Progetto preliminare programmazione data intensive Albonetti Simone

Previsione del prezzo di diamanti

simone.albonetti@studio.unibo.it

Setup

- Installare il package opendatasets (nel caso non fosse già presente) per scaricare direttamente il dataset da Kaggle
- Importare le librerie necessarie

In [1]:
```
#pip install opendatasets
import numpy as np
import pandas as pd
import os.path
import opendatasets as od
import matplotlib.pyplot as plt


from sklearn.exceptions import ConvergenceWarning
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import r2_score
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from sklearn.linear_model import ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.tree import plot_tree
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score

import warnings
warnings.simplefilter("ignore", category=ConvergenceWarning)
```

Predizione del valore dei diamanti

Questo notebook si occupa di determinare il prezzo di un diamante. Questo valore dipende dalle caratteristiche fisiche che possiede.
Per prima cosa bisogna scaricare il dataset contenente le informazioni sui diamanti.

Per scaricare il dataset:

1. Eseguire la cella seguente
2. Inserire il proprio **username** Kaggle
3. Inserire il **token api** Kaggle

In [2]:
```
if not os.path.exists("/diamonds/diamonds.csv"):
    od.download("https://www.kaggle.com/datasets/shivam2503/diamonds/download?da
```

```
Skipping, found downloaded files in ".\diamonds" (use force=True to force downlo
ad)
```

Altrimenti è possibile scaricare manualmente il dataset da questo link:
https://www.kaggle.com/datasets/shivam2503/diamonds

Il dataset è sotto forma di file CSV.
Per leggere questo file usiamo la funzione `read_csv` di Pandas per creare direttamente il dataframe `data` contenente i dati (*utilizziamo la prima colonna come indice per le righe*).

In [3]:
```
data = pd.read_csv("diamonds/diamonds.csv", index_col=0)
```

1a) Descrizione del problema e comprensione dei dati

Il dataset è composto da 53940 istanze. Le feature di ogni diamante sono:

- `carat` : Indica il peso del diamante.
- `cut` : Indica la qualità del taglio.
- `color` : Indica il colore del diamante.
- `clarity` : Indica la purezza del diamante. Cioè quante impurità sono presenti all'interno.
- `x` , `y` , `z` : Indicano le dimensioni del diamante (lunghezza, larghezza e profondità) in **mm**
- `depth` : Questa proprietà è chiamata **total depth percentage** e indica se il diamante è *sotto peso* oppure *sopra peso*. E' un dato derivabile infatti per calcolarlo la formula è **2 * z / (x + y)** , quindi **ATTENZIONE ALLA COLLINEARITA'**
- `table` : Indica la dimensione della superficie piatta che si trova sopra al diamante.
- `price` : Indica il prezzo in **dollari** del diamante. Questa è la **VARIABILE DA PREDIRE** in funzione delle altre.

Poichè `price` è una variabile **continua** si tratta di un problema di *Regressione* (*multivariata* perchè dipende da più di una feature)

In [4]:
```
data.head()
```

Out[4]:

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

1b) Analisi esplorativa dei dati

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 53940 entries, 1 to 53940
Data columns (total 10 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   carat    53940 non-null  float64
 1   cut      53940 non-null  object 
 2   color    53940 non-null  object 
 3   clarity  53940 non-null  object 
 4   depth    53940 non-null  float64
 5   table    53940 non-null  float64
 6   price    53940 non-null  int64  
 7   x        53940 non-null  float64
 8   y        53940 non-null  float64
 9   z        53940 non-null  float64
dtypes: float64(6), int64(1), object(3)
memory usage: 4.5+ MB
```

Variabili CATEGORICHE

Alcune feature sono categoriche (`cut` , `color` e `clarity`), quindi le dichiariamo di tipo `category` (andando anche a ridurre lo spazio del dataset).

In [6]:
```
custom_dtypes = {
    "cut" : "category",
    "color" : "category",
    "clarity" : "category"
}

categorical_var = ["cut", "color", "clarity"]
numerical_var = ["carat", "table", "price", "x", "y", "z" ]
numerical_var_not_price = numerical_var.copy()
numerical_var_not_price.remove("price")

data = pd.read_csv("diamonds/diamonds.csv", index_col=0, dtype=custom_dtypes)
```

In [7]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 53940 entries, 1 to 53940
Data columns (total 10 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   carat    53940 non-null  float64
 1   cut      53940 non-null  category
 2   color    53940 non-null  category
 3   clarity  53940 non-null  category
 4   depth    53940 non-null  float64
 5   table    53940 non-null  float64
 6   price    53940 non-null  int64
 7   x        53940 non-null  float64
 8   y        53940 non-null  float64
 9   z        53940 non-null  float64
dtypes: category(3), float64(6), int64(1)
memory usage: 3.4 MB
```

I possibili valori delle variabili categoriche sono:

In [8]:
```
print("data['cut']:" , data["cut"].unique(), "\n")
print("data['color']:" , data["color"].unique(), "\n")
print("data['clarity']:" , data["clarity"].unique())
```

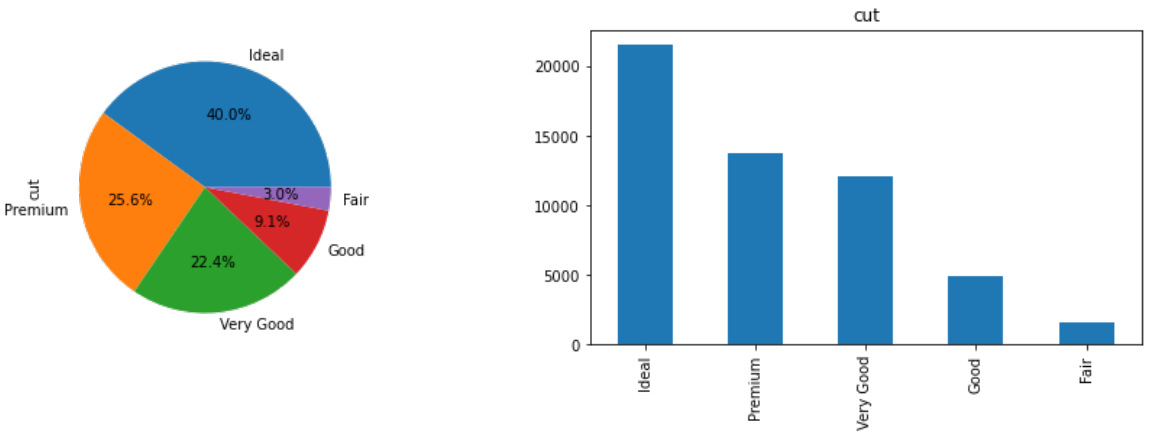
```
data['cut']: ['Ideal', 'Premium', 'Good', 'Very Good', 'Fair']
Categories (5, object): ['Fair', 'Good', 'Ideal', 'Premium', 'Very Good']

data['color']: ['E', 'I', 'J', 'H', 'F', 'G', 'D']
Categories (7, object): ['D', 'E', 'F', 'G', 'H', 'I', 'J']

data['clarity']: ['SI2', 'SI1', 'VS1', 'VS2', 'VVS2', 'VVS1', 'I1', 'IF']
Categories (8, object): ['I1', 'IF', 'SI1', 'SI2', 'VS1', 'VS2', 'VVS1', 'VVS2']
```

