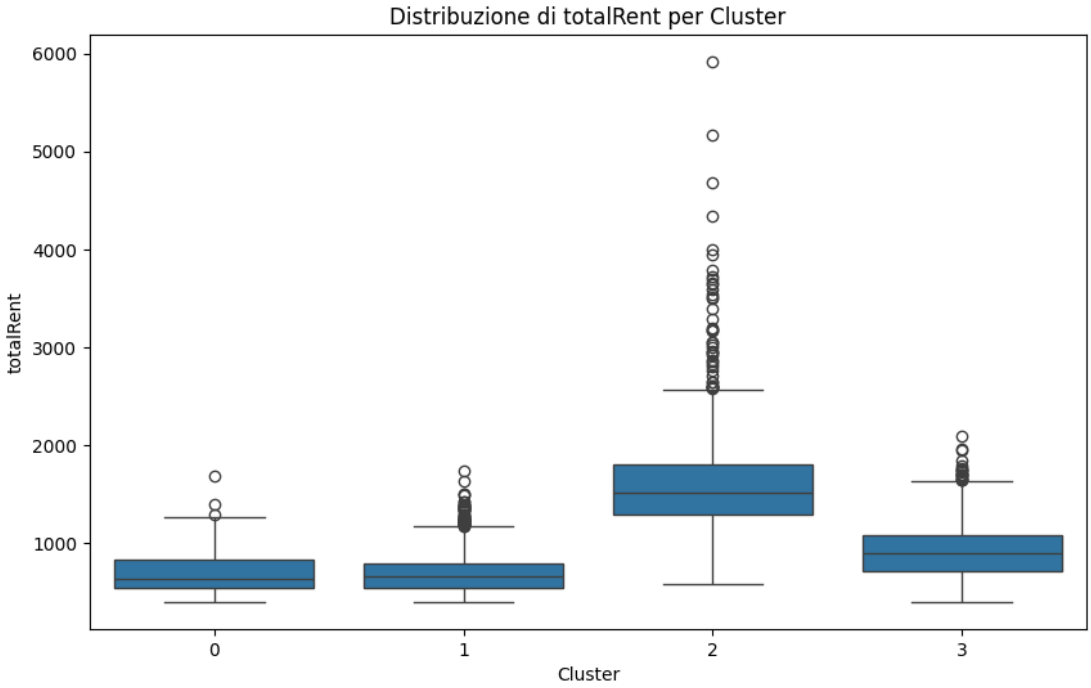
Il k ottimale è 4.Di conseguenza l'algoritmo di clustering raggrupperà le proprietà in affitto in 4 cluster.

Grafico dei cluster generato:

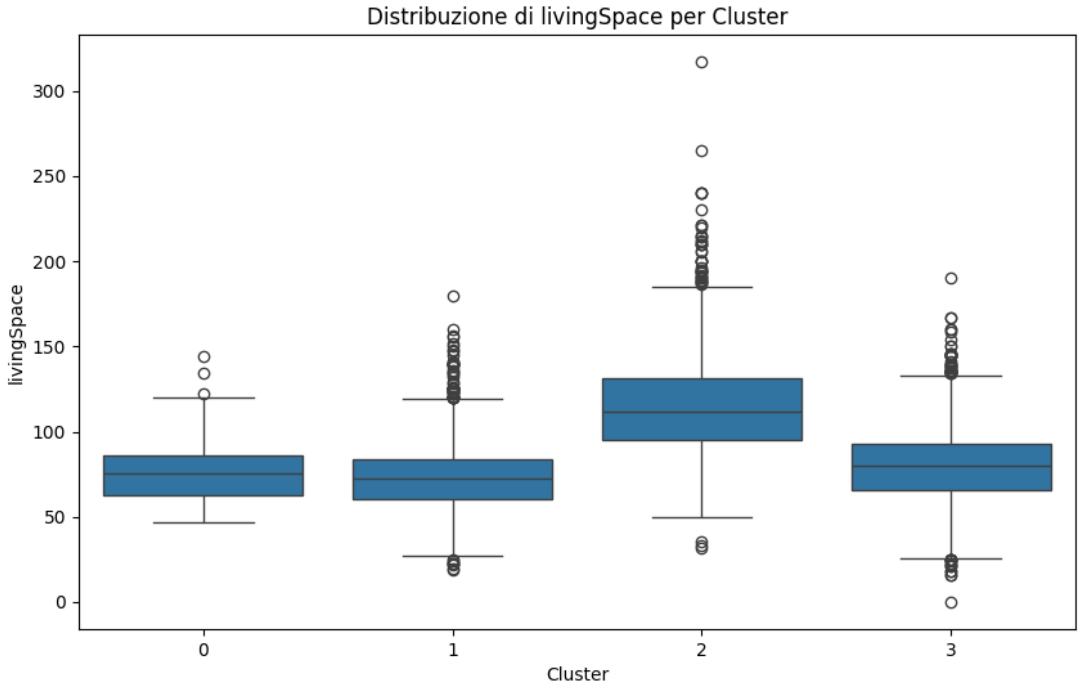


Oltre alla visualizzazione dei vari cluster, vengono generati anche grafici per ciascuna delle variabili considerate nel clustering. Questi grafici mostrano come i valori delle diverse variabili sono distribuiti all'interno di ciascun cluster. In questo modo, è possibile osservare le caratteristiche distintive di ogni cluster e capire meglio come le variabili influenzano la formazione dei gruppi, fornendo una rappresentazione visiva della distribuzione dei valori in relazione al cluster di appartenenza.

