Caso di studio per Ingegneria della Conoscenza

MODELLO DI PREDIZIONE DEL PREZZO DI UNA MACCHINA

Componenti gruppo: Sara Padovano, matr. 759496

Link GitHub: https://github.com/SaraPadovano/iCon.git

A.A. 2024/25

INDICE

1. INTRODUZIONE

Il mondo delle macchine è un mondo affascinante sempre in continua crescita ed evoluzione. A rendere questi mezzi a quattro ruote tanto affascinanti hanno senza dubbio contribuito le corse automobiliste, tra le più famose sicuramente rally e formula 1, nonché la popolarissima saga di film Fast & Furious che ha plasmato un'ideale di macchina nell'immaginario collettivo.

Non dimentichiamo però che le macchine hanno da sempre affascinato l'uomo grazie alla loro tecnologia e sono il mezzo di trasporto privato più usato al mondo. Infatti non si deve essere degli appassionati senza eguali per poter guidare un automobile.

Tuttavia, nonostante le auto sono direttamente o indirettamente così popolari, quanti effettivamente riescono a riconoscere il valore di una macchina date le sue caratteristiche principali? Ecco questo caso di studio si propone di creare un sistema che date le caratteristiche principali di una macchina quali per esempio l'mpg, i cilindri, il peso, l'accelerazione, etc.

Conoscere effettivamente il prezzo della macchina che si va a comprare date queste caratteristiche fondamentali e a volte non considerate e ai più sconosciute si potrebbe dire, ci consente di poter dare un valore di prestazione alla macchina che si vuole eventualmente comprare cercando magari il giusto rapporto qualità-prezzo.

2. CREAZIONE DEL DATASET

Come primo passo si è pensato alla creazione del dataset. Il dataset che è stato usato nel progetto è stato preso da Kaggle. La ricerca del dataset non è stata facile, questo è stato dovuto dal fatto che per il progetto si è cercato un dataset con tutte le caratteristiche principali, alcune delle quali riportate sopra, e con l'aggiunta del prezzo di valore del mercato attuale della macchina e la casa automobilistica di produzione. Purtroppo, questa combinazione features non erano presenti in alcun dataset esplorato. Si è quindi pensato di prendere come dataset base quello usato nel progetto e di ampliarlo aggiungendo le features mancati di creator e price. Per la feature price si è preso in considerazione il prezzo in euro facendo riferimento al prezzo di mercato attuale, prendendo quindi i dati da Classic.com principalmente che consente di avere una visuale dell'andamento del prezzo della macchina mettendo a disposizione un prezzo medio da prendere in considerazione (prezzo espresso in dollaro e poi convertito). Qualora su Classic.com non fosse stato disponibile il prezzo dell'auto cercata si è usata un'altra fonte quale AutoScout24 che consente di avere anche in questo caso una media del prezzo in euro, altrimenti anche Automoto.it che consente di prendere visione del listino prezzo attuale della macchina in base alle sue caratteristiche e al modello, Hagerty e altri siti molto meno usati in caso di mancanza di dati da questi.
Per rendere il dataset più ricco e completo l'ho poi ampliato aggiungendo altre macchine e recuperando le informazioni delle schede tecniche da alcuni siti come Car and Driver, Edmunds, Motor Trend e AutoWeek e qualche altro sito sporadico.
La data di produzione del modello della macchina inoltre parte dal 1970 e non ha un anno limite, cioè può arrivare tranquillamente fino al 2024.

2.1 FEATURES DI UNA MACCHINA

Le macchine presenti nel dataset vengono descritte attraverso le seguenti caratteristiche fondamentali:

-**name** stringa che rappresenta il nome del modello della macchina compreso di anno di produzione del veicolo per disambiguarlo da eventuali altri modelli che presentano lo stesso nome ma hanno un anno di produzione diverso e che quindi richiamano valori diversi per le altre features;

-**mpg** intero che misura l'efficienza del consumo di carburante di un veicolo. Essendo un acronimo per "miles per gallon" possiamo dire che indica quante miglia un veicolo può percorrere utilizzando un gallone di carburante. Questa unità di misura è molto usata nei paesi anglosassoni e più alto è il valore dell'mpg più efficiente è il consumo di carburante del veicolo;

-**cylinders** intero che si riferisce al numero di cilindri, componenti del motore dove avviene la combustione di carburante e la generazione di energia. È una feature molto importante

perché il numero di cilindri presenti nel motore determina le sue caratteristiche prestazionali come potenza e consumi di carburante. Ovviamente la presenza di più cilindri (dagli 8 in su) identifica principalmente macchine di lusso, sportive o mezzi pesanti che necessitano di un maggior numero di cilindri per una maggiore fluidità ma consumano anche di più. Invece macchine con meno cilindri (dai 6 in giù) soffriranno di una minore fluidità e potenza ma allo stesso tempo ridurranno i consumi e avranno anche un peso più leggero;

-**displacement** intero ossia la cilindrata di un motore non indica altro se non una misura del volume totale di aria e carburante che viene spostato dai pistoni in tutti i cilindri durante un ciclo completo del motore. Esso influisce sulle prestazioni del motore in quanto un motore con maggiore cilindrata generalmente produce più potenza ma consuma anche più carburante.

-**horsepower** intero è l'unità di misura della potenza di un motore. Maggiore è l'horsepower più è potente il motore, tutto ciò porta quindi ad avere una maggiore accelerazione e velocità massima nonché ad un maggior consumo di carburante. Ovviamente un veicolo tanto più è pesante tanto più necessita di un maggior numero di horsepower.

-**weight** intero espresso in libre che identifica il peso del veicolo. Come si può già aver letto dalle features precedenti il peso di un veicolo influisce molto sugli horsepower, cylinders e di conseguenza quindi potenza, velocità e accelerazione del mezzo.

-**acceleration** intero identifica la capacità di un veicolo di aumentare la sua velocità in un determinato periodo di tempo. È stata espressa da 0-60 mph equivalente a 0-100 km/h. Tra i vari fattori che possono influenzare l'accelerazione di un veicolo ovviamente ci sono l'horsepower e il peso.

-**model_year** intero il cui range va da [1970,2021] che identifica l'anno di produzione del veicolo. Si è voluto mantenere questa feature perché il prezzo di mercato e le varie caratteristiche di un motore ovviamente possono essere influenzate dal periodo di produzione dovuto appunto a tecnologie e conoscenze più o meno avanzate.

-**origin** corrisponde al luogo di produzione del veicolo. È una feature categorica che può quindi avere dominio [usa, japan, europe].

-**creator** corrisponde alla casa produttrice del veicolo ed è anch'essa una feature categorica con dominio [chevrolet, buick, plymouth, amc, ford, pontiac, dodge, toyota, datsun, peugeot, audi, saab, bmw, opel, fiat, volkswagen, mercury, oldsmobile, chrysler, mazda, volvo, renault, honda, mercedes, subaru, nissan, porsche, ferrari, mitsubishi, jeep, jaguar, lamborghini]. Questa feature è importante perché case produttrice diverse hanno standard diversi per i propri veicoli che quindi possono influenzare il prezzo del veicolo.

-**price** intero e identifica il prezzo di mercato che il veicolo ha assunto negli ultimi anni. Questa rappresenta la nostra **target feature**.

2.2 PRE-PROCESSING DEI DATI

La fase di pre-processing dei dati riguarda la semplice rimozione della feature **origin** poiché la conoscenza del luogo di produzione del veicolo non porta conoscenza e non pesa sul prezzo del veicolo stesso.
Essendo quindi una feature inutile allo scopo finale del progetto, origin è stata eliminata creando così il file *Automobile_cleaned.csv*.
L'elenco quindi delle feature aggiornato presente nel dataset è il seguente:
-**name**;
-**mpg**;
-**cylinders**;
-**displacement**;
-**horsepower**;
-**weight**;
-**acceleration**;
-**model_year**;
-**creator**;
-**price**.
All'interno del dataset non sono presenti dati mancanti ma comunque è stata fatta una pulizia del file per sicurezza.

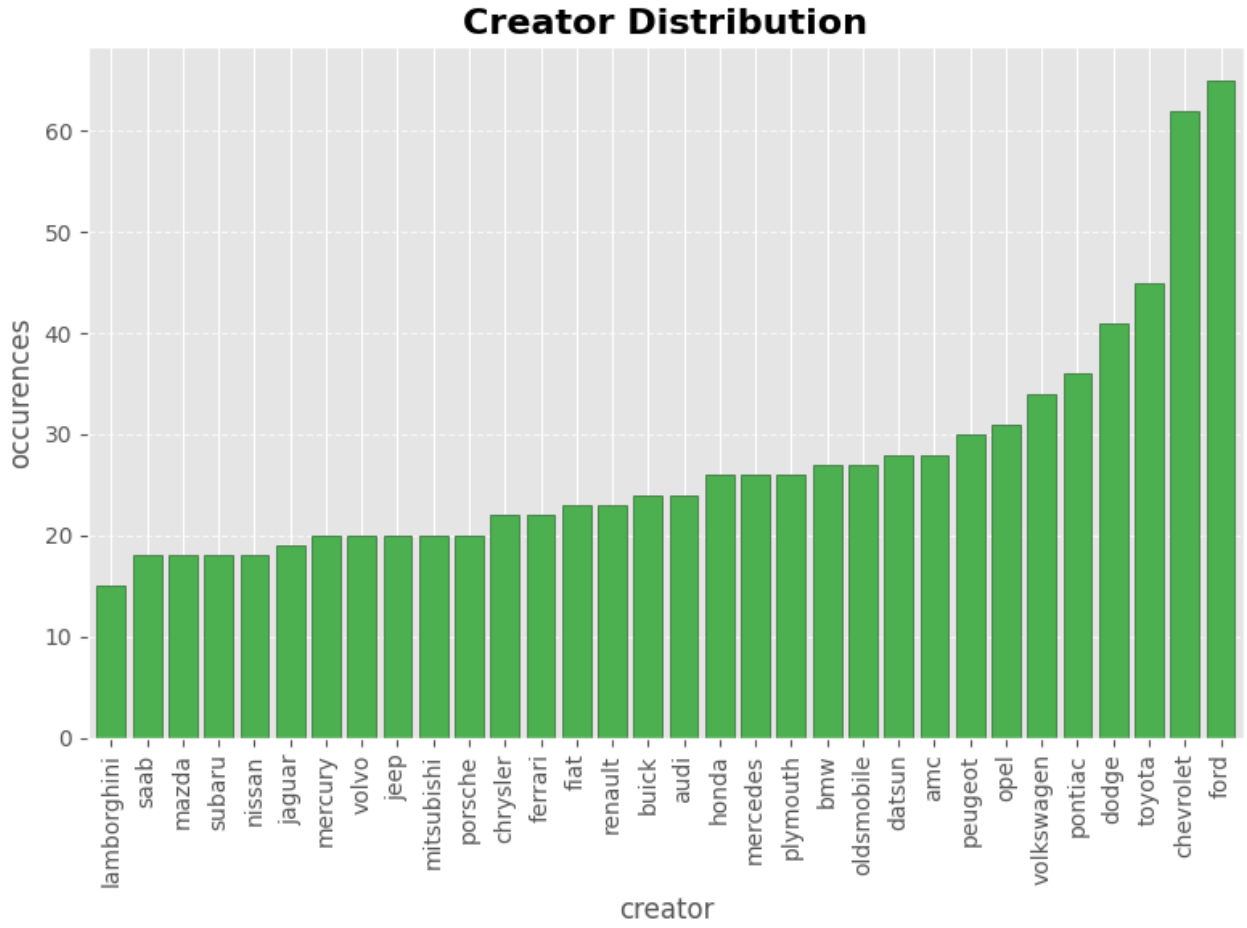
2.3 FEATURE ENGINEERING DEI DATI

Per preparare i dati ai vari task di apprendimento che verranno descritti in seguito alle colonne già presenti (descritte e modificate nei paragrafi precedenti) sono aggiunte altre colonne che riporto di seguito:

- La feature **creator** è stata suddivisa creando una colonna per ogni casa produttrice applicando quindi la tecnica *one hot encoding* che consente di trasformare feature categoriche in feature booleane attraverso l'uso di variabili indicatrici.
- In aggiunta a **model_year** si è deciso di aggiungere le colonne **recent**, identificata come una feature booleana, per indicare se una macchina è stata realizzata prima o dopo il 2015 e quindi magari essere più veloci nel capirne il prezzo collegato.