Le distribuzioni delle feature categoriche sono le seguenti (**normalizzate**):

In [9]:
```
print(data["cut"].value_counts(normalize=True) * 100)

plt.figure(figsize=(15, 4))
data["cut"].value_counts(normalize=True).plot.pie(autopct='%1.1f%%', ax=plt.subp
data["cut"].value_counts().plot.bar(ax=plt.subplot(1, 2, 2), title = "cut")
```

```
Ideal        39.953652
Premium      25.567297
Very Good    22.398962
Good          9.095291
Fair          2.984798
Name: cut, dtype: float64
```

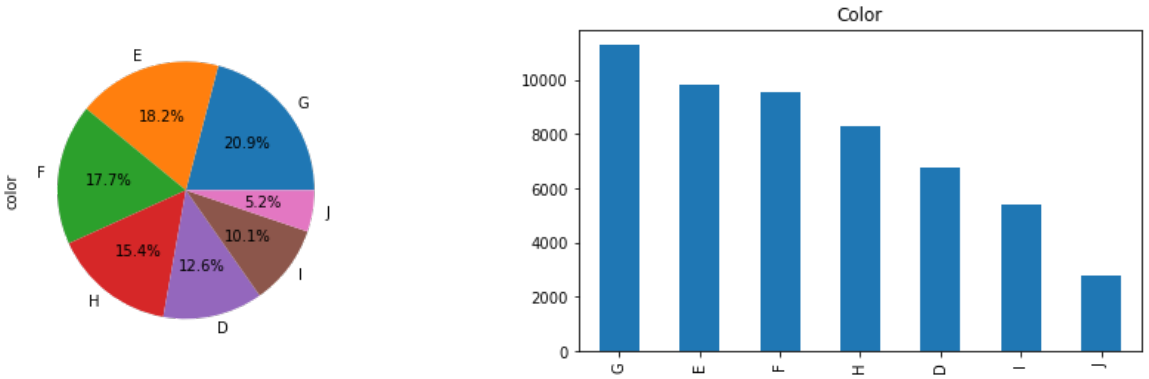
Out[9]:
```
<AxesSubplot:title={'center':'cut'}>
```



In [10]:
```
print(data["color"].value_counts(normalize=True) * 100)
plt.figure(figsize=(15, 4))
data["color"].value_counts(normalize=True).plot.pie(autopct='%1.1f%%', ax=plt.su
data["color"].value_counts().plot.bar(ax=plt.subplot(1, 2, 2), title = "Color")
```

```
G    20.934372
E    18.162773
F    17.690026
H    15.394883
D    12.560252
I    10.051910
J     5.205784
Name: color, dtype: float64
```

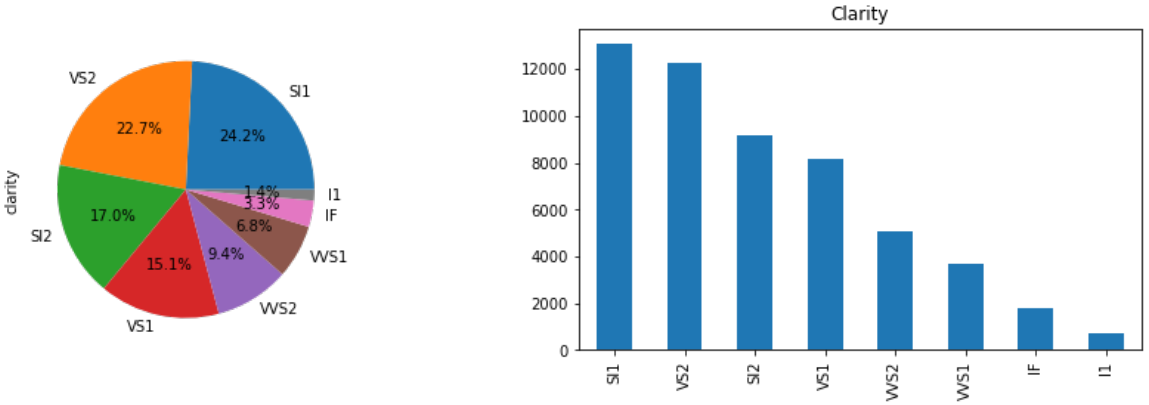
Out[10]:
```
<AxesSubplot:title={'center':'Color'}>
```



In [11]:
```
print(data["clarity"].value_counts(normalize=True) * 100)
plt.figure(figsize=(15, 4))
data["clarity"].value_counts(normalize=True).plot.pie(autopct='%1.1f%%', ax=plt.
data["clarity"].value_counts().plot.bar(ax=plt.subplot(1, 2, 2), title = "Clarit
```