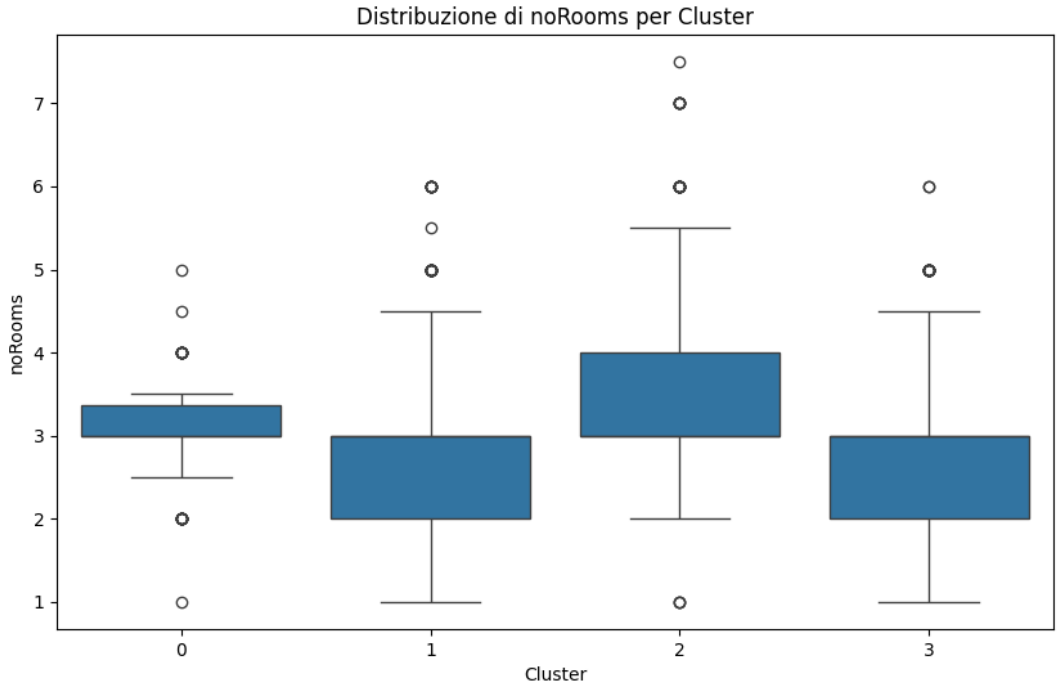
Esempio totalRent:



Esempio livingSpace:



Esempio noRooms:



5.Rete bayesiana

Il ragionamento probabilistico utilizza la teoria della probabilità per gestire l'incertezza, focalizzandosi sulle relazioni di dipendenza e indipendenza tra le variabili, così come sulla regola di Bayes. In questo contesto, si assegnano probabilità a diverse ipotesi ed eventi e si utilizzano le probabilità a posteriori per fare deduzioni e prendere decisioni. Le reti bayesiane sono un esempio di applicazione del ragionamento probabilistico. Queste reti sono rappresentate tramite grafi orientati aciclici (DAG), in cui ogni nodo corrisponde a una variabile e gli archi rappresentano le relazioni di dipendenza probabilistica tra tali variabili.

Il primo passo per costruire una rete bayesiana è stato discretizzare le variabili da includere nel grafo aciclico orientato (DAG). Inizialmente, ho tentato di utilizzare tutte le feature presenti nel dataset come input per la rete bayesiana.

Tuttavia, questo ha causato un eccessivo utilizzo della memoria RAM, portando al crash del programma.

Per affrontare questa problematica, ho scelto di selezionare solo alcune feature, al fine di semplificare i calcoli e diminuire il consumo di memoria RAM. Ho anche sostituito la variabile continua totalRent con una variabile categorica denominata Tipologia Affitti, precedentemente calcolata tramite Prolog. Questa variabile permette di classificare gli affitti in tre categorie: economico, medio e costoso.

Le variabili utilizzate come feature di input per la rete bayesiana sono:

- balcony;
- condition;
- interiorQual;
- hasKitchen;
- lift;
- livingSpace;
- noRooms;
- floor;
- Tipologia_Affitti;
- NewlyConst;
- typeOfFlat.

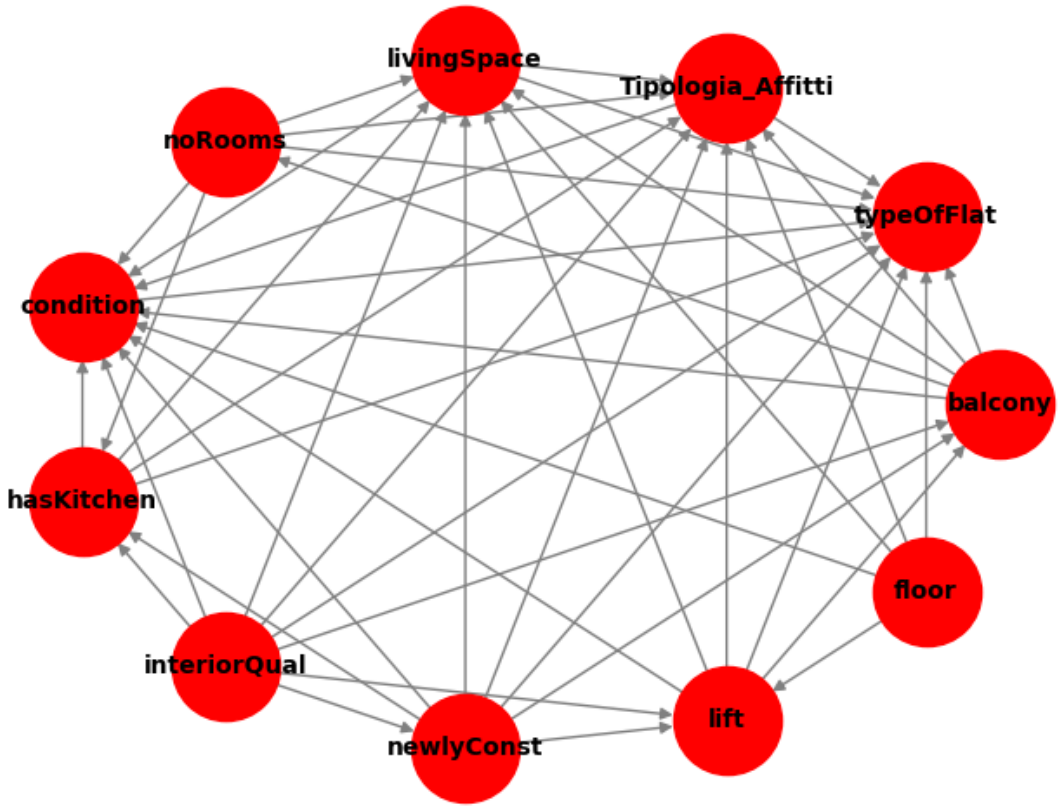
```
df = pd.read_csv(percorso_file_dataset)

# Seleziona le caratteristiche desiderate
features = ['balcony', 'condition', 'interiorQual', 'hasKitchen',
            'lift', 'livingSpace', 'noRooms', 'floor', 'Tipologia_Affitti', 'newlyConst','typeOfFlat']
df = df[features]

# Discretizza le variabili categoriche
for col in df.columns:
    if df[col].dtype == 'object':
        df[col] = pd.Categorical(df[col]).codes

# Discretizza le variabili booleane
for col in df.columns:
    if df[col].dtype == 'bool':
        df[col] = df[col].astype(int)

#discretizza il livingspace in 5 intervalli di grandezza uguale
df['livingSpace'] = pd.qcut(df['livingSpace'], q=5, labels=False)
#fai lo shuffle delle righe dl dataset
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
#elimina valori nulli
df = df.dropna()
```