Tutte le altre features sono poi state normalizzate per ridurre la loro cardinalità e quindi, come già detto sopra, per renderle pronte ai vari task di apprendimento. In particolare si è principalmente usato con le feature intere con il *MinMaxScaler* per avere valori compresi tra 0 e 1 (considerando che poi i generale tutti i dati non hanno una grande variazione ossia rientrano in uno stesso range e non si discostano, al massimo se si discostano si discostano di pochissimo). Invece per la target feature è stato usato come normalizzazione il log1 (non i

log normale per evitare eventuali problemi in caso di valori nulli) per rendere i dati più gestibili ed evitare eventuali outliers.



Come si può vedere dal grafico dei creator, non vi è uniformità tra le varie tipologie di macchine prodotte dalle varie case produttrici. Questo distacco si può notare particolarmente tra le prime case (volvo, mercury,..) e le ultime case riportate (chevrolet, ford). Il dataset quindi è sbilanciato per quanto riguarda il numero di macchine presenti per case automobilistiche.

3. RAGIONAMENTO LOGICO E PROLOG

Il **ragionamento logico** è un ragionamento che avviene sfruttando la logica matematica con cui si istanziano delle regole logiche che consentono di dedurre nuovi fatti da quelli già esistenti. Si va quindi così a creare una *knowledge base* che contiene vari assiomi (un' assioma è un'informazione che viene sempre considerata vera e che non ha bisogno di dimostrazioni per affermare la sua veridicità. Nel nostro caso abbiamo definito una knowledge base contenente le varie informazioni importanti per quanto riguarda le automobili. Il tutto si può ritrovare in *src/kb.pl* dove è stata quindi creata la nostra knowledge base in cui sono descritti i fatti e le regole della nostra kb
Quindi per prima cosa si sono definiti i vari creator (le case produttrici) possibili per ciascuna istanza dell'automobile. Sono quindi stati scritti nella nostra knowledge base utilizzando come linguaggio **Prolog** tramite la libreria *pyswip* per integrarlo con Python. La scrittura dei creator nel file è quindi portato a questo risultato:

```
creator('chevrolet').
creator('buick').
creator('plymouth').
creator('amc').
creator('ford').
creator('pontiac').
creator('dodge').
creator('toyota').
creator('datsun').
creator('peugeot').
creator('audi').
creator('saab').
creator('bmw').
creator('opel').
creator('fiat').
creator('volkswagen').
creator('mercury').
creator('oldsmobile').
creator('chrysler').
creator('mazda').
creator('volvo').
creator('renault').
creator('honda').
creator('mercedes').
creator('subaru').
creator('nissan').
creator('porsche').
creator('ferrari').
creator('mitsubishi').
creator('jeep').
creator('jaguar').
creator('lamborghini').
```

Si è poi proceduto all'aggiunta dei fatti che riguardano le automobili. Si riportano di seguito alcune istanze presenti nella kb:

```
auto('chevrolet chevelle malibu 1970',18,8,307,130,3504,12,1970,'chevrolet',23000).
auto('buick skylark 320 1970',15,8,350,165,3693,11,1970,'buick',13000).
auto('plymouth satellite 1970',18,8,318,150,3436,11,1970,'plymouth',16000).
auto('amc rebel sst 1970',16,8,304,150,3433,12,1970,'amc',22400).
auto('ford torino 1970',17,8,302,140,3449,10,1970,'ford',45000).
auto('ford galaxie 500 1970',15,8,429,198,4341,10,1970,'ford',13000).
auto('chevrolet impala 1970',14,8,454,220,4354,9,1970,'chevrolet',20000).
auto('plymouth fury iii 1970',14,8,440,215,4312,8,1970,'plymouth',20000).
auto('pontiac catalina 1970',14,8,455,225,4425,10,1970,'pontiac',9500).
auto('amc ambassador dpl 1970',15,8,390,190,3850,8,1970,'amc',10000).
auto('dodge challenger se 1970',15,8,383,170,3563,10,1970,'dodge',52000).
auto('plymouth cuda 340 1970',14,8,340,160,3609,8,1970,'plymouth',40000).
auto('chevrolet monte carlo 1970',15,8,400,150,3761,9,1970,'chevrolet',30000).
auto('buick estate wagon 1970',14,8,455,225,3086,10,1970,'buick',13000).
auto('toyota corona mark ii 1970',24,4,113,95,2372,15,1970,'toyota',14000).
auto('plymouth duster 1970',22,6,198,95,2833,15,1970,'plymouth',10250).
auto('amc hornet 1970',18,6,199,97,2774,15,1970,'amc',11000).
auto('ford maverick 1970',21,6,200,85,2587,16,1970,'ford',9000).
auto('datsun pl510 1970',27,4,97,88,2130,14,1970,'datsun',21000).
auto('peugeot 504 1970',25,4,110,87,2672,17,1970,'peugeot',10000).
auto('audi 100 ls 1970',24,4,107,90,2430,14,1970,'audi',15000).
auto('saab 99e 1970',25,4,104,95,2375,17,1970,'saab',8000).
auto('bmw 2002 1970',26,4,121,113,2234,12,1970,'bmw',24000).
auto('amc gremlin 1970',21,6,199,90,2648,15,1970,'amc',15300).
auto('ford f250 1970',10,8,360,215,4615,14,1970,'ford',17500).
auto('chevrolet c20 1970',10,8,307,200,4376,15,1970,'chevrolet',17000).
auto('dodge d200 1970',11,8,318,210,4382,13,1970,'dodge',16000).
auto('chevrolet vega 2300 1971',28,4,140,90,2264,15,1971,'chevrolet',6470).
auto('toyota corona 1971',25,4,113,95,2228,14,1971,'toyota',10000).
auto('plymouth satellite custom 1971',16,6,225,105,3439,15,1971,'plymouth',5850).
auto('amc matador 1971',18,6,232,100,3288,15,1971,'amc',12500).
```

Per la definizione delle regole si è identificata una per quanto riguarda il controllo del modello dell'anno anche perché tra le varie componenti mpg, cylinders, horsepower, displacement, price, model_year, accelearation non vi è una vera e propria correlazione che consente di definire una regola precisa neanche tra macchine appartenenti alla stessa casa produttrice poiché la stessa casa può dedicarsi alla produzione di macchine diverse (dalle sportive, alle più comuni, a quelle a uso familiare). Certamente macchine come porsche, lamborghini, ferrari sono più dedite ad un tipo di macchina prettamente sportiva (tolti alcuni modelli) ma comunque ciò non consente di definire una regola precisa tra i vari componenti sopra citati.
La scrittura dei fatti in Prolog per popolare la nostra kb si ferma qui.
Per scrivere i nostri fatti nella kb il codice è stato scritto nello script *src/auto_prolog.py* in cui sono definite principalmente le due funzioni per la scrittura dei fatti dei creator e delle auto e dove inoltre per quest'ultima parte è pure presente un controllo per verificare che i fatti nuovi che vengono scritti non siano poi duplicati nella kb.
Queste funzioni poi vengono richiamate nel *src/main.py:*

```
# RAGIONAMENTO LOGICO
write_creators(fileName_cleaned)
write_auto_info(fileName_cleaned, file_know_base)
```

Visto che al momento non sono state applicate alcun tipo di regole alla kb non è stato necessario creare un nuovo file csv contenente i nuovi dati. Anche perché dopo la pulizia dei dati presenti in */dataset/Automobile_cleaned.csv* dopo le varie pulizie (rimozione dei dati mancanti, controllo delle colonne numeriche,...-> il tutto presente nel main) e dopo anche l'applicazione delle funzioni per il feature engineering che ha portato alla creazione del file */dataset/Automobile_features.csv* (in cui appunto sono presenti le nuove colonne e la normalizzazione dei dati numerici) sembra superflua la creazione di un altro file csv per ribadire qualcosa già presente e ampiamente controllata nel file Automobile_cleaned.csv.

4. APPRENDIMENTO NON SUPERVISIONATO

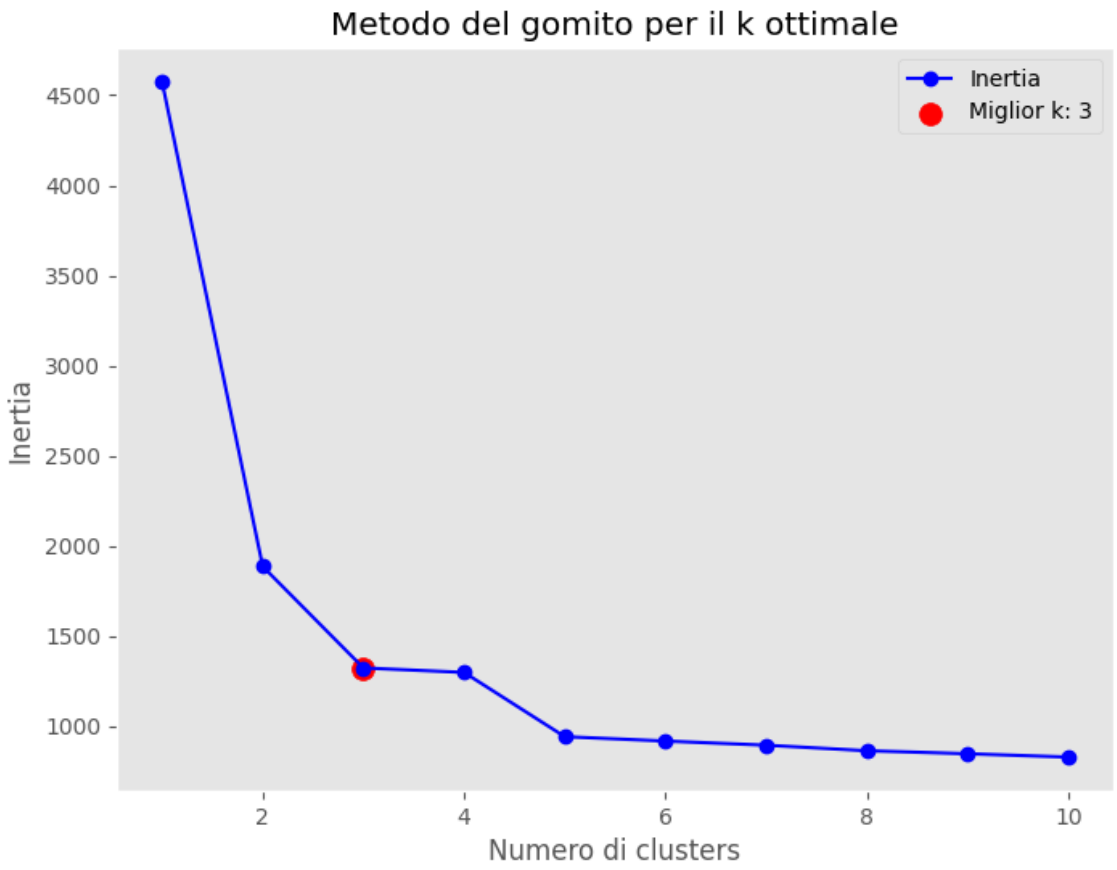
L'apprendimento non supervisionato fa parte dell'apprendimento automatico ed in particolare si riferisce al caso in cui un modello venga addestrato senza informazioni sulla feature target di riferimento. Tra i principali tipi di apprendimento non supervisionato abbiamo il clustering (raggruppare elementi date delle somiglianze) e la riduzione della dimensionalità (ridurre il numero di feature mantenendo le informazioni più importanti). Noi ci focalizzeremo sul **clustering**, cercheremo quindi di raggruppare le macchine con feature simili scoprendo qualche pattern magari nascosto, "indiretto" e che per esempio nella sezione precedente non ci aveva consentito di trovare delle regole precise da applicare alla nostra kb ma che adesso, grazie al clustering, potrà venire a galla.
Anche in questo caso il clustering si divide in **hard clustering** (si associa un esempio ad un cluster specifico) e **soft clustering** (si associa un esempio la probabilità di appartenere ad ogni cluster). Noi ci concentreremo sull'hard clustering utilizzando l'algoritmo **KMeans** (dato un numero di cluster divide gli esempi tra di essi cercando di modificare la distribuzione cercando di minimizzare la variabilità nei vari cluster) affiancato da una strategia chiamata **metodo del gomito** per individuare il numero di cluster necessari all'algoritmo.