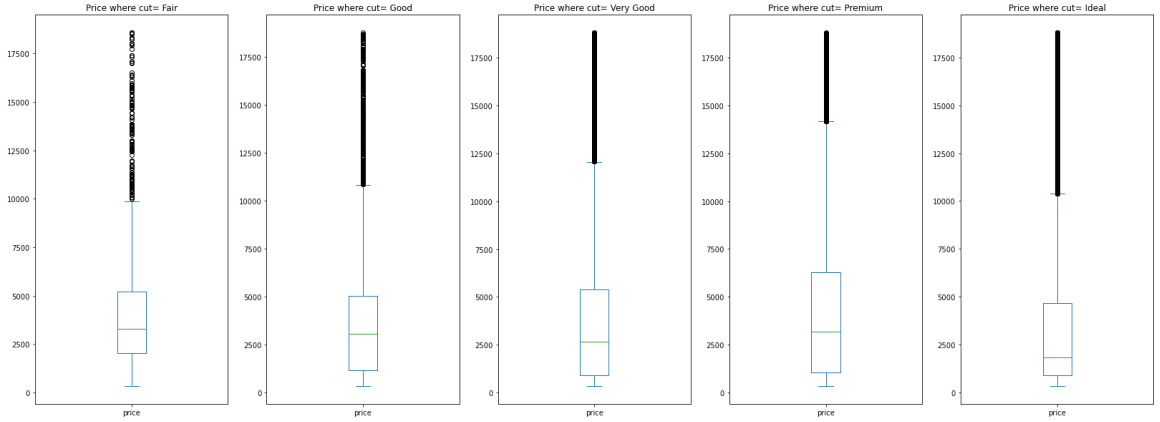
```
SI1     24.221357
VS2     22.725250
SI2     17.044865
VS1     15.148313
VVS2     9.391917
VVS1     6.776047
IF       3.318502
I1       1.373749
Name: clarity, dtype: float64
```

Out[11]:
```
<AxesSubplot:title={'center':'Clarity'}>
```

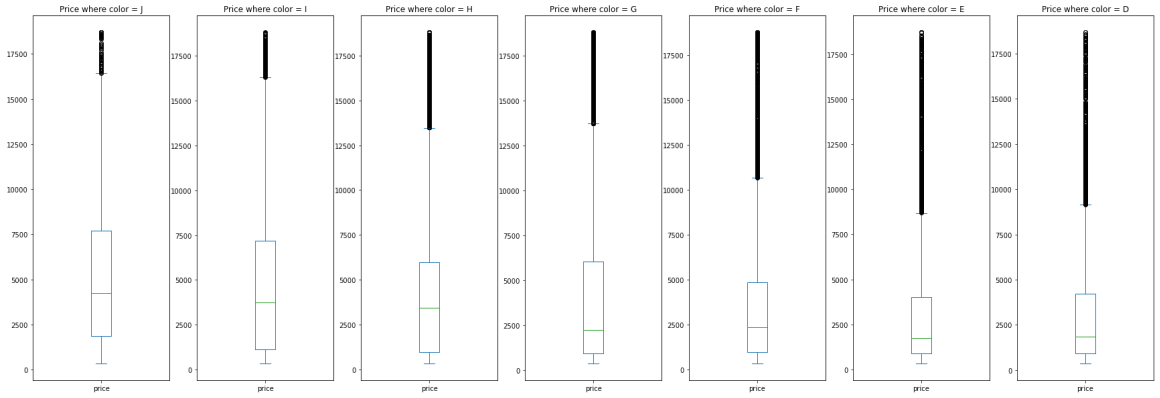


Dopo aver visto le distribuzioni delle variabili categoriche controlliamo se ci sono eventuali correlazioni tra queste variabili e `price` .

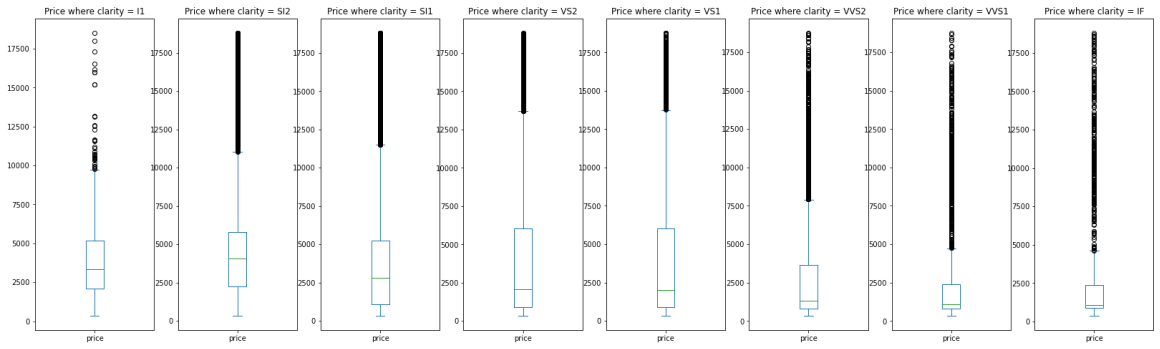
In [12]:
```
plt.figure(figsize=(28, 10))
for index, value in enumerate(["Fair", "Good", "Very Good", "Premium", "Ideal"],
    data[data["cut"] == value]["price"].plot.box(ax=plt.subplot(1,5,index), titl
```



In [13]:
```
plt.figure(figsize=(30,10))
for index,value in enumerate(sorted(data["color"].unique(), reverse=True), start
    data[data["color"] == value]["price"].plot.box(ax=plt.subplot(1,7,index), ti
```



In [14]:
```
plt.figure(figsize=(28, 8))
for index, value in enumerate(["I1", "SI2", "SI1", "VS2", "VS1", "VVS2", "VVS1",
    data[data["clarity"] == value]["price"].plot.box(ax=plt.subplot(1,8,index),
```

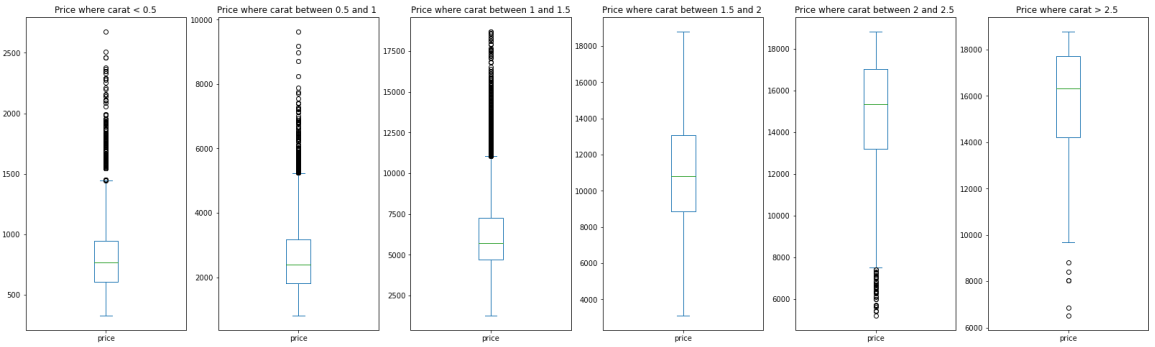


Per quanto riguarda le variabili `cut`, `color` e `clarity` non sembra esserci correlazione con la variabile `price` perchè al variare del valore delle categorie, il valore di `price` non cambia considerevolmente. Come succede invece nella cella seguente utilizzando la variabile `carat`.