Possiamo adesso passare alla creazione della rete bayesiana:

```
# Ricerca della struttura ottimale
hc_k2 = HillClimbSearch(train_df)
k2_model = hc_k2.estimate(scoring_method='k2score')

# Creazione della rete bayesiana
model = BayesianNetwork(k2_model.edges())
model.fit(train_df, estimator=MaximumLikelihoodEstimator, n_jobs=-1)

# Stampa i nodi e gli archi del modello
print("Nodi (variabili) nel modello:", model.nodes())
print("Archi (relazioni) nel modello:", model.edges())
```

Utilizzo un approccio di ricerca hill-climbing per determinare la struttura ottimale del grafo (ovvero quali variabili sono dipendenti da altre) e successivamente addestra il modello di rete bayesiana sui dati forniti. Infine, stampa i nodi e gli archi, che rappresentano rispettivamente le variabili del modello e le relazioni tra di esse.

Visualizzazione del grafico DAC



Possibile spiegazione dei collegamenti tra le feature:

- livingSpace (Spazio Abitativo): Maggiore spazio abitativo tende ad aumentare il costo dell'affitto.
- noRooms (Numero di Stanze): Più stanze sono generalmente associate a un costo di affitto più elevato.
- condition (Condizioni dell'Appartamento): Migliori condizioni dell'appartamento solitamente portano a un affitto più alto.
- hasKitchen (Presenza di Cucina): La presenza di una cucina moderna può aumentare il valore dell'affitto.
- interiorQual (Qualità degli Interni): Interni di alta qualità influiscono positivamente sul prezzo dell'affitto.
- newlyConst (Costruzione Recente): Appartamenti di nuova costruzione tendono ad avere affitti più elevati.
- Tipologia_Affitti (Tipologia di Affitto): Riflette la categoria di prezzo dell'affitto, influenzata dalle altre caratteristiche dell'appartamento.
- typeOfFlat (Tipo di Appartamento): Il tipo di appartamento (ad esempio, attico o monolocale) può influenzare il prezzo dell'affitto.
- balcony (Presenza di Balcone): Un balcone aggiunge valore all'appartamento, aumentando il costo dell'affitto.
- floor (Piano dell'Appartamento): Gli appartamenti ai piani più alti, specialmente con vista, tendono ad essere più costosi.
- lift (Presenza di Ascensore): Un ascensore rende l'appartamento più desiderabile, aumentando il valore dell'affitto.

Queste connessioni mostrano come diverse caratteristiche dell'appartamento influenzino il costo dell'affitto.

Una volta costruita la rete bayesiana, ho provato a effettuare delle previsioni sulla tipologia di affitto utilizzando le altre variabili di input. Ho deciso di testarla con un esempio generato casualmente.

```python
def crea_esempio(model):
    return model.simulate(n_samples=1).drop(columns=['Tipologia_Affitti'])
```

```
Esempio creato:
   lift  typeOfFlat  newlyConst  floor  noRooms  interiorQual  livingSpace  balcony  hasKitchen  condition
0     1           0           0    1.0      2.0             3            2        1           0          3
```

Ho stampato le probabilità che l'esempio creato appartenga ad una determinata tipologia di affitto: economico, medio o costoso.

Tipologia_Affitti	phi(Tipologia_Affitti)
Tipologia_Affitti(0)	1.0000
Tipologia_Affitti(1)	0.0000
Tipologia_Affitti(2)	0.0000

La predizione sembra essere affidabile poiché viene considerato costoso, ed ha le seguenti proprietà:

- lift: è presente l'ascensore;
- typeOfFlat: è un appartamento;
- newlyConst: non è una nuova costruzione;
- Floor: è al primo piano;
- noRooms: è formato da 2 stanze;
- interiorQual: è considerato sofisticato;
- livingSpace: in media una dimensione di 80 mq^2;
- balcony: sono presenti dei balconi;
- hasKitchen: non ha la cucina attrezzata;
- condition: ed è in ottime condizioni.

Come detto in precedenza tutte queste variabili confermano che si tratta di una proprietà costosa.

6.Apprendimento Supervisionato

All'interno di questo progetto, l'apprendimento supervisionato è stato utilizzato con l'obiettivo di prevedere la tipologia di affitto di un appartamento. Nello

specifico, il nostro obiettivo è predire se un appartamento rientrerà nella categoria di affitto "economico", "medio" o "costoso", in base a una serie di caratteristiche dell'appartamento (come superficie abitabile, numero di stanze, presenza di balcone o ascensore) e ad altre informazioni specifiche (come la qualità degli interni e lo stato di conservazione).