4.1 REGOLA DEL GOMITO

Per determinare il numero di cluster necessari all'algoritmo KMeans abbiamo applicato una strategia chiamata **regola del gomito**. La regola del gomito si basa sulla variazione dell'*inertia* al variare del numero di cluster (variazione poi mostrata in un grafico) dove l'inertia non è altro che la somma dei quadrati delle distanze tra i punti e i centroidi del cluster, rispetto al numero di cluster k. L'algoritmo lavora eseguendo il KMeans per diversi valori di numero di cluster, per ognuno di essi calcola l'inertia e plotta la curva che serve a identificare il gomito ovvero il punto in cui l'inertia diventa meno significativa.
Questo è il codice dell'algoritmo (presente nello script *scr/unsupervised_learning*):

```python
# Funzione che calcola il numero di cluster ottimale per il dataset mediante la regola del gomito
1 usage  SaraPadovano
def regola_gomito(dataset, file):
    inertia = []
    # Fisso un range di k da 1 a 10
    k_range = range(1,11)
    for k in k_range:
        #random restart
        kmeans = KMeans(n_clusters=k, n_init=5, init='random')
        kmeans.fit(dataset)
        inertia.append(kmeans.inertia_)
    kl = KneeLocator(k_range, inertia, curve="convex", direction="decreasing")
    plt.figure(figsize=(8, 6))
    plt.plot( *args: k_range, inertia, marker='o', linestyle='-', color='b', label='Inertia')
    plt.scatter(kl.elbow, inertia[kl.elbow - 1], c='red', s=100, label=f'Miglior k: {kl.elbow}')
    plt.xlabel('Numero di clusters')
    plt.ylabel('Inertia')
    plt.title('Metodo del gomito per il k ottimale')
    plt.legend()
    plt.grid(False)
    plt.savefig(file)
    plt.close()
    return kl.elbow
```

Come si può vedere si è scelto di usare un k, numero di cluster, uguale a 10 (identificato dal range k_range). Si esegue poi l'algoritmo KMeans per ogni k impostando come numero delle *random restart* 5 (*n_init=5*) (all'inizio avevo scelto come valore 10 poi ho deciso di provare con 5 per vedere se c'erano variazioni e se si poteva ottenere lo stesso risultato facendo meno random restart, visto che non c'erano variazioni ho deciso di lasciare 5). Per ogni k quindi l'algoritmo verrà quindi eseguito 5 volte e restituirà poi il clustering migliore. E infine abbiamo l'inizializzazione casuale dei centroidi con *init='random'*.
Una volta fatto ciò si usa *KneeLocator* per individuare la posizione del gomito nella curva dell'inertia.
Alla fine si è ottenuto questo grafico dove si è visto che il giusto numero di cluster è 3:

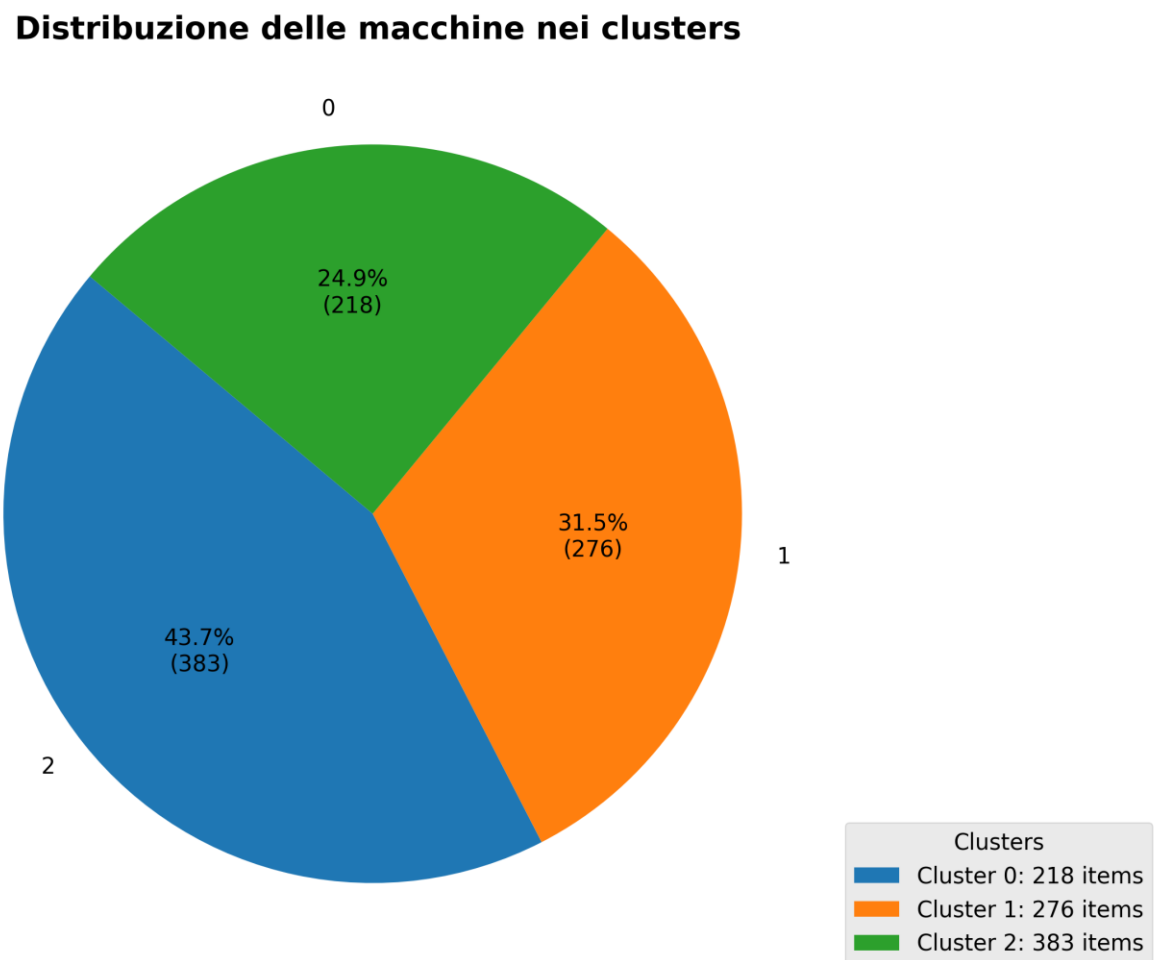


4.2 HARD CLUSTERING

Adesso è possibile eseguire l'hard clustering usando l'algoritmo KMeans. Per usare l'algoritmo sono state considerate le feature più rilevanti:

```python
# APPRENDIMENTO NON SUPERVISIONATO
df = pd.read_csv(fileName_features, encoding='utf-8-sig')
features = [
    'recent', 'normalized_mpg', 'cylinders', 'normalized_displacement', 'normalized_horsepower',
    'normalized_weight', 'normalized_acceleration', 'chevrolet', 'buick', 'plymouth', 'amc', 'ford', 'pontiac'
    'dodge', 'toyota', 'datsun', 'peugeot', 'audi', 'saab', 'bmw', 'opel', 'fiat', 'volkswagen', 'mercury',
    'oldsmobile', 'chrysler', 'mazda', 'volvo', 'renault', 'honda', 'mercedes', 'subaru', 'nissan', 'porsche',
    'ferrari', 'mitsubishi', 'jeep', 'jaguar', 'lamborghini'
]
clusters, centroids = cluster(df, features, filek: '../png/best_k', filec: '../png/distribution_cars_in_clusters',
```

Non c'è stato bisogno di eliminare le features non numeriche perché abbiamo considerato il file fileName_features ottenuta dal feature engineering dei dati dove sono presenti solo feature numeriche. Di queste feature, come si può vedere dall'immagine, sono poi state scelte solo le feature più importanti che hanno una stretta correlazione con poi il prezzo della macchina. Si è quindi poi applicato l'algoritmo KMeans ottenendo il seguente grafico a torta:



Come si può vedere dal grafico la distribuzione delle macchine nei vari cluster non è bilanciata (conoscendo il mio dataset me lo aspettavo). Alla fine non sono completamente sbilanciati ma comunque la cosa può influire sugli altri task di apprendimento supervisionato quindi si applicheranno delle strategie per cercare di risolvere lo sbilanciamento.
Inoltre dal clustering è stato poi creato un altro file *dataset/Automobile_clusters.csv* dove è presente una nuova feature cluster che, per ogni auto, mi identifica il relativo cluster di appartenenza.

5. APPRENDIMENTO SUPERVISIONATO

Nell'apprendimento supervisionato un modello viene addestrato su un set di dati etichettati. I task che si possono portare avanti sono di due tipi:

- **Classificazione**: l'etichetta ha un dominio discreto;
- **Regressione**: l'etichetta ha un dominio continuo.

Il nostro caso, ossia quello di predire il prezzo di una macchina, è un task di regressione proprio perché la nostra target feature price è continua.
Prima di dire i modelli che ho deciso di usare per questa parte del progetto, si elencano di seguito le fasi che dovremo attraversare per poter addestrare e trovare il modello più adatto:

1) Scegliere gli iper-parametri (per ogni modello);
2) Addestrare il modello in base agli iper-parametri scelti;
3) Testing;
4) Valutazione dei modelli.