Infatti andando a creare delle "categorie" fittizie (in realtà sono stati creati degli intervalli nella variabile `carat` per simulare delle categorie), si nota che la scala di valore di `price` cambia da categoria a categoria.

In [15]:
```
plt.figure(figsize=(28, 8))
data[data["carat"] < 0.5]["price"].plot.box(ax=plt.subplot(1,6,1), title=("Price
data[(data["carat"] < 1) & (data["carat"] > 0.5)]["price"].plot.box(ax=plt.subpl
data[(data["carat"] < 1.5) & (data["carat"] > 1)]["price"].plot.box(ax=plt.subpl
data[(data["carat"] < 2) & (data["carat"] > 1.5)]["price"].plot.box(ax=plt.subpl
data[(data["carat"] < 2.5) & (data["carat"] > 2)]["price"].plot.box(ax=plt.subpl
data[data["carat"] > 2.5]["price"].plot.box(ax=plt.subplot(1,6,6), title=("Price
```

Out[15]:
```
<AxesSubplot:title={'center':'Price where carat > 2.5 '}>
```



Variabili CONTINUE

Come si può notare dalla cella seguente, non ci sono valori mancanti, infatti tutte le feature hanno lo stesso numero di valori, pari a quello delle istanze (539400).

In [16]:
```
data.describe()
```

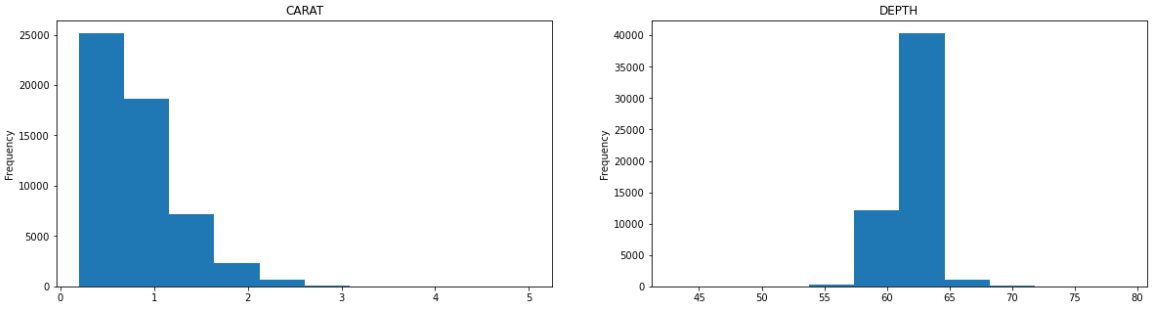
Out[16]:

	carat	depth	table	price	x	y	
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	

Però ci sono alcuni diamanti che sono senza dimensioni (perchè il min di `x`, `y` e `z` è 0). Per questo motivo bisogna **eliminare le istanze che non hanno dimensioni**.

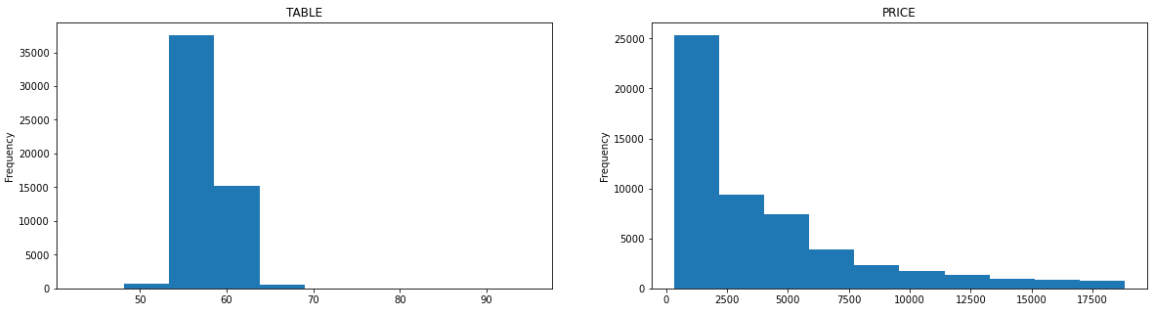
I grafici seguenti mostrano le distribuzioni delle variabili continue (`carat`, `depth`, `table`, `x`, `y` e `z`).

In [17]:
```
plt.figure(figsize=(20, 5))
for n, col in enumerate(["carat", "depth"], start=1):
    data[col].plot.hist(ax=plt.subplot(1, 2, n), title = col.upper())
```



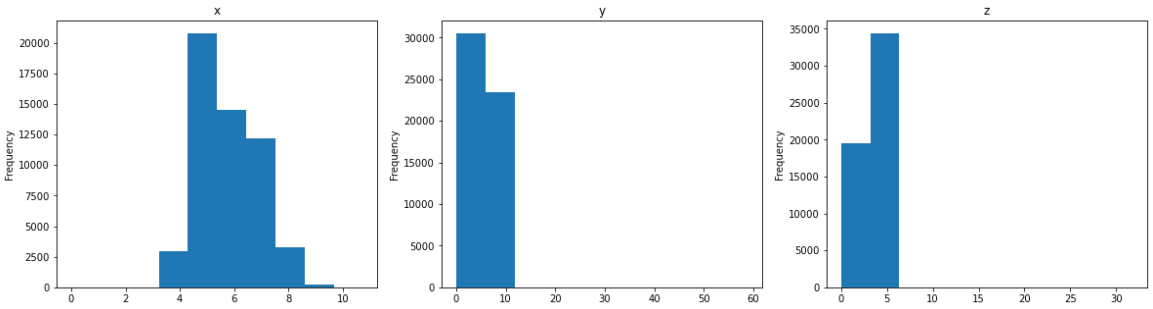
- **`carat`** : Il valore massimo è 5.01, ma valori superiori a 1 sono meno comuni
- **`depth`** : In questo caso abbiamo una grossa distribuzione di valori tra 60 e 63, mentre gli altri valori sono poco comuni.