Il primo passo è stato discretizzare tutte le nostre feature di input e selezionare quelle da utilizzare per gli algoritmi di apprendimento supervisionato. Ho deciso di eliminare alcune delle feature di input che sembravano non contribuire significativamente alla capacità predittiva del modello.

Le variabili eliminate sono:

- Unnamed: 0;
- totalRent;
- ID;
- Regio1;
- Tipologia_Propieta;
- Street;
- houseNumber;
- streetplain;
- geo_krs;
- petsallowed;

La variabili restanti sono state discretizzate:

```
dataset['regio2'] = dataset['regio2'].astype('category').cat.codes
dataset['typeOfFlat'] = dataset['typeOfFlat'].astype('category').cat.codes
dataset['condition'] = dataset['condition'].astype('category').cat.codes
dataset['interiorQual'] = dataset['interiorQual'].astype('category').cat.codes
dataset['Tipologia_Affitti'] = dataset['Tipologia_Affitti'].astype('category').cat.codes
dataset['livingSpace'] = pd.qcut(dataset['livingSpace'], q=5, labels=False)
dataset['yearConstructed'] = pd.qcut(dataset['yearConstructed'], q=5, labels=False)
```

Per questo progetto ho deciso di utilizzare due modelli di apprendimento automatico:

DecisionTreeClassifier

Un Decision Tree Classifier (classificatore ad albero decisionale) è un algoritmo di apprendimento supervisionato utilizzato per la classificazione. Funziona

creando un albero in cui ogni nodo rappresenta una decisione basata su un attributo del dataset, mentre ogni ramo rappresenta l'esito di quella decisione. Le foglie dell'albero rappresentano le classi finali o le decisioni previste.

Una volta costruito, l'albero può essere utilizzato per classificare nuove istanze seguendo il percorso dalle radici fino a una foglia, basandosi sui valori degli attributi dell'istanza da classificare. I decision tree sono facili da interpretare e visualizzare, ma possono soffrire di overfitting se non regolati correttamente.

RandomForestClassifier

Il RandomForestClassifier è un algoritmo di apprendimento supervisionato utilizzato per la classificazione, che costruisce una "foresta" di molti alberi decisionali (decision trees). Ogni albero è addestrato su un diverso sottoinsieme casuale dei dati e su una selezione casuale di caratteristiche, un processo chiamato "bagging".

Il RandomForestClassifier è noto per la sua efficacia, facilità d'uso, e per essere meno sensibile ai dati anomali rispetto a un singolo albero decisionale.

Un altro passo cruciale per ottimizzare i nostri modelli di apprendimento automatico è la selezione degli iperparametri appropriati.

DecisionTree Hyperparametres

- **DecisionTree__max_depth [10,20,30] :** Definisce la profondità massima dell'albero. Limitare la profondità può aiutare a evitare l'overfitting, mentre una maggiore profondità può aumentare la complessità del modello e la sua capacità di apprendere dettagli dai dati.
- **DecisionTree__min_samples_split [2,5,10,20,50] :** Specifica il numero minimo di campioni richiesti per dividere un nodo interno. Valori più elevati possono ridurre l'overfitting, impedendo che l'albero diventi troppo complesso.
- **DecisionTree__min_samples_leaf [1,2,5,10,20] :** Indica il numero minimo di campioni richiesti per essere in un nodo foglia. Aumentare questo valore può portare a un albero più semplice e a una maggiore generalizzazione.
- **DecisionTree__criterion:** Determina la funzione utilizzata per misurare la qualità di una suddivisione. Le opzioni includono:

- 'gini': Misura l'impurità di Gini.
- 'entropy': Misura l'entropia dell'informazione.
- 'log_loss': Misura la log-loss (specifica per le decisioni di classificazione probabilistica).

RandomForest Hyperparameters

- **RandomForest__n_estimators [30,50,75] :** Definisce il numero di alberi nella foresta. Un numero maggiore di alberi può migliorare le prestazioni del modello riducendo la varianza, ma può aumentare anche il tempo di addestramento.
- **RandomForest__max_depth [30,50]:** Imposta la profondità massima di ciascun albero nella foresta. Limitare la profondità può prevenire l'overfitting, mentre una maggiore profondità consente agli alberi di apprendere dettagli più complessi dai dati.
- **RandomForest__min_samples_split [2,5,10,20] :** Specifica il numero minimo di campioni richiesti per dividere un nodo interno. Valori più elevati aiutano a ridurre l'overfitting impedendo che gli alberi diventino troppo complessi.
- **RandomForest__min_samples_leaf [1,2,5,10,20]:** Indica il numero minimo di campioni richiesti per essere in un nodo foglia. Aumentare questo valore può portare a una foresta più generalizzata e prevenire l'overfitting.
- **RandomForest__criterion**: Determina la funzione utilizzata per misurare la qualità di una suddivisione. Le opzioni includono:
  - 'gini': Misura l'impurità di Gini.
  - 'entropy': Misura l'entropia dell'informazione.
  - 'log_loss': Misura la log-loss (specifica per le decisioni di classificazione probabilistica).

Per trovare gli iperparametri ottimali, abbiamo adottato il metodo della Grid Search con Cross Validation. Questo approccio implica la definizione di un insieme predefinito di valori per ciascun iperparametro, creando una "griglia" di possibili combinazioni. La Cross Validation suddivide il dataset di addestramento in K sottoinsiemi (folds). Per ciascuna combinazione di iperparametri, il modello viene addestrato su K-1 folds e testato sul fold

rimanente. Questo processo viene ripetuto K volte, cambiando ad ogni iterazione il fold di test. La media delle prestazioni ottenute in tutte le iterazioni fornisce una valutazione complessiva delle performance del modello.

Abbiamo utilizzato la funzione GridSearchCV di scikit-learn per eseguire la ricerca degli iperparametri più efficaci per ciascun modello. GridSearchCV effettua la Cross Validation per ogni combinazione di iperparametri e misura le prestazioni del modello utilizzando l'errore quadratico medio negativo come metrica.