5.1 MODELLI

I modelli considerati per il task di apprendimento supervisionato sono i seguenti:

- **DecisionTree** (alberi di decisione) -> il classificatore si presenta con una struttura ad albero dove la radice e i nodi interni rappresentano delle condizioni/degli stati delle features di input che, in base alla loro condizione, portano ai nodi foglia che rappresentano il valore continuo da predire (differenza con i decision tree per classificazione in cui invece nei nodi foglia vi è la classe di appartenenza).
- **RandomForest** -> estende i decision trees creando appunto una foresta di questi alberi indipendenti. La predizione finale si ottiene mediando le decisioni finali dei singoli alberi.
- **LightGBM** -> Modello basato su boosting, ottimizzato per scalabilità e gestione di dati con feature categoriche. Ho scelto questo modello perché rispetto agli altri risulta avere una maggiore scalabilità e velocità di apprendimento. In principio si è pure provato a usare *Catboost*, un altro modello basato su boosting come LightGBM. Non si è usato alla fine perché le librerie usate da CatBoost andavano in contrasto con alcune librerie interne come matplotlib e numpy.

5.2 FASE 1) SCELTA DEGLI IPERPARAMETRI

Gli **iperparametri** di un modello di apprendimento sono dei parametri che devono essere stabiliti prima della fase di addestramento del modello poiché la loro scelta influirà sulle prestazioni del modello stesso e sulla sua complessità. La loro scelta è quindi cruciale (anche perché oltre ai problemi qui citati possono portare ad altri problemi quali sovradattamento e sottodattamento del modello).

1. DecisionTree:
    a. **criterion** (misura la qualità di uno split):
        - **squared_error(mse)**: calcola la somma dei quadrati degli errori residui (mean squared error) per valutare la purezza del nodo. Solitamente è una scelta più comune e usata per i task di regressione in quanto minimizza l'errore quadratico medio;
        - **absolute_error**: usa l'errore mediano per suddividere i dati. Utile se i dati contengono outlier;
        - **poisson**: usa la devianza di Poisson come misura;
        - **friedman_mse**: variante ottimizzata dello squared_error poiché tiene conto sia della riduzione dell'errore che della struttura dell'albero.
    b. **max_depht** (profondità massima dell'albero):
        - valori presi in considerazione sono [None, 5, 10] visto che si ha un dataset piccolo.
    c. **splitter** (strategia da usare per il criterio di split):
        - ho usato la strategia di default **best** che indica il miglior criterio di split possibile senza doverlo ricercare.
    d. **min_samples_split** (specifica il numero minimo di campioni richiesti in un nodo per poter eseguire uno split):
        - valori usati visto il dataset piccolo [2, 5, 10, 20].
    e. **min_samples_leaf** (numero minimo di campioni richiesti in un nodo foglia):
        - valori usati [1, 2, 5, 10, 20] per il dataset piccolo.
    f. **random_state** (per controllare la casualità dello stimatore ed evitare di avere ad ogni run valori diversi):
        - valore scelto è 42.
2. RandomForest (si ribadisce come sopra criterion, max_depht e random_state):
    a. in aggiunta come altro iperparametro abbiamo **n_estimators** (che si riferisce al numero di alberi decisionali che verranno costruiti nel modello):
        - valori usati [10, 20, 50, 100]. Oltre il 100 non voglio andare sia perché abbiamo un dataset piccolo (100 infatti è da vedere) sia perché si rischia di avere troppo tempo di computazione.
3. LightGBM :
    a. **n_estimators** come in randomForest;
    b. **random_state** come nei due modelli precedenti;
    c. **learning_rate** (velocità con cui il modello apprende i dati durante l'addestramento):
        - valori usati per un piccolo dataset [0.01, 0.05, 0.1].
    d. **max_depht** (profondità massima degli alberi): come sopra;
    e. **class_weight** (il peso dato a ciascuna classe):
        - valore scelto ["balanced"] per calcolare automaticamente il peso delle classi in modo inversamente proporzionale allo loro frequenza.

f. **verbose** (consente di evitare di avere in output le notifiche del sistema. All'inizio non era stata considerata perché il suo valore di default è 0. Vuol dire che quindi ignora le notifiche più superficiali e dà in output solo quelle importanti. Si è deciso di inserire modificando il valore quando in run il programma ha dato in stampa troppe informazioni non utili ai fini del progetto poiché informazioni di lavoro di base di LightGBM):
    - valore usato: [-1]

5.3 RICERCA DEI MIGLIORI IPERPARAMETRI

Come abbiamo già detto all'inizio del paragrafo precedente, la scelta degli iperparametri è una scelta molto importante. Per poter fare in modo di scegliere i migliori iperparametri si è applicata una tecnica chiamata **k-fold cross validation**. Questa tecnica divide il dataset in k insiemi disgiunti (fold) in cui un fold viene usato per il test e i restanti k-1 per l'addestramento. L'azione verrà ripetuta per k volte prendendo fold diversi per il test e l'addestramento ogni volta per addestrare al meglio il modello. La strategia scelta per la ricerca degli iperparametri è la **GridSearch** con cross validation in cui si definiscono le griglie dei valori possibili per gli iperparametri e si esplorano tutte le combinazioni possibili alla ricerca della migliore. Il k scelto per addestrare il modello è stato 5 e si è impostata la GridSearch facendo in modo che gli iperparametri trovati minimizzino la metrica *neg_mean_squared_error*(errore quadratico medio negativo). È stata fatta questa scelta per questioni legate alla libreria di scikit-learn che di per sé cerca sempre di massimizzare la metrica scelta. Ecco perché non si è potuto scegliere l'errore quadratico medio positivo perché è proprio quello che si vuole minimizzare (si massimizzerebbe l'errore invece di ridurlo). Si è poi impostato n_jobs a -1 per cercare di usare tutti i core disponibili durante la GridSearch e infine error_score='raise' per fare in modo che se si verifica un errore durante la ricerca dei parametri si solleva immediatamente un'eccezione che interrompe il processo (molto utile per il debug).

```
gridSearchCV_lgbmr = GridSearchCV(Pipeline([('LightGBM', lgbmr)]), LGBMHyperparameters, cv=5, n_jobs=-1,
                                  scoring='neg_mean_squared_error', error_score='raise')
```

Alla fine i migliori iperparametri scelti per ogni modello sono (si può trovare in *text/hyperparameters.txt*):

```
Best DecisionTree parameters found:
DecisionTree__criterion            squared_error
DecisionTree__max_depth            10
DecisionTree__min_samples_leaf     20
DecisionTree__min_samples_split    2
DecisionTree__random_state         42
DecisionTree__splitter             best
Best RandomForest parameters found:
RandomForest__criterion            absolute_error
RandomForest__max_depth            10
RandomForest__n_estimators         100
RandomForest__random_state         42
Best LightGBM parameters found:
LightGBM__class_weight             balanced
LightGBM__learning_rate            0.05
LightGBM__max_depth                5
LightGBM__n_estimators             100
LightGBM__random_state             42
LightGBM__verbose                  -1
```

(Nonostante è stato impostato il seed di random_state per evitare di avere iperparametri sempre diversi ad ogni run, il sistema può comunque variare alcuni iperparametri dovuto a GridSearchCV).

5.4 FASE 2-3) ADDESTRAMENTO E TEST

Per le fasi di test e addestramento i modelli sono stati addestrati usando la *repeated k-fold cross validation* che non è altro che la tecnica del k-fold cross validation ripetuta n volte. N_repeat è stato impostato a 3 visto il dataset piccolo ed è stato impostato random_state = 42 per fare in modo che la divisione in fold possa essere uguale per ogni run del progetto. In generale il progetto è abbastanza veloce solo durante le operazioni in cui è coinvolta la RandomForest c'è un rallentamento (probabilmente dovuto anche al computer). Sono quindi presenti dei print all'interno del codice per monitorare dove si trova il sistema e per avere degli output durante l'attesa delle computazioni.

5.5 FASE 4) VALUTAZIONE DEI MODELLI

Come già scritto sopra per la valutazione si è usata la repeated k-fold cross validation con k=5 e n=3. Le metriche usate per la valutazione dei modelli sono le seguenti:

- **MSE (MEAN SQUARED ERROR)**: calcola l'errore quadratico medio ossia calcola la differenza tra i valori previsti e i valori ottenuti da un modello di regressione.

Rappresenta il valore medio del quadrato degli errori, dove l'errore è la differenza tra il valore osservato e quello previsto. (Valori più bassi indicano una migliore accuratezza).

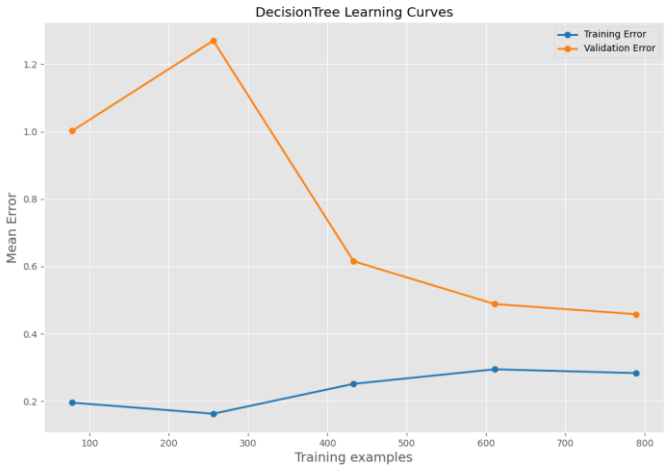
- **MAE (MEAN ABSOLUTE ERROR)**: misura la precisione di un modello calcolando la media delle differenze assolute tra i valori previsti e quelli osservati. È meno sensibile agli outlier rispetto al mse. (Valori più bassi indicano una migliore accuratezza).
- **MSLE (MEAN SQUARED LOGARITHMIC ERROR)**: misura la differenza tra i valori previsti e quelli effettivi in base ai loro logaritmi naturali. Calcola la media del quadrato delle differenze tra il logaritmo dei valori previsti e il logaritmo dei valori reali. Questa metrica è particolarmente utile quando si vuole dare maggiore enfasi agli errori relativi per valori più piccoli e ridurre l'impatto di grandi discrepanze sui valori maggiori. (Valori più bassi indicano una migliore accuratezza).
- $\boldsymbol{R^2}$ misura la proporzione di varianza nella variabile dipendente (la feature target) spiegata dalle variabili indipendenti del modello. Indica quindi quanto bene il modello riesce a rappresentare i dati osservati. (Valore maggiore indica che la metrica riesce a spiegare la varianza dei dati fino ad un massimo valore di 1.0. Può raggiungere anche valori negativi).