In [18]:
```
plt.figure(figsize=(20, 5))
for n, col in enumerate(["table", "price"], start=1):
    data[col].plot.hist(ax=plt.subplot(1, 2, n), title = col.upper())
```



- **`table`** : Come per la `depth` anche qui abbiamo una grossa distribuzione di valori intorno al 57, mentre altri valori sono meno comuni.
- **`price`** : Il suo valore massimo è 18823, ma la maggior parte dei valori si trova tra 0 e 2500, valori maggiori sono meno comuni.

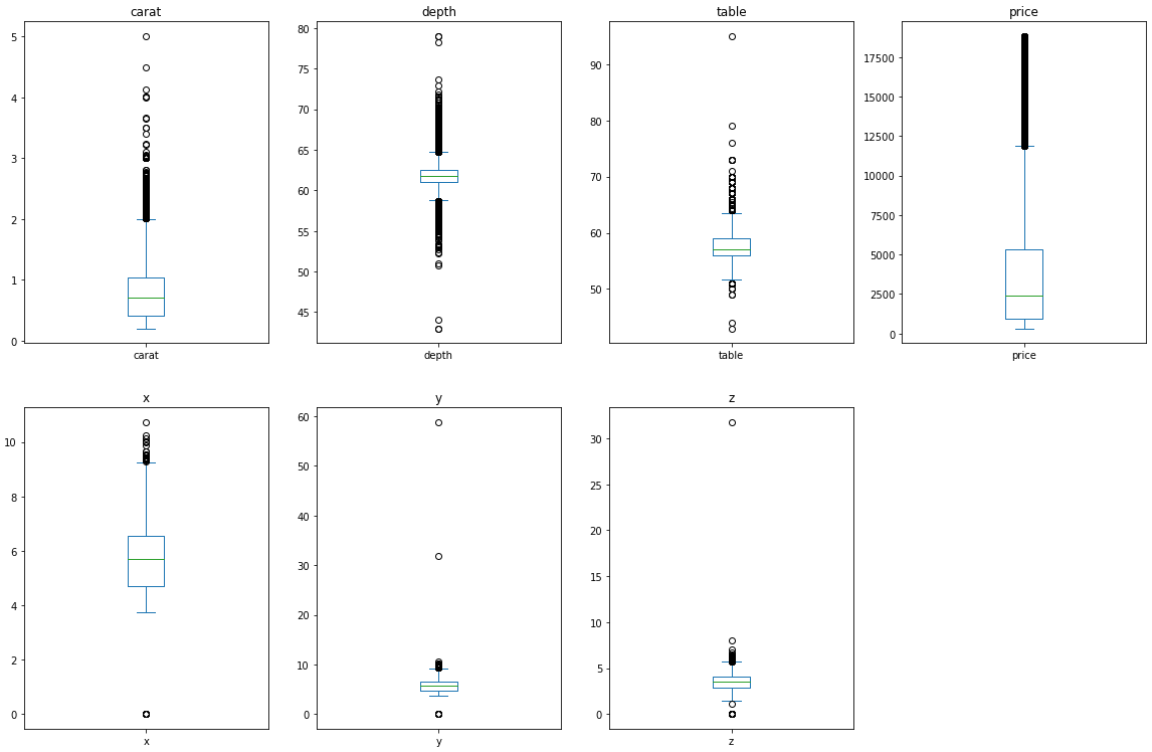
In [19]:
```
plt.figure(figsize=(20, 5))
for n, col in enumerate(["x", "y", "z"], start=1):
    data[col].plot.hist(ax=plt.subplot(1, 3, n), title = col)
```



- **`x`** , **`y`** , **`z`** : Le distribuzioni delle dimensioni sono abbastanza simili ed omogenee a parte per qualche istanza che ha valori di altezza e profondità (`y` e `z`) molto elevati (**outlier**).

Nella cella seguenti sono presenti dei *box plot* per evidenziare tutti gli **outlier** di tutte le variabili continue.

In [20]:
```
plt.figure(figsize=(20, 20))
for n, col in enumerate(["carat", "depth", "table", "price", "x", "y", "z"], sta
    data[col].plot.box(ax=plt.subplot(3, 4, n), title = col)
```



Si vede chiaramente che nelle variabili `depth`, `table`, `y` e `z` ci sono degli **outlier** che hanno valori molto distanti dai valori "ordinari".
Quindi andranno eliminati.

2) Preparazione dei dati

- Eliminiamo quelle istanze che hanno 1 o più dimensioni uguale a 0

In [21]:
```
data.drop(data[data['x'] == 0].index, inplace=True)
data.drop(data[data['y'] == 0].index, inplace=True)
data.drop(data[data['z'] == 0].index, inplace=True)
print(data.shape)
```

```
(53920, 10)
```

Il numero di istanze è passato a 53920

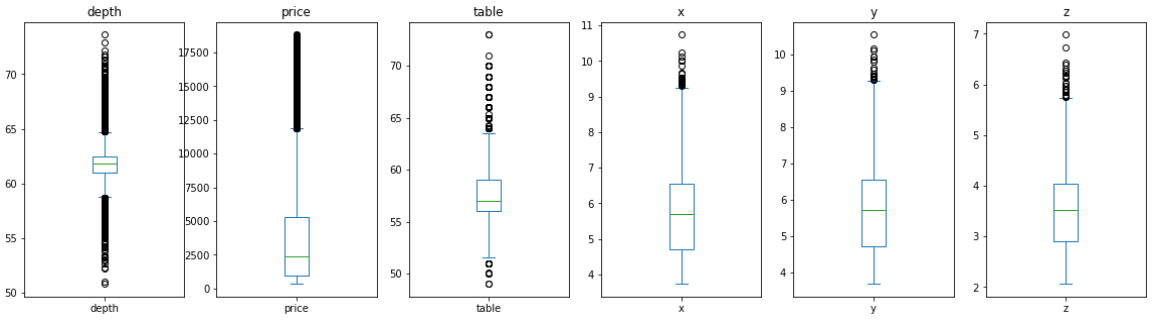
- Eliminiamo le istanze che sono delgli outlier per le variabili `depth`, `table`, `y` e `z`