Al termine della valutazione, GridSearchCV identifica la combinazione di iperparametri che ha prodotto le migliori prestazioni durante la Cross Validation. Questa combinazione viene considerata la più adatta per il modello, in base al dataset e alla metrica utilizzata.

I parametri restituiti per il DecisionTree sono:

Parametro	Best Parametro
DecisionTree__max_depth	30
DecisionTree__min_samples_split	2
DecisionTree__min_samples_leaf	1
DecisionTree__criterion	log_loss

I parametri restituiti per il RandomForest sono:

Parametro	Best Parametro
RandomForest__max_depth	50
RandomForest__min_samples_split	2
RandomForest__min_samples_leaf	1
RandomForest__criterion	Log_loss
RandomForest__n_estimators	30

Fase di addestramento e test

Durante la fase di addestramento e test, i modelli sono stati addestrati utilizzando una Cross Validation con 10 fold, in considerazione della dimensione moderata del dataset.

```
# Use KFold for cross-validation
kFold = KFold(n_splits=10,random_state=42, shuffle=True)

model['DecisionTreeClassifier']['accuracy_list'] = cross_val_score(DecisionTreeModel, X, y, cv=kFold, scoring='accuracy', n_jobs=-1)
model['DecisionTreeClassifier']['precision_list'] = cross_val_score(DecisionTreeModel, X, y, cv=kFold, scoring='precision_macro', n_jobs=-1)
model['DecisionTreeClassifier']['recall_list'] = cross_val_score(DecisionTreeModel, X, y, cv=kFold, scoring='recall_macro', n_jobs=-1)
model['DecisionTreeClassifier']['f1'] = cross_val_score(DecisionTreeModel, X, y, cv=kFold, scoring='f1_macro', n_jobs=-1)
model['RandomForestClassifier']['accuracy_list'] = cross_val_score(RandomForestModel, X, y, cv=kFold, scoring='accuracy', n_jobs=-1)
model['RandomForestClassifier']['precision_list'] = cross_val_score(RandomForestModel, X, y, cv=kFold, scoring='precision_macro', n_jobs=-1)
model['RandomForestClassifier']['recall_list'] = cross_val_score(RandomForestModel, X, y, cv=kFold, scoring='recall_macro', n_jobs=-1)
model['RandomForestClassifier']['f1'] = cross_val_score(RandomForestModel, X, y, cv=kFold, scoring='f1_macro', n_jobs=-1)
```

Fase di valutazione

Essendo il mio un problema di classificazione ho usato le seguenti metriche:

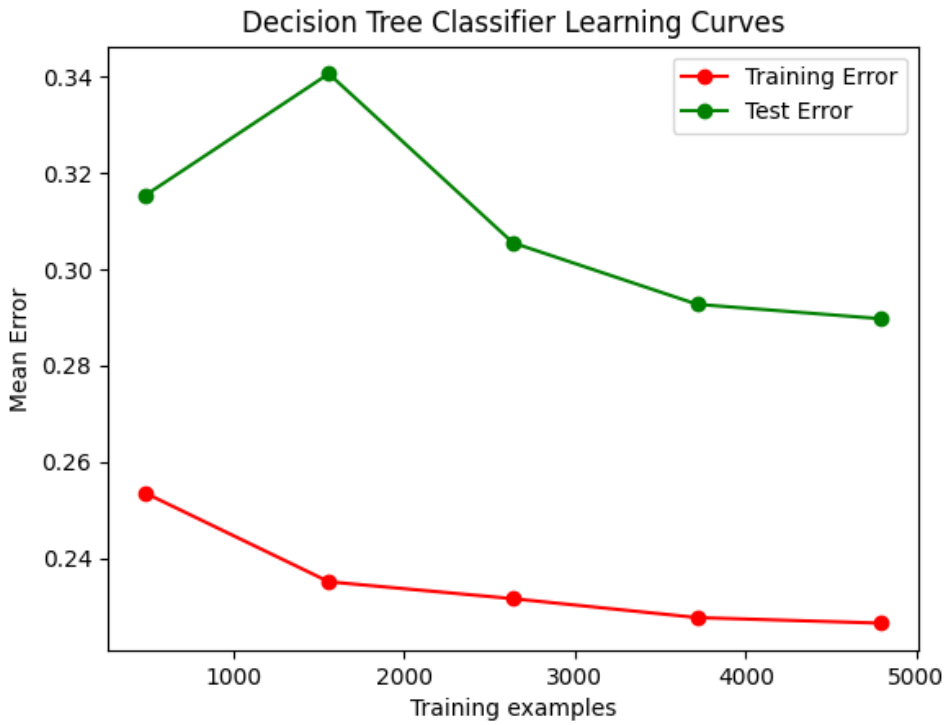
- **Accuracy**: Indica la proporzione di previsioni corrette sul totale delle previsioni effettuate. È una misura generale delle prestazioni di un modello di classificazione, ma può essere fuorviante se le classi sono sbilanciate.
- **Precision (macro)**: Misura la capacità del modello di non classificare in modo errato un campione negativo come positivo. La precisione macro calcola la precisione per ciascuna classe separatamente e poi fa la media aritmetica delle precisioni, trattando ogni classe allo stesso modo, indipendentemente dalla loro frequenza.
- **Recall (macro)**: Indica la capacità del modello di identificare correttamente tutti i campioni positivi. Come per la precisione macro, il recall macro è la media aritmetica del recall calcolato per ciascuna classe, trattando ogni classe in modo uguale.
- **F1 (macro)**: È la media armonica di precision e recall, utilizzata per bilanciare le due metriche. L'F1 macro calcola l'F1 score per ciascuna classe e poi fa la media, trattando tutte le classi con uguale importanza, indipendentemente dal loro numero di campioni.