5.6 RISULTATI OTTENUTI

- Per DecisionTree abbiamo i seguenti risultati delle metriche (si può trovare in *text/metrics.txt*):

METRIC	SCORE MEAN	SCORE VARIANCE	SCORE STD
MSE	0.39864491008375214	0.0021629382197854226	0.046507399623989115
MAE	0.48981343029465757	0.0007994856414712737	0.028275177125374012
MSLE	0.00363086105647814	1.706581747238763e-07	0.000413107945607290
R2	0.6230707079638885	0.0029462401944306995	0.05427927960493488

La curva di apprendimento invece del DecisionTree è la seguente (l'immagine si può trovare in *png/decisionTree_curve.png* e i dati della costruzione della curva in *text/log_DecisionTree.txt*):

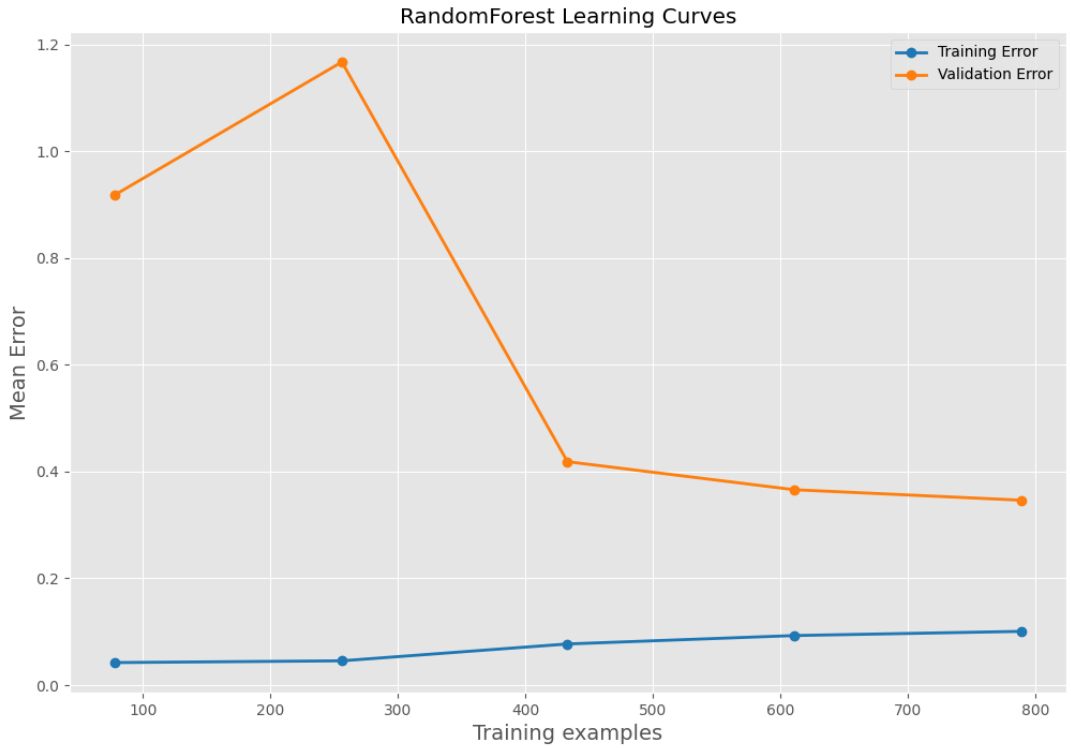


Analizzando il grafico del modello DecisionTree possiamo notare come all'aumentare dei training examples il validation error diminuisca ma allo stesso tempo notiamo come il training error aumenti. Questo è un buon segno perché indica che man mano che si stanno aggiungendo dati il sistema sta diventando più generalizzante (il divario che continua ad esserci tra training error e validation error suggerisce un leggero overfitting) . Anche guardando i risultati delle metriche ci fanno pensare che al momento il modello stia lavorando bene perché mse, mae e msle sono abbastanza bassi. R2 ha un buon valore ma non dei migliori perché vuol dire che non riesce a cogliere abbastanza la variazione dei dati (solo il 62%). Questo potrebbe essere dovuto anche al fatto che si sta lavorando con un dataset piccolo.

- Per RandomForest i risultati delle metriche sono:

METRIC	SCORE MEAN	SCORE VARIANCE	SCORE STD
MSE	0.2799811428879132	0.001998496872211747	0.04470455091164374
MAE	0.4057926941880703	0.000886385355044797	0.02977222455653586
MSLE	0.0026142732063535	1.75838499837177e-07	0.00041933101463781
R2	0.7366238459831249	0.001540663179348575	0.03925128251851875

E la curva di apprendimento ottenuta è la seguente (l'immagine si può trovare in *png/randomForest_curve.png* e i dati della costruzione della curva in *text/log_RandomForest.txt*):

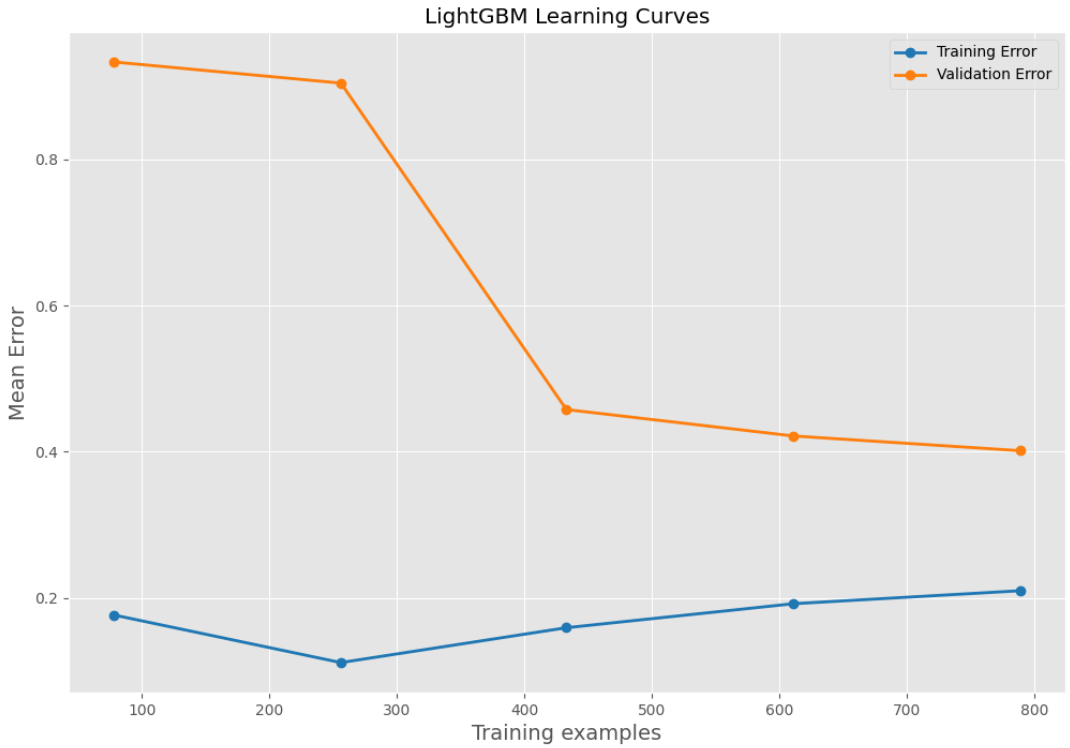


Analizzando il grafico possiamo notare come non sembra esserci il problema dell'overfitting e anzi, osservando i risultati delle metriche, possiamo notare come nel caso precedente delle buone performance che superano quelle del decision tree come possiamo vedere tranquillamente dal valore di R2 dove siamo passati da un 62% di spiegazione dei dati a un buon 73%. Si può però osservare come ad un certo punto del nostro grafico il validation error sembri aver quasi raggiunto un punto stabile e non tenda più di tanto a scendere, questo può indicare come il modello possa aver raggiunto la saturazione e abbia appreso tutto quello che poteva dai dati.

- Per quanto riguarda LightGBM, per le metriche abbiamo:

METRIC	SCORE MEAN	SCORE VARIANCE	SCORE STD
MSE	0.334255009454137	0.0011978406528316853	0.03460983462589333
MAE	0.447505589691525	0.0005180332850578	0.02276034457247517
MSLE	0.003159410582246	1.277655970973354e-07	0.00035744313827143
R2	0.685165314511349	0.0012217923319014358	0.03495414613320479

Invece la curva di apprendimento è (immagine si può trovare in *png/lightGBM.png* e i dati della curva in *text/log_LightGBM.txt*):



Analizzando le metriche non notiamo grande differenza rispetto gli altri due modelli precedenti, infatti in generale i valori si mantengono tra quelli del decisiontree e quelli del randomforest, indicando quindi un possibile miglioramento nei dati. Osservando poi la curva di apprendimento qua sopra possiamo notare come ad una certa il validation error tenda a scendere più lentamente, probabilmente sintomo del fatto che il modello ha appreso dai dati la maggior parte di quello che poteva apprendere e che quindi il modello fatica a migliorare e un aumento invece ad una certa del training error. In generale comunque non sembrano esserci sintomi preoccupanti di overfitting dei dati.

In generale possiamo dire che in base alle analisi appena fatte il modello che sembra avere il comportamento migliore è proprio quello RandomForest, in cui si combina una curva che riesce ben a generalizzare i dati e delle metriche che tra i tre modelli hanno i valori migliori con R2 che riesce a spiegare più del 70% della variazione dei dati. Alla fine però tutti i modelli hanno un buon comportamento e non sembrano presentare sintomi di overfitting dei dati (probabilmente grazie al fatto che si sta lavorando su un piccolo dataset).

6. OVERSAMPLING

Come abbiamo già visto nel secondo capitolo di questa documentazione, le classi presentate nel nostro dataset presentano degli sbilanciamenti. Questi sbilanciamenti possono portare anche a problemi come l'overfitting (problema che per fortuna non abbiamo riscontrato grazie alle piccole dimensioni del dataset). Se quindi si deciderà di ampliare in futuro il nostro dataset il problema dell'overfitting ha alte probabilità di presentarsi. Nonostante questo un "problema" che vogliamo già cercare di risolvere è sicuramente quello di migliore le prestazioni dei nostri modelli (come si è potuto tranquillamente vedere dai valori delle metriche in cui R2 potrebbe raggiungere un livello più alto e le altre metriche potrebbero avere valori anche più bassi).
Per poter fare ciò si è deciso di applicare delle tecniche di **oversampling** dove l'oversampling non è altro che un tecnica usata per cercare di bilanciare al meglio le classi date del dataset (dare quindi una migliore e maggiore visibilità alle classi meno rappresentate).

6.1 OVERSAMPLING NELLA REGRESSIONE

Per effettuare l'oversampling si era pensato di utilizzare una tecnica chiamata adasyn dove si generano campioni sintetici per la classe minoritaria. Il suo vantaggio è che focalizzandosi sulle aree più difficili aiutando il modello sui dati sbilanciati. Nonostante Adasyn (con particolari accorgimenti) possa anche essere usata nei problemi di regressione, purtroppo le tecniche di oversampling in problemi di regressione con dataset sbilanciati è un campo poco esplorato soprattutto nel campo delle implementazioni disponibili (sono infatti state trovate altre tecniche come Smoter, KNNOR-Reg che non sono altro che modifiche di smote per rendere queste tecniche applicabili ai problemi di regressione). Purtroppo queste sono tecniche più complesse e non facilmente implementabili, per questo si è scelto di provare ad utilizzare Adasyn con un problema di regressione.
Purtroppo, come abbiamo già detto, Adasyn è usato principalmente per problemi di classificazione quindi lavora con valori discreti. Per questo il nostro tentativo di usare Adasyn ha avuto un riscontro negativo visto che la target feature dev'essere discreta. Ho voluto comunque portare avanti l'oversampling con adasyn per vedere il comportamento delle curve d'apprendimento dei modelli se da regressione si passa a classificazione (visto che la feature target sarebbe discreta). Si è provato a usare quindi nel progetto Adasyn per oversampling non ottenendo risultati disastrosi come immaginato ma comunque privi di significato visto che il task di apprendimento supervisionato è di regressione e non di classificazione e visti poi i valori delle metriche mse, msle e mae decisamente peggiorati rispetto a quelle del semplice task di apprendimento supervisionato senza oversampling.
Visto che la semplice applicazione dell'utilizzo di adasyn come esperimento per vedere il comportamento del task di regressione non mi soddisfaceva perché non lo ritenevo completamente pertinente alla risoluzione del nostro problema iniziale spulciando tra articoli e librerie python sono riuscita a trovare Smogn che è una libreria disponibile su python specializzata per l'oversampling in problemi di regressione sfruttando il rumore gaussiano.