In [22]:
```
data.drop(data[data["depth"] > 75].index, inplace=True)
data.drop(data[data["depth"] < 45].index, inplace=True)
data.drop(data[data["table"] > 75].index, inplace=True)
data.drop(data[data["table"] < 45].index, inplace=True)
data.drop(data[data["y"] > 15].index, inplace=True)
data.drop(data[data["z"] > 10].index, inplace=True)
data.drop(data[data["z"] < 2].index, inplace=True)
print(data.shape)
```

```
(53903, 10)
```

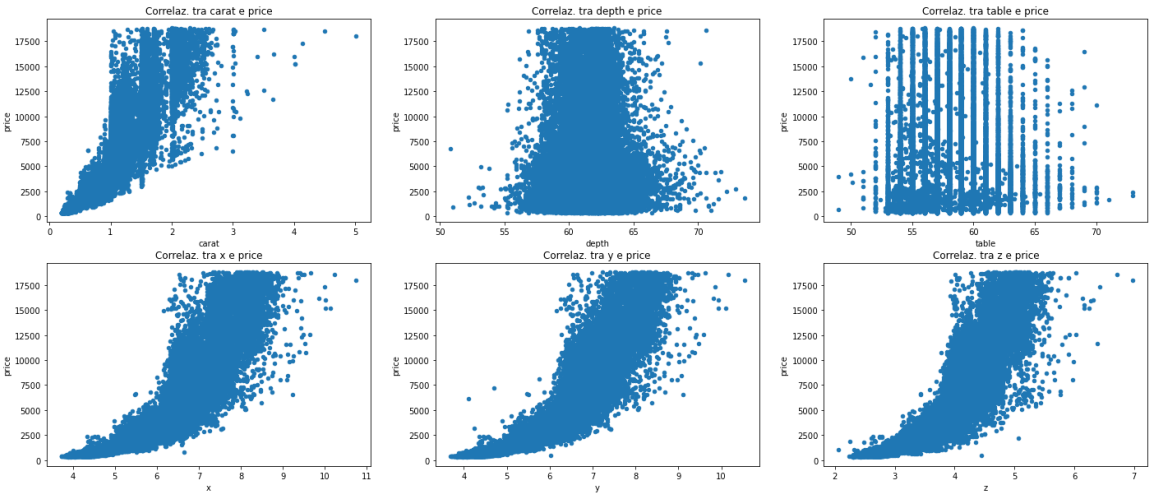
Ora il numero di istanze è 53903, e come si può vedere dai grafici seguenti, gli *outlier rimasti sono meno distanti/significativi*.

In [23]:
```
plt.figure(figsize=(20, 5))
for n, col in enumerate(["depth","price", "table", "x", "y", "z"], start=1):
    data[col].plot.box(ax=plt.subplot(1, 6, n), title = col)
```



Mostriamo attraverso un grafico a dispersione le varie correlazioni con la variabile `price` .

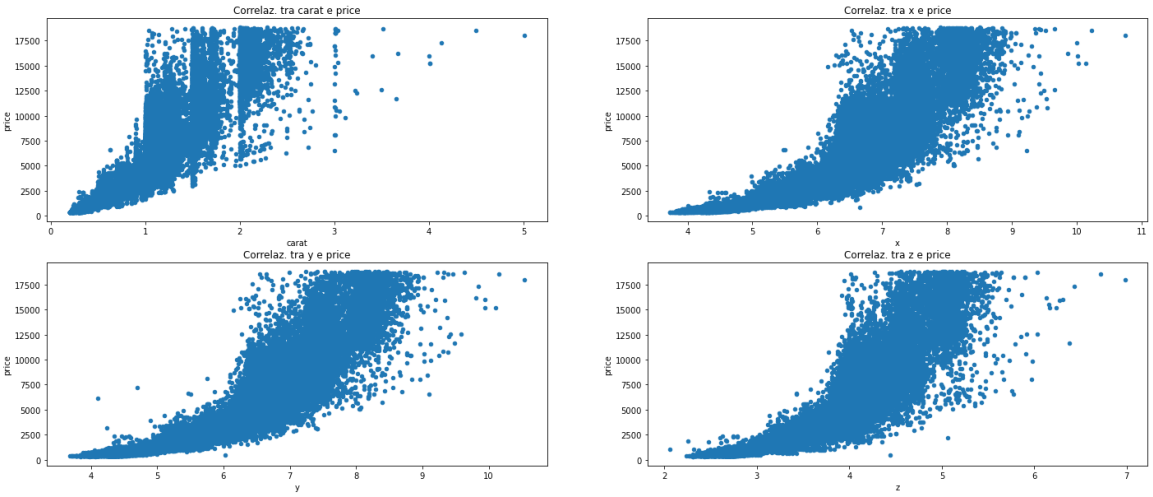
In [24]:
```
plt.figure(figsize=(24, 10))
for n, col in enumerate(["carat", "depth", "table", "x", "y", "z"], start=1):
    data.plot.scatter(col, "price", ax=plt.subplot(2, 3, n), title = "Correlaz.
```



`depth` e `table` non mostrano una correlazione con `price` , mentre le dimensioni `x` , `y` , `z` e `carat` sembrano avere una relazione con `price` .

Inoltre, `depth` è una variabile dipendente e derivabile da `x` , `y` e `z` . Quindi per non generare *collinearità* **NON VERRA' PRESA IN CONSIDERAZIONE**

In [25]:
```
plt.figure(figsize=(24, 10))
for n, col in enumerate(["carat", "x", "y", "z"], start=1):
    data.plot.scatter(col, "price", ax=plt.subplot(2, 2, n), title = "Correlaz.
```



Per determinare quali siano le feature più rilevanti, andremo a utilizzare la regressione **`Lasso`** , che andrà ad eliminare le feature meno significative.

All'inizio inseriremo tutte le feature all'interno del modello, quindi useremo:

- `OneHotEncoder` per binarizzare le variabili categoriche
- `MinMaxScaler` per normalizzare le variabili numeriche

Utilizziamo una `Pipeline` per eseguire tute queste operazioni e anche un `ColumnTransformer` per poter applicare l' `OneHotEncoder` e il `MinMaxScaler` solo alle colonne scelte (passate come argomenti).

Per prima cosa generiamo i 4 set necessari all'addestramento e alla validazione del modello (`x_complete_train` , `x_complete_val` , `y_complete_train` e `y_coplete_val`).