Risultati ottenuti:

```
Decision Tree Classifier
Accuracy:  0.717763686890773
Precision:  0.7109927780959441
Recall:  0.7098215390207916
F1:  0.7103028479093794
```

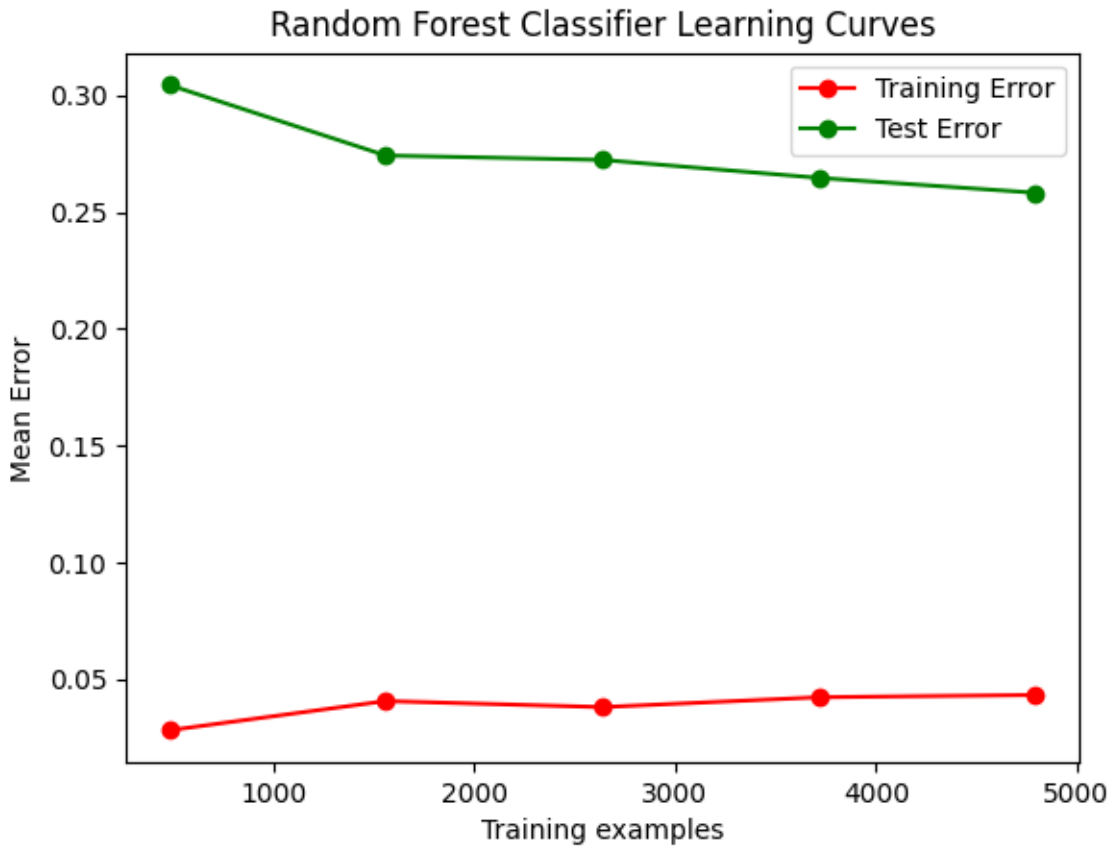
```
Random Forest Classifier
Accuracy:  0.7397237230035689
Precision:  0.7277449176626749
Recall:  0.7340755042191544
F1:  0.7278497577271318
```

I risultati ottenuti sono accettabili, ma potrebbero essere ulteriormente migliorati. Dopo averli analizzati, possiamo visualizzare le curve di apprendimento per entrambi i modelli per identificare le aree di miglioramento e ottimizzare le prestazioni.



Nel grafico precedente, l'errore di addestramento (curva rossa) diminuisce costantemente con l'aumento del numero di esempi di addestramento, raggiungendo un valore relativamente basso, mentre l'errore di test (curva verde) parte da un valore elevato, riducendosi solo parzialmente e stabilizzandosi a un livello significativamente più alto rispetto all'errore di addestramento.

Questa situazione indica che il modello si adatta molto bene ai dati di addestramento, ma non riesce a generalizzare adeguatamente sui dati di test, suggerendo un caso di overfitting.



Nel grafico del modello Random Forest, l'errore di addestramento (curva rossa) è molto basso e rimane quasi costante all'aumentare del numero di esempi di addestramento, indicando che il modello si adatta molto bene ai dati di training. L'errore di test (curva verde) è significativamente più alto, anche se tende a diminuire leggermente all'aumentare del numero di esempi.

Questa configurazione suggerisce che il modello potrebbe essere in una situazione di overfitting: il modello cattura molto bene i dettagli dei dati di addestramento ma non generalizza altrettanto bene sui dati di test, come dimostra il divario tra le due curve.

Varianza e deviazione standard

Per verificare la nostra ipotesi di overfitting basata sulle curve di apprendimento, possiamo esaminare i valori di varianza e deviazione standard del modello.

La varianza è una misura che quantifica quanto i dati di un insieme si discostano dalla loro media. Si calcola come la media dei quadrati delle differenze tra ciascun dato e la media.

La deviazione standard è la radice quadrata della varianza e fornisce una misura della dispersione dei dati rispetto alla media, ma nella stessa unità di misura dei dati stessi.

	DecisonTree	RandomForest
Deviazione Train	1.052895636641638e-05	7.7849314668119 69e-06
Deviazione Test	0.0004314230202006289	0.0003099586533014822
Varianza Train	0.0031609342774793236	0.002686279296711873
Varianza Test	0.020287358517246744	0.017387681661072896

Il Decision Tree mostra segni di overfitting, con una deviazione standard e varianza molto basse sui dati di addestramento e significativamente più alte sui dati di test, indicando una scarsa capacità di generalizzazione. Al contrario, il Random Forest ha deviazioni standard e varianze sui dati di addestramento e test più equilibrate, suggerendo una migliore capacità di generalizzazione. In sintesi, il Random Forest gestisce meglio il trade-off tra adattamento ai dati di addestramento e prestazioni sui dati di test rispetto al Decision Tree.