6.2 SMOGN

Come già detto Smogn è una libreria che consente l'implementazione per task di regressione. Prima di arrivare a smogn si è provata un'altra libreria resreg che aveva al suo interno la funzione smoter che consentiva l'oversampling anche per la regressione. Il problema di questa libreria era il calcolo della variabile more_size (size in smoter) nella funzione dell'oversampling. Si è provato a usare vari dataset, anche più semplici di quello relativo al nostro problema, ma tutti i tentativi non hanno fatto supporre che il problema derivasse dal dataset o dalla manipolazione delle features. Si è quindi deciso di usare smogn visto che risultava una libreria più affidabile e più ristretta al nostro problema.

Prima di applicare l'oversampling al nostro dataset si sono applicate delle trasformazioni alle features per assicurarci che siano di tipo numerico (si è fatto perché altrimenti l'applicazione di smoter, la funzione principale nella libreria smogn, dava errore):

```
df = pd.read_csv(fileName_clusters, encoding='utf-8-sig')
targetColumn = 'log_price'
df[targetColumn] = pd.to_numeric(df[targetColumn], errors='coerce')
dfOver = df.copy()
dfOver = dfOver.replace({True: 1, False: 0}).infer_objects(copy=False)
X = dfOver.drop(columns=[targetColumn]).to_numpy()
y = dfOver[targetColumn].to_numpy()
assert len(X) == len(y)
print("Inizio oversampling")
X_over, y_over = oversampling_smogn(X, y, targetColumn)
print("Fine oversampling")
```

Si è poi quindi richiamata la funzione oversampling_smogn presente nello script *oversampling.py*:

```
def oversampling_smogn(X, y, target):
    X, y = np.asarray(X), np.squeeze(np.asarray(y))
    train_data = pd.concat( objs: [pd.DataFrame(X), pd.Series(y, name=target)], axis=1)

    train_data_resampled = smogn.smoter(data=train_data, y=target, samp_method='extreme')
    X = train_data_resampled.drop(columns=[target]).to_numpy()
    y = train_data_resampled[target].to_numpy()
    X = pd.DataFrame(X)
    dataSet_resampled = pd.DataFrame(X, columns=X.columns)
    dataSet_resampled[target] = y
    file_path = "../dataset/Automobile_resampled.csv"
    dataSet_resampled.to_csv(file_path, index=False)
    return X, y
```

All'interno di questa funzione, come si può vedere sopra, ci si assicura che X e y siano nel formato giusto. Si crea quindi train_data come dataFrame con il giusto formato di X e y e si applica smoter della libreria smogn. Si ricostruiscono X e y dal nuovo train_data_resampled e infine si crea il dataset_resampled ossia il dataset con i valori risultati dall'oversampling (che si può trovare in *dataset/Automobile_resampled.csv*).
Scendendo un po' più nel dettaglio della funzione principale di smogn possiamo dire che è una funzione che ha lo stesso scopo di smote dell'oversampling classico per classificazione. Smogn lavora effettuando l'undersampling della classe maggioritaria (per **undersampling** si intende la tecnica inversa all'oversampling, ossia cerca di ridurre il numero di esempi presenti nella classe maggioritaria cercando di raggiungere lo stesso scopo dell'oversampling) e l'oversampling della classe minoritaria nel dataset. La procedura inizia con il preprocessing dei dati per verificare se ci sono valori nulli. Si passa poi all'osservazione della variabile target y che viene ordinata per capire, tramite una funzione ϕ, se l'osservazione è normale o rara in base alla soglia del rel_thres (solitamente se non si specifica nulla, come nel nostro caso, rel_thres prende il valore di 0.5). Le osservazioni normali ottenute vengono quindi inserite in un sottoinsieme della classe maggioritaria detto normal bin, mentre le osservazioni rare in un sottoinsieme della classe minoritaria chiamato rare bin.
Per normal bin viene quindi applicato l'undersampling in base a una percentuale data dal valore della variabile samp_method (nel nostro caso è stato scelto extreme quindi avremo sia un oversampling che un undersamplig maggiore).
Per rare bin invece si applica l'oversampling con la tecnica smoter o smoter-gn. In entrambi i casi si applica l'interpolazione di smoter ma nel secondo caso, ossia quello di smoter-gn, si perturbano i valori con il rumore gaussiano. Per selezionare smoter o smoter-gn si considera la distanza tra l'osservazione data e un vicino selezionato. Se la distanza rientra entro una soglia massima si applica smoter altrimenti smoter-gn.
Si conclude il tutto con la restituzione di un dataframe pandas conteneti le osservazioni dopo l'oversampling e l'undersampling.

6.3 APPRENDIMENTO SUPERVISIONATO DOPO SMOGN

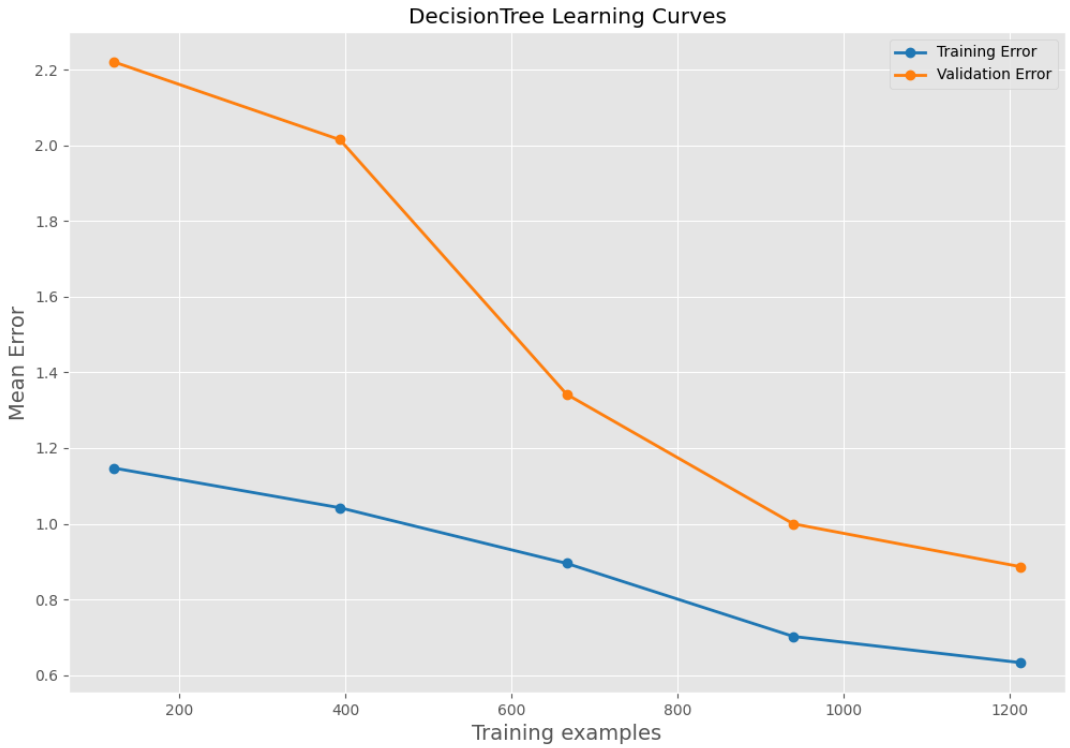
Dopo aver applicato smogn al nostro dataset abbiamo deciso di rifare l'apprendimento supervisionato sul nostro nuovo dataset per capire se e come ci sono stati dei miglioramenti nei nostri modelli.
Gli iperparametri selezionati per i modelli dopo l'oversampling sono stati (si possono trovare in *text/hyperparameters_oversampled.txt*):

```
Best DecisionTree parameters found:
DecisionTree__criterion            absolute_error
DecisionTree__max_depth            10
DecisionTree__min_samples_leaf     10
DecisionTree__min_samples_split    2
DecisionTree__random_state         42
DecisionTree__splitter             best
Best RandomForest parameters found:
RandomForest__criterion            friedman_mse
RandomForest__max_depth            None
RandomForest__n_estimators         10
RandomForest__random_state         42
Best LightGBM parameters found:
LightGBM__class_weight             balanced
LightGBM__learning_rate            0.1
LightGBM__max_depth                5
LightGBM__n_estimators             100
LightGBM__random_state             42
LightGBM__verbose                  -1
```

- Per il decisionTree abbiamo che le metriche sono (si possono trovare in *text/metrics_oversampling.txt*):

METRIC	SCORE MEAN	SCORE VARIANCE	SCORE STD
MSE	0.7514892412943163	0.08053836885390808	0.28379282734753547
MAE	0.4627029863736832	0.002805330356546403	0.05296536940819353
MSLE	0.0045791734047074	1.78317293136554e-06	0.00133535498327805
R2	0.6746483268417626	0.006235854200075272	0.07896742493000054

Possiamo già notare come rispetto a prima dell'oversampling, le metriche mse e msle hanno subito un aumento ma comunque continuano ad avere un valore basso, mae è leggermente diminuita che quindi è un buon segno e anche r2 è passato da un 62% a un 67%. Per quanto riguarda le metriche quindi l'oversampling con smogn ha portato un miglioramento per il decisionTree.
Osservando anche il grafico (che si può trovare in *png/decisionTree_oversampled_curve.png*):



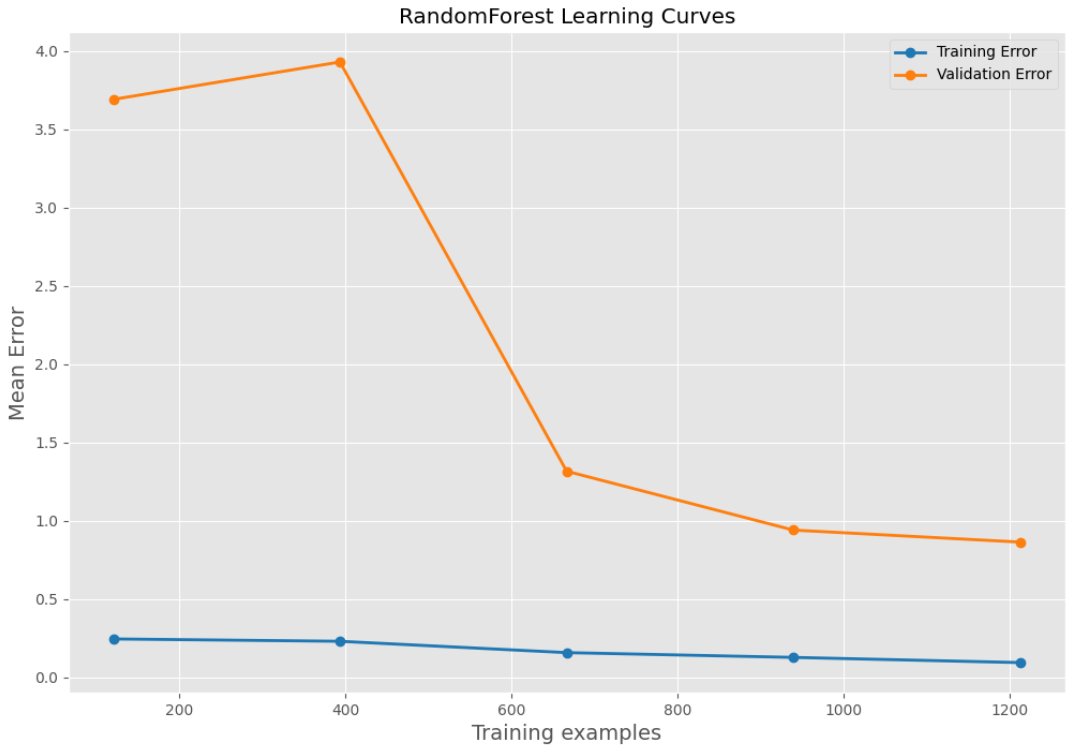
Anche in questo caso analizzando il grafico possiamo notare come rispetto a prima ci sia stato un miglioramento delle performance del modello (non risulta essere presente overfitting, nonostante il fatto che la curva del validation error non riesca ancora a colmare la distanza con quella del training error il fatto che continui a diminuire è un buon segno).
Possiamo dire in questo caso che in generale considerando le metriche e la curva di apprendimento l'oversampling con smogn ha dato un miglioramento del modello per il decisionTree dovuto soprattutto alla diminuzione del training error (nel grafico passato aumentava simbolo che il modello non riesce più ad adattarsi perfettamente ai dati di training con più esempi).