In [26]:
```
x_complete = data.drop(columns=["price", "depth"])
y_complete = data["price"]

x_complete_train, x_complete_val, y_complete_train, y_complete_val = train_test_
```

Prima di iniziare, creo questi 4 dizionari che avranno come **chiave**, il *nome dell'algoritmo utilizzato*, e come **valore**, la *valutazione ottenuta*.

Serviranno nel punto 3), quando faremo le valutazioni degli algoritmi

In [27]:
```
models_mse = {}
models_re = {}
models_r2 = {}
models_rmspe = {}
models_hyperparameter = {}
models_feature_coeff = {}

# Contiene lo scarto quadratico medio e la deviazione standard dell'errore degli
# ottenuti in k cross fold validation
models_result_gs_kf = {}
```

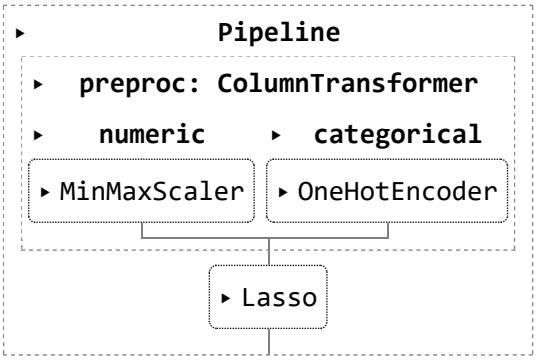
Creiamo una funzione come quella usata nei laboratori per determinare l'efficacia dei modelli.

In [28]:
```
def relative_error(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true))

def rmspe(y_real, y_pred):
    return np.sqrt(np.mean((y_pred /y_real - 1) ** 2))

def print_eval(X, y, model, name_for_dictionary=1, hyperparameter=None, coeff=No
    y_pred = model.predict(X)
    mse = mean_squared_error(y, y_pred)
    re = relative_error(y,y_pred)
    r2 = r2_score(y, y_pred)
    rmsp = rmspe(y, y_pred)
    print(f"           MSE: {mse:12.4f}")
    print(f"Relative error: {re:12.4f}")
    print(f"     R-squared: {r2:12.4f}")
    print(f"         RMSPE: {rmsp:12.4f}")

    #SE abbiamo fornito un nome del nostro modello, possiamo salvare le valutazi
    if(name_for_dictionary != 1):
        models_mse[name_for_dictionary] = mse
        models_re[name_for_dictionary] = re
        models_r2[name_for_dictionary] = r2
        models_rmspe[name_for_dictionary] = rmsp

        #SE abbiamo fornito gli iperparametri li salviamo nel dictionary models_
        if(hyperparameter is not None):
            models_hyperparameter[name_for_dictionary] = hyperparameter

        #SE abbiamo fornito i coefficenti delle feature le salviamo nel dictiona
        if(coeff is not None):
            models_feature_coeff[name_for_dictionary] = coeff
```

Addestriamo il primo modello (quello contenente tutte le feature).

In [29]:
```
model_complete = Pipeline ([
    ("preproc", ColumnTransformer([
        ("numeric", MinMaxScaler(), numerical_var_not_price),
        ("categorical", OneHotEncoder(), categorical_var)
    ])),
    ("regr", Lasso(alpha=1))
    ])
model_complete.fit(x_complete_train, y_complete_train)
```

Out[29]:



Mostriamo le **feature più rilevanti** con i rispettivi **coefficenti**

In [30]:
```
print(pd.Series(model_complete.named_steps["regr"].coef_, model_complete.named_s
print("Feature utilizzate: ",(model_complete.named_steps["regr"].coef_ !=0).sum(
```

```
numeric__carat                46825.083262
categorical__clarity_IF         983.201132
categorical__clarity_VVS1       614.111897
categorical__clarity_VVS2       587.147167
categorical__color_D            457.364503
categorical__color_E            243.412674
categorical__clarity_VS1        210.308099
categorical__color_F            199.134502
categorical__cut_Ideal          117.012519
numeric__y                       -0.000000
categorical__cut_Premium          0.000000
categorical__cut_Very Good        0.000000
categorical__color_G              0.000000
categorical__clarity_VS2        -84.629783
categorical__cut_Good          -159.730673
numeric__table                 -489.765089
categorical__color_H           -492.660063
categorical__clarity_SI1       -702.360420
categorical__cut_Fair          -742.732037
categorical__color_I           -964.249314
categorical__clarity_SI2      -1645.118731
categorical__color_J          -1871.665560
numeric__x                    -1956.125251
categorical__clarity_I1       -4297.157739
numeric__z                    -4383.003806
dtype: float64
Feature utilizzate:  21
```

Lo **score** del modello è il seguente:

In [31]:
```
print_eval(x_complete_val, y_complete_val, model_complete)
```

```
           MSE: 1294466.5914
Relative error:       0.3900
     R-squared:       0.9173
         RMSPE:       0.7516
```

Il modello con tutte le feature, utilizza 21 variabili (sono più di quelle iniziali perchè con l' `OneHotEncoder` viene generata una variabile binaria per ogni possibile valore di ogni variabile categorica).

Ha un errore relativo abbastanza alto (39%) ma anche un R-squared molto alto.

Proviamo ora ad utilizzare **solo le 4 feature** (`carat` , `x` , `y` e `z`) che secondo i grafici avevano una correlazione con `price` .

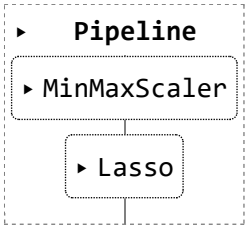
In [32]:
```
x = data.drop(columns=["cut", "color", "clarity", "depth", "table", "price"])
y = data["price"]

x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=1/3, random_st
x_train.columns
```

Out[32]:
```
Index(['carat', 'x', 'y', 'z'], dtype='object')
```

Creiamo un modello semplificato `model_simple` che non tiene conto delle variabili categoriche (quindi non serve l' `OneHotEncoder`), di `depth` e di `table` .

In [33]:
```
model_simple = Pipeline ([
    ("numeric", MinMaxScaler()),
    ("regr", Lasso(alpha=1))
])
model_simple.fit(x_train, y_train)
```

Out[33]:



Questi sono i coefficenti delle feature più rilevanti.