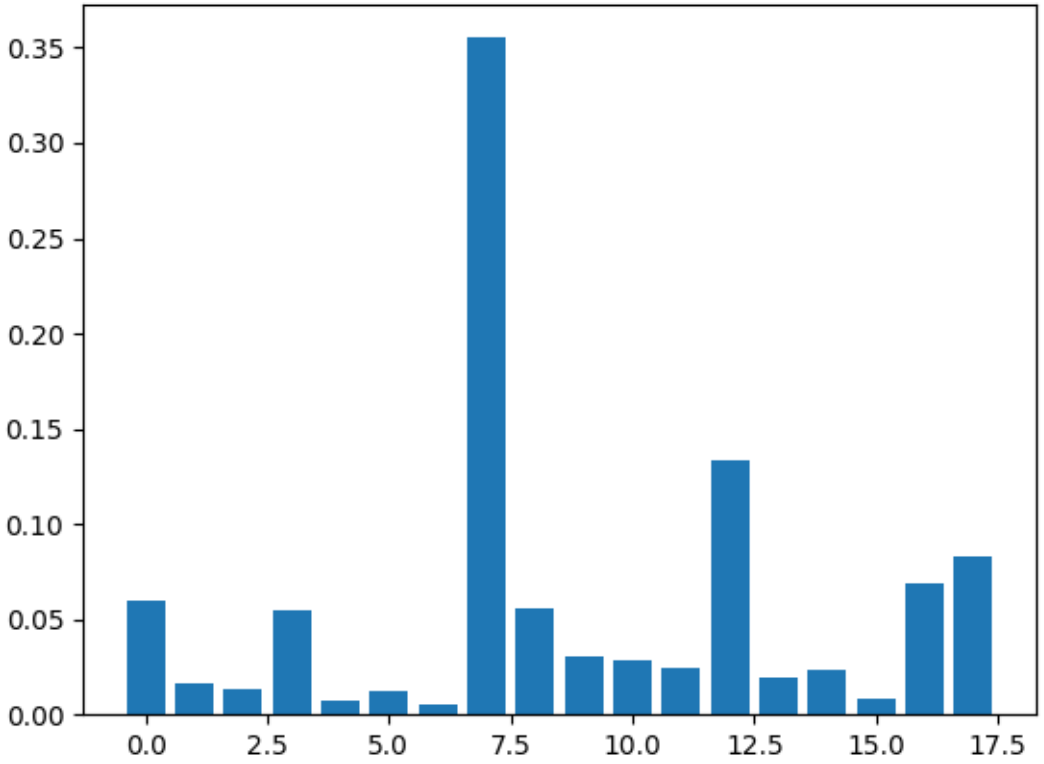
Soluzione overfitting

Per affrontare l'overfitting, si possono adottare due strategie principali. La prima consiste nell'utilizzare una metodologia di oversampling per bilanciare i valori nella colonna target, poiché dopo l'analisi i dati risultano sbilanciati.

```
def oversampling(dataset):
    df_majority = dataset[dataset['Tipologia_Affitti'] == 1]
    df_minor1 = dataset[dataset['Tipologia_Affitti'] == 0]
    df_minor2 = dataset[dataset['Tipologia_Affitti'] == 2]
    # Sottocampiona la classe maggioritaria
    df_minority_upsampled1 = resample(df_minor1,
                                      replace=True,  # Sostituzione per permettere il campionamento ripetuto
                                      n_samples=len(df_majority),  # Eguaglia la classe maggioritaria
                                      random_state=42)  # Per la riproducibilità
    df_minority_upsampled2 = resample(df_minor2,
                                      replace=True,  # Sostituzione per permettere il campionamento ripetuto
                                      n_samples=len(df_majority),  # Eguaglia la classe maggioritaria
                                      random_state=42)  # Per la riproducibilità

    # Unisci i dati rimanenti
    dataset = pd.concat([df_majority, df_minority_upsampled1, df_minority_upsampled2])
    #fau un shuffle delle righe del dataset
    dataset = dataset.sample(frac=1).reset_index(drop=True)
    return dataset
```

La seconda tecnica prevede la rimozione di alcune feature di input che non contribuiscono significativamente alla capacità predittiva del modello. Questo può essere realizzato utilizzando la funzione di importanza delle feature del

modello RandomForestClassifier, eliminando le variabili con un'importanza inferiore a 0,02.



Eliminiamo le feature meno significative:

- 1: Unnamed 01;
- 2 : Newconstly;
- 4: yearConstructed;
- 6: hasKitchen;
- 5: noParkSpaces;
- 15: numFloors.

Adesso analizziamo i risultati dopo aver applicato queste 2 tecniche:

Metriche finali:

```
Decision Tree Classifier
Accuracy:  0.8442188406894289
Precision:  0.8486857855148496
Recall:  0.8458896969914635
F1:  0.8449849423394415
```