- Per il modello RandomForest abbiamo che le metriche sono:

METRIC	SCORE MEAN	SCORE VARIANCE	SCORE STD
MSE	0.6220274818879356	0.02931315715883631	0.17121085584400397
MAE	0.4238265278604739	0.001074105074860260	0.03277354229954797
MSLE	0.0037780592127712	5.85661449418015e-07	0.00076528520789181
R2	0.724849838831299	0.001765515260401192	0.04201803494216730

Per quanto riguarda le metriche per il randomForest rispetto a prima dell'oversampling possiamo notare un leggero peggioramento dovuto sia ad un leggero aumento della metrica mse (mae e msle in verità sono leggermente migliorate) come nel decisionTree, sia ad una leggera diminuzione da 73% a 72% di r2. In generale comunque il peggioramento non è grave, anzi è molto leggero. Per capire meglio

guardiamo la curva di apprendimento (che si può trovare in *png/randomForest_oversampled_curve.png*):



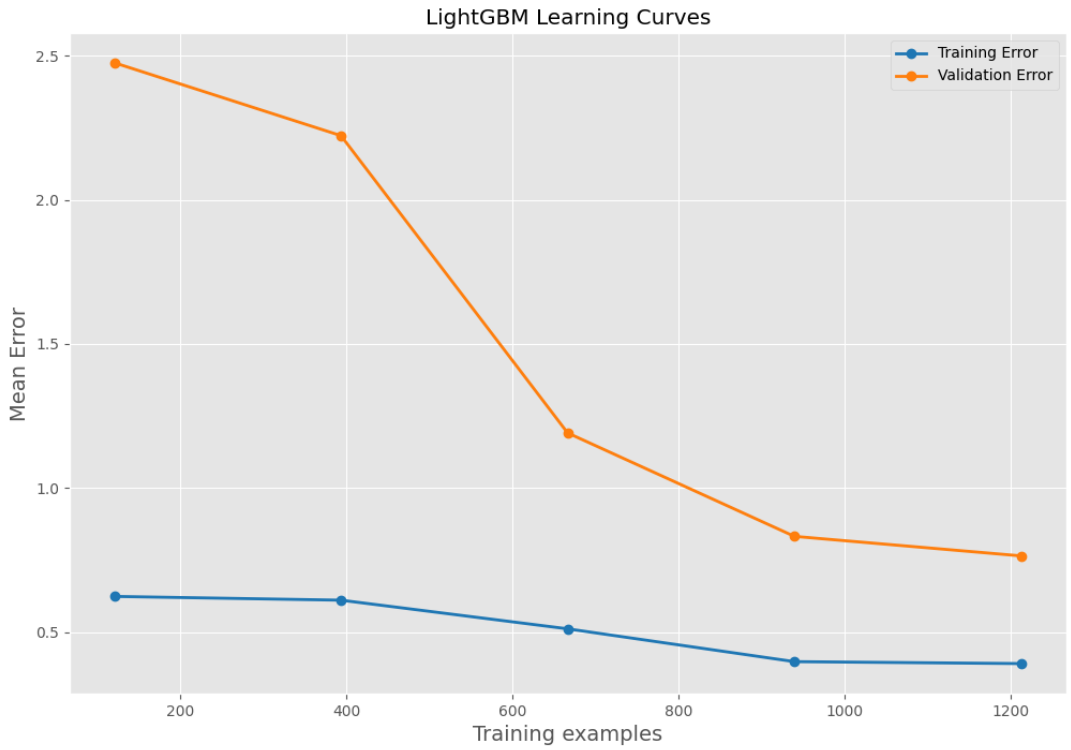
Analizzando la curva di apprendimento possiamo già notare come dopo l'oversampling con smogn la curva non presenti segni di overfitting (il che presenta un buon segno in generale e riprende quello che abbiamo già detto alla curva del decisionTree, forse la diminuzione più lenta del validation error potrebbe essere già un segnale di overfitting ma comunque al momento non presenta un problema). Rispetto quindi alla curva precedente non notiamo cambiamenti.
In generale quindi possiamo dire che RandomForest non ha particolarmente beneficiato dell'oversampling con smogn (certo la situazione non è neanche peggiorata se non leggermente per i valori delle metriche ma alla fine la differenza è minima che può essere trascurata).

- Infine consideriamo le metriche per LightGBM:

METRIC	SCORE MEAN	SCORE VARIANCE	SCORE STD
MSE	0.6371034637390943	0.04705949725128509	0.21693201066528905
MAE	0.44922173041617913	0.00160809752736873	0.04010109134884906
MSLE	0.00392749497644578	9.7519807528486e-07	0.00098752117713235
R2	0.722225095151677	0.00289019751583919	0.05376055725008065

Possiamo vedere come nel caso del LightGBM c'è stato un leggero aumento nella metrica mse mentre per mae e msle la situazione è rimasta invariata e invece abbiamo

avuto un buon aumento della r2 da 68 a 72. In generale quindi le metriche per il LightGBM hanno benificiato dell'oversampling.
Per quanto riguarda la cruva (che si può trovare in *png/lightGBM_oversampled_curve.png*):



Anche in questo caso, come nei precedenti, non sono presenti sintomi di overfitting dei dati e anzi rispetto al grafico precedente LightGBM sembra aver avuto un miglioramento vista la diminuzione della distanza tra validation e trainining error e inoltre anche il fatto che rispetto al grafico precedente, come già detto, non ci siano sintomi di overfitting (che possano essere più o meno gravi) fa pensare quindi che in generale l'oversampling con smogn abbia dato i suoi frutti.

In generale possiamo dire che l'oversampling con smogn sembra aver portato benefici a tutti i modelli soprattutto per quanto riguarda il training error che riesce ad adattarsi meglio ai dati. Portando LightGBM a diventare il miglior modello, a parimerito per le metriche con il RandomForest, ma migliore in fatto di curva di apprendimento. Il randomForest non lo definirei più come milgior modello principalmente visto che dopo l'oversampling la situazione non sembra essere molto migliorata e visto il leggero peggioramento della metrica r2. Anche decionTree è migliorato ma viene comunque superato dai due modelli precedenti.

7. APPRENDIMENTO PROBABILISTICO

Un altro tipo di apprendimento è quello **probabilistico** che si basa sulla teoria della probabilità che viene sfruttata per rappresentare e gestire l'incertezza intrinseca nei dati e nei modelli. Le **reti bayesiane** sono uno strumento fondamentale dell'apprendimento probabilistico. Esse sono dei modelli grafici probabilistici che rappresentano un insieme di variabili casuali e le loro dipendenze attraverso un grafo orientato aciclico (DAG).
Il meccanismo consiste nell'utilizzare il ragionamento probabilistico per apprendere una rete bayesiana, con l'obiettivo di analizzare il dataset e ottenere informazioni sulla distribuzione dei campioni al suo interno. Questo processo consente di generare nuovi esempi, utili per aggiornare le credenze dei modelli relative a un determinato fenomeno.

7.1 RETE BAYESIANA

Come primo passo dobbiamo definire la nostra rete bayesiana che deve avere la struttura di un grafo orientato aciclico(DAG). Abbiamo quindi cercato di individuare tutte le dipendenze più importanti tra le nostre variabili e definito la nostra rete bayesiana (*src/bayesian_net.py*):

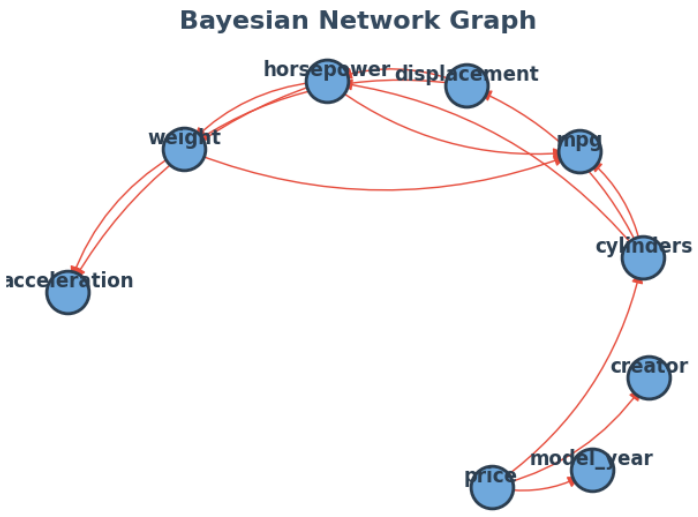
```python
def create_bayesian_network(dataset):
    edges = []
    edges.append(('price', 'model_year'))
    edges.append(('price', 'creator'))
    edges.append(('price', 'cylinders'))
    edges.append(('cylinders', 'mpg'))
    edges.append(('cylinders', 'displacement'))
    edges.append(('cylinders', 'horsepower'))
    edges.append(('displacement', 'weight'))
    edges.append(('displacement', 'horsepower'))
    edges.append(('horsepower', 'acceleration'))
    edges.append(('horsepower', 'weight'))
    edges.append(('horsepower', 'mpg'))
    edges.append(('weight', 'mpg'))
    edges.append(('weight', 'acceleration'))

    bn = BayesianNetwork(edges)
    bn.fit(dataset, estimator=MaximumLikelihoodEstimator, n_jobs=-1)
    # Salvo la rete bayesiana su file
    with open('../dataset/Bayesian_network.pkl', 'wb') as output:
        pickle.dump(bn, output)
    return bn
```

Qui possiamo ben vedere le dipendenze che sono state fatte e diamo adesso una spiegazione tecnica sul perché sono state scelte tali dipendenze (ovviamente è stato controllato che le dipendenze non creassero alcun ciclo all'interno del grafo e verificarlo è semplice visto che non ci sono frecce che puntano al senso opposto, non puntano indietro):

- [price, model_year] -> dipendenza creata visto che il prezzo varia con l'anno dei modelli (modelli con più anni hanno solitamente prezzi più alti, non considerando eventuali modelli particolari o rari);
- [price, creator] -> giustificata dal fatto che il marchio giustifica il prezzo;
- [price, cylinders] -> numero maggiore di cilindri implica un motore più grande e un prezzo più alto di conseguenza;
- [cylinders, mpg] -> più cilindri tendono a ridurre il carburante;
- [cylinders, displacement] -> un numero maggiore di cilindri è associato a una maggiore cilindrata;
- [cylinders, horsepower] -> più cilindri indicano una maggiore potenza (non contando la tecnologia del motore);
- [displacement, horsepower] -> anche in questo caso come nella penultima dipendenza una maggiore cilindrata indica una maggiore potenza (potrebbe anche essere una dipendenza un po' ridondante ma ho comunque preferito inserirla per indicarla in modo chiaro);
- [displacement, weight] -> maggiore cilindrata indica anche un maggiore peso dei veicoli;
- [weight, mpg] -> un maggiore peso indica un maggior consumo di carburante;
- [horsepower, acceleration] -> una maggiore potenza migliora anche l'accelerazione;
- [horsepower, weight] -> veicoli più pesanti richiedono più potenza;
- [weight, acceleration] -> veicoli più pesanti hanno una minore accelerazione;
- [horsepower, mpg] -> veicoli più potenti consumano più carburante.

Alla fine la rete bayesiana creata è stata la seguente (*png/Bayesian_network_graph.png*):



Sono state poi calcolate le varie cpd (distribuzioni di probabilità condizionata) delle variabili in base alle dipendenze (si possono trovare in *text/bayesian_network_cpd.txt*):

```
CPD of price:
+------------+-----------+
| price(0.0) | 0.0729761 |
+------------+-----------+
| price(1.0) | 0.126568  |
+------------+-----------+
| price(2.0) | 0.0900798 |
+------------+-----------+
| price(3.0) | 0.0752566 |
+------------+-----------+
| price(4.0) | 0.0980616 |
+------------+-----------+
| price(5.0) | 0.129989  |
+------------+-----------+
| price(6.0) | 0.107184  |
+------------+-----------+
| price(7.0) | 0.0992018 |
+------------+-----------+
| price(8.0) | 0.0980616 |
+------------+-----------+
| price(9.0) | 0.102623  |
+------------+-----------+
```

Possiamo notare come la nostra variabile price è indipendente visto che ha un cpd che non è influenzato da altre variabili all'interno della rete bayesiana. (Voglio sottolineare il fatto che tutti i valori ottenuti sono dati da varie fasi di preprocessing e una discretizzazione delle colonne usando la strategia quantile che consente di avere una maggiore precisione nei dati che si può proprio riflettere nel file cpd delle variabili. Si era infatti usata un'altra strategia, la uniform, che però non consentiva di avere valori precisi che creavano poi esempi sintetici meno realistici di quelli che poi mostreremo nel prossimo paragrafo).

7.2 GENERAZIONE DI SAMPLE

Come abbiamo già detto, con l'apprendimento probabilistico e quindi la creazione della rete bayesiana è possibile generare degli esempi sintetici. Questo è uno degli esempi che sono stati generati:

Example:	displacement	mpg	horsepower	creator	cylinders	model_year	weight	acceleration
0	4.0	9.0	1.0	volkswagen	6	1977	3.0	9.0

(consideriamo anche che i valori fatti vedere, tranne model_year e cylinders, sono stati discretizzati).

Provando a predire il prezzo della macchina, abbiamo ottenuto:

price	phi(price)
price(0.0)	0.0000
price(1.0)	0.2918
price(2.0)	0.0000
price(3.0)	0.1806
price(4.0)	0.1772
price(5.0)	0.2882
price(6.0)	0.0622
price(7.0)	0.0000
price(8.0)	0.0000
price(9.0)	0.0000

In questo caso notiamo che la predizione del prezzo ci indica un valore abbastanza basso. Analizzando l'esempio generato a primo impatto ci potrebbe sembrare strano visto che tutte le features sembrano avere un valore abbastanza alto che indicherebbe un prezzo molto più alto. Guardando bene notiamo che horsepower è leggermente sopra i valori minimi. Questo è un'ottima giustificazione della predizione del prezzo basso visto che con un horsepower piccolo la macchina ha pochissima potenza e questo influenza negativamente il prezzo.

Possiamo quindi dire come la nostra rete bayesiana genera esempi abbastanza buoni e realistici generalmente. La si potrebbe usare per ampliare il nostro dataset e migliorarlo quindi, portando così benefici ai vari task.

7.3 SAMPLE DATO L'INPUT UTENTE

Si è poi deciso di rendere il tutto più interattivo con l'utente consentendo a quest'ultimo la possibilità di inserire i valori delle feature di input dell'agente e gli verrà restituita la predizione del prezzo come si può vedere sopra.

```
Vuole inserire i valori dell'auto per predire il prezzo della sua macchina? S/nsi
Inserire l'mpg (valore minimo 5 e massimo 200):200
Inserire il numero di cilindri (valori possibili: 2, 3, 4, 6, 8, 10, 12):12
Inserire la cilindrata (valore minimo 70 e massimo 500):500
Inserire la potenza (valore minimo 50 e massimo 400):400
Inserire il peso (valore minimo 1000 e massimo 5000):5000
Inserire l'accelerazione (valore minimo 2 e massimo 25):25
Inserire l'anno del modello (valore minimo 1970 e massimo 2020):2020
Inserire la caso automobilista di produzione del modello
 Scrivere una delle seguenti opzioni:
-chevrolet
 -buick
 -plymouth
 -amc
 -ford
 -pontiac
 -dodge
, -toyota
 -datsun
 -peugeot
 -audi
 -saab
 -bmw
 -opel
 -fiat
 -volkswagen
 -mercury
 -oldsmobile
 -chrysler
 -mazda
 -volvo
 -renault
 -honda
 -mercedes
 -subaru
```

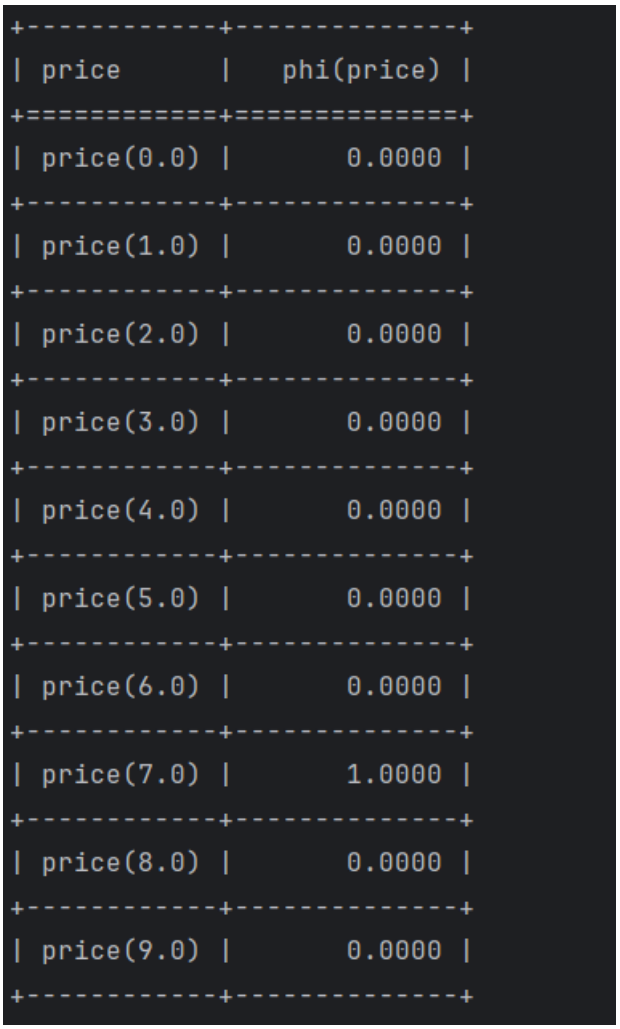
```
 -audi
 -saab
 -bmw
 -opel
 -fiat
 -volkswagen
 -mercury
 -oldsmobile
 -chrysler
 -mazda
 -volvo
 -renault
 -honda
 -mercedes
 -subaru
 -nissan
 -porsche
 -ferrari
 -mitsubishi
 -jeep
 -jaguar
 -lamborghini
volvo
```

Come si può vedere dagli screen sopra l'utente può tranquillamente inserire i valori che rientrano nel range (si sono presi dei valori standard sia per auto economiche sia per auto di lusso/sportive per cercare di effettuare dei controlli sull'input dell'utente onde evitare di avere esempi poco realistici. Se si vuole impostare il minimo a 0 il minimo e aumentare il massimo si può fare. Anche per quanto riguarda le case automobilistiche sono state scelte quelle già presenti nel sistema (che comunque sono tante) e sono date all'utente per consentirgli di sapere quali case sono state prese in considerazione per rendere l'input valido). Infine si è poi fatto vedere a stampa all'utente i valori scelti e la loro discretizzazione.

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	creator
0	200	12	500	400	5000	25	2020	volvo

	cylinders	acceleration	model_year	creator	mpg	weight	horsepower	displacement
0	12	9.0	2020	volvo	9.0	9.0	9.0	9.0

Le variabili sono state discretizzate con la stessa tecnica con cui è stata discretizzata la rete bayesiana.
Come detto nel paragrafo precedente, le predizioni sono abbastanza buone:



price	phi(price)
price(0.0)	0.0000
price(1.0)	0.0000
price(2.0)	0.0000
price(3.0)	0.0000
price(4.0)	0.0000
price(5.0)	0.0000
price(6.0)	0.0000
price(7.0)	1.0000
price(8.0)	0.0000
price(9.0)	0.0000

Considerando la tipologia di macchina data al sistema possiamo tranquillamente confermare quello che abbiamo detto poco fa. Ossia che la predizione è buona, essa infatti dà al veicolo dell'utente un valore molto alto ma non troppo considerando il valore dell'accelerazione della casa automobilistica di appartenenza del modello.

8. CONCLUSIONI

All’interno di questo caso di studio sono state esplorate le varie tecniche per l’apprendimento supervisionato utilizzando vari modelli come randomforest, decisiontree e lightgbm. Si è vista in particolare la tecnica smogn per effettuare oversampling per migliorare i modelli sopra citati dando una visione dell’oversampling per regressione. Si è visto altre tecniche come l’apprendimento non supervisionato per suddividere il dataset in clusters. Abbiamo creato e appreso una rete bayesiana sfruttandola per la creazione di esempi randomici e per poter far creare all’utente una sua domanda che gli restituisse poi la probabilità del range del prezzo (dico range perché è stato tutto discretizzato).
In generale possiamo concludere che si sono ottenuti i risultati sperati. Il tutto richiederebbe un miglioramento dal punto di vista di un aumento degli esempi del dataset, della qualità e una maggiore lavorazione delle feature di contesto.
Possiamo quindi dire che per gli sviluppi futuri si può pensare di lavorare maggiormente sul dataset in fatto di feature di contesto (ossia capire meglio il contesto delle auto e dare feature magari anche più dettagliate per il contesto del mondo delle auto) sia in fatto di un aumento degli esempi presenti. Si può pensare di ampliare il task di apprendimento supervisionato prendendo in considerazione altri modelli. Si può anche pensare di mettere in atto altre strategie di oversampling prendenod spunto anche da recenti articoli sull’oversampling per problemi di regressione.

RIFERIMENTI BIBLIOGRAFICI

Ragionamento logico: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3rd Ed. Cambridge University Press [Ch.5]

Apprendimento non supervisionato: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3rd Ed. Cambridge University Press [Ch.10]

Apprendimento supervisionato: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3rd Ed. Cambridge University Press [Ch.7]

Tecniche di oversampling: https://sbelhaouari.github.io/assets/publications/2024/knnor-reg.pdf

Ragionamento probabilistico e reti bayesiane: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3rd Ed. Cambridge University Press [Ch.9]

Metriche: https://scikit-learn.org/stable/modules/model_evaluation.html