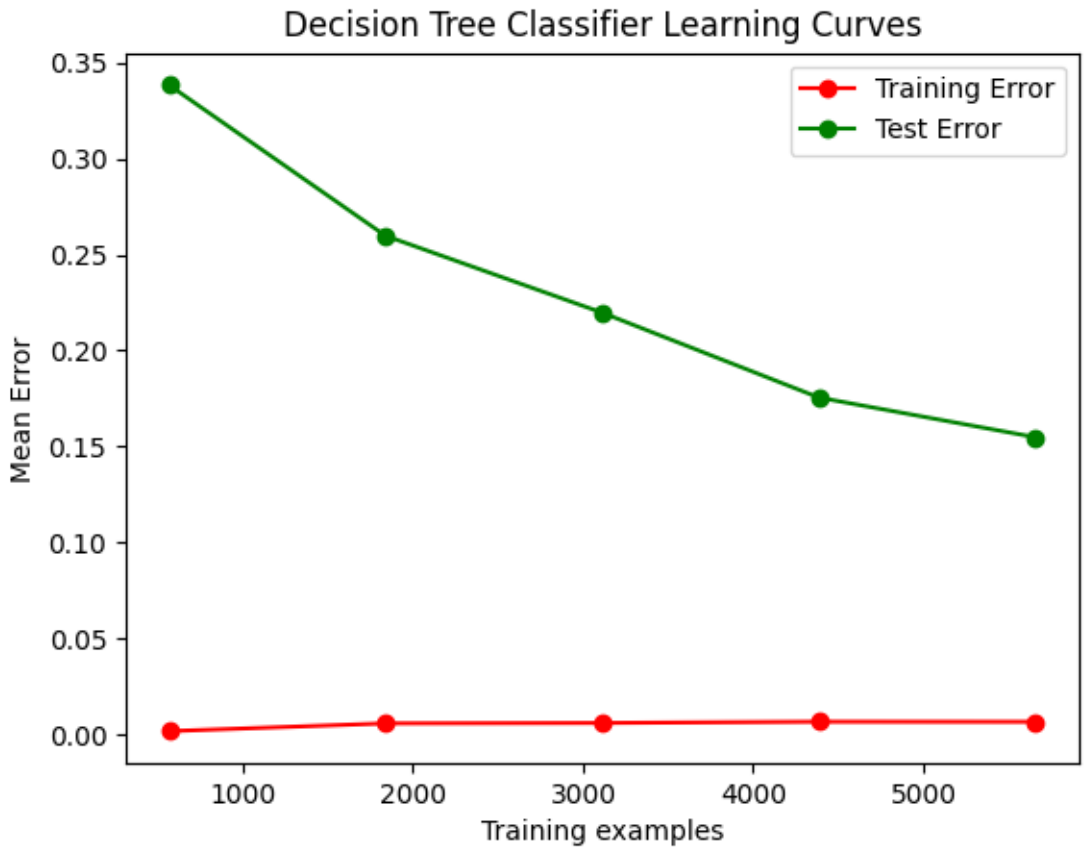
```
Random Forest Classifier
Accuracy:  0.8675872511166629
Precision:  0.8703623125078824
Recall:  0.8677235067946558
F1:  0.8688340529782709
```

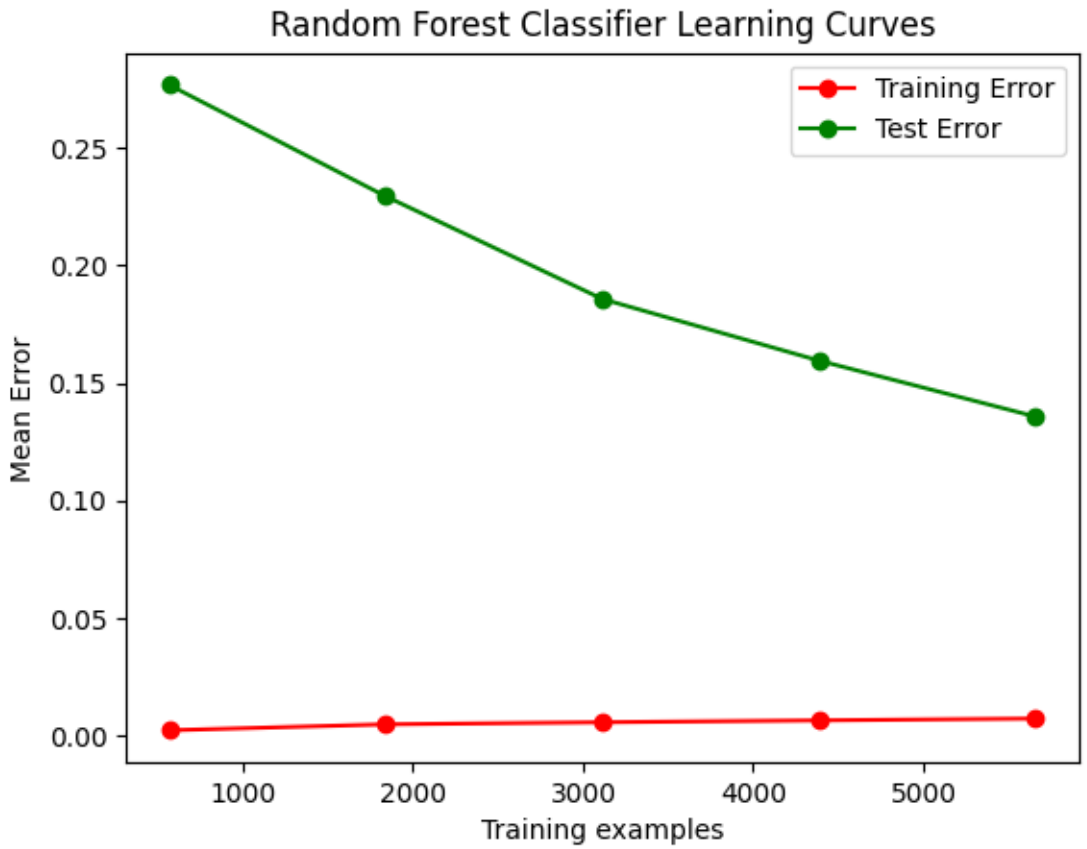
Abbiamo ottenuto un miglioramento di almeno 10% quindi possiamo affermare che il nostro modello ha aumentato le sue capacità predittive.

Analizziamo adesso deviazione standard e varianza.

	DecisonTree	RandomForest
Deviazione Train	2.979070391314114e-07	3.219850804165293e-07
Deviazione Test	0.00022469836826866132	0.00018873704551357494
Varianza Train	0.00039721599584880974	0.00042223529017814 68
Varianza Test	0.014553656743980056	0.013170558842001832

Curve di addestramento





Conclusioni

L'overfitting è diminuito, soprattutto per il modello Random Forest, che mostra una maggiore stabilità tra i dati di addestramento e test. Per il Decision Tree, anche se c'è stata una riduzione rispetto ai valori precedenti, l'overfitting è ancora presente, poiché la differenza tra le prestazioni sui dati di addestramento e test rimane significativa. In sintesi, l'oversampling e la rimozione delle feature non rilevanti hanno avuto un effetto positivo, ma il Random Forest mostra un miglioramento più marcato nella generalizzazione rispetto al Decision Tree.