In [34]:
```
print(pd.Series(model_simple.named_steps["regr"].coef_, x.columns).sort_values(a
print("Feature utilizzate: ",(model_simple.named_steps["regr"].coef_ !=0).sum())
```

```
carat    44078.243632
y          434.027298
x           -0.000000
z        -8675.797090
dtype: float64
Feature utilizzate:  3
```

In [35]:
```
print_eval(x_val, y_val, model_simple)
```

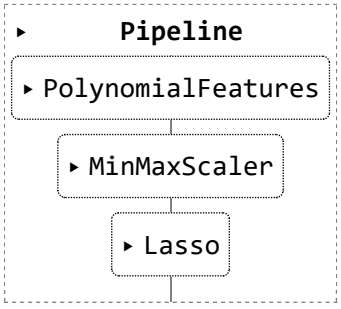
```
           MSE: 2298602.8214
Relative error:       0.2747
     R-squared:       0.8531
         RMSPE:       0.3550
```

L'errore relativo si è ridotto, e anche l'RMSPE è dimunito molto. Anche l'accuratezza è diminuita del 6%, ma ora stiamo utilizzando solo **3 feature** invece che **22**.

Proviamo a utilizzare solo le **4 feature iniziali** e a generare alcune **feature polinomiali derivate**, attraverso l'uso di `PolynomialFeatures` .

In [36]:
```
model_poly = Pipeline ([
    ("poly", PolynomialFeatures(degree=4, include_bias=False)),
    ("numeric", MinMaxScaler()),
    ("regr", Lasso(alpha=1))
])
model_poly.fit(x_train, y_train)
```

Out[36]:



In [37]:
```
print(pd.Series(model_poly.named_steps["regr"].coef_, model_poly.named_steps["po
print("Feature utilizzate: ",(model_poly.named_steps["regr"].coef_ !=0).sum())
```

```
y^3          12898.264779
x y^2        10311.044421
y^4           8287.427208
carat y       5185.540643
x y z         4467.983320
                 ...
z^3              0.000000
carat^3 y       -0.000000
carat^3 x   -10163.493510
x           -11216.052867
carat^4     -14454.527851
Length: 69, dtype: float64
Feature utilizzate:  14
```

Con il `PolynomialFeatures` vengono create delle variabili polinomiali e in questo modo si usano **14 feature**.

In [38]:
```
print_eval(x_val, y_val, model_poly)
```

```
           MSE: 2058049.7634
Relative error:       0.2390
     R-squared:       0.8685
         RMSPE:       0.3078
```

L'errore relativo e l'RMSPE sono scesi ancora, mentre R-squared è aumentato del 5%.

Nella cella seguente sono mostrati i risultati dei 3 modelli per fare un confronto.

In [39]:
```
print("Modello completo con tutte le feature:")
print_eval(x_complete_val, y_complete_val, model_complete)

print("\nModello con solo 4 feature (carat, x, y, z):")
print_eval(x_val, y_val, model_simple)

print("\nModello con 4 feature + feature polinomiali:")
print_eval(x_val, y_val, model_poly)
```

```
Modello completo con tutte le feature:
           MSE: 1294466.5914
Relative error:       0.3900
     R-squared:       0.9173
         RMSPE:       0.7516

Modello con solo 4 feature (carat, x, y, z):
           MSE: 2298602.8214
Relative error:       0.2747
     R-squared:       0.8531
         RMSPE:       0.3550

Modello con 4 feature + feature polinomiali:
           MSE: 2058049.7634
Relative error:       0.2390
     R-squared:       0.8685
         RMSPE:       0.3078
```

Visti i risultati, per il punto 3) useremo le seguenti feature:

- `carat`
- `x`
- `y`
- `z`
- Eventuali feature polinomiali generate con il `PolinomialFeatures`

3) Generazione dei modelli

Utilizzeremo la grid search (`GridSearchCV`) per determinare i migliori iperparametri del modello finale.

Inoltre utilizzeremo la K-Fold Cross Validation (`KFold`) per valutare in modo più preciso l'accuratezza dei modelli.

Quindi creiamo un oggetto `KFold` a 7 fold.

In [40]:
```
kf = KFold(7, shuffle=True, random_state=42)
```

Prima riprendiamo i dati che ci servono

In [41]:
```
x = data.drop(columns=["cut", "color", "clarity", "depth", "table", "price"])
y = data["price"]

x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=1/3, random_st
```

Creiamo una funzione che:

- Addestra i modelli
- Mostra i migliori parametri ottenuti dalla `GridSearch`
- Calcola la media e la deviazione standard degli score ottenuti
- Inserisce i migliori parametri e le feature usate nei dictionary creati in precedenza

In [42]:
```
def grid_test(model, grid, model_name=1):
    gs = GridSearchCV(model, grid, cv=kf, verbose=0)
    gs.fit(x_train, y_train)
    print(gs.best_params_)

    score = cross_val_score(model, x_val, y_val)
    mean = score.mean()
    std = score.std()

    if model_name != 1:
        models_result_gs_kf[model_name] = pd.DataFrame(data={"Media": mean, "Dev
                                                             index=[model_name])

    features_coeff = pd.Series(gs.best_estimator_.named_steps["reg"].coef_,
                               gs.best_estimator_.named_steps["poly"].get_fea
    print_eval(x_val, y_val, gs, model_name, gs.best_params_, features_coeff)
```

Ora procediamo a creare diversi modelli, **uno per ogni algoritmo di regressione**.

LinearRegression

In [43]:
```
model_linear = Pipeline([
    ("scaler", None),
    ("poly", PolynomialFeatures(include_bias=False)),
    ("reg", LinearRegression())
])

grid_linear = {
    "scaler": [None, MinMaxScaler(), StandardScaler()],
    "poly__degree": [1,2,4]
}

%time grid_test(model_linear, grid_linear, "LinearRegression")
models_result_gs_kf["LinearRegression"]
```

```
{'poly__degree': 2, 'scaler': None}
           MSE: 1967546.2722
Relative error:       0.2136
     R-squared:       0.8743
         RMSPE:       0.2832
CPU times: total: 7.67 s
Wall time: 1.93 s
```

Out[43]:

	Media	Deviazione standard
LinearRegression	0.869846	0.00826

Regressione Ridge

In [44]:
```
model_ridge = Pipeline([
    ("scaler", None),
    ("poly", PolynomialFeatures(include_bias=False)),
    ("reg", Ridge())
])

grid_ridge = {
    "scaler": [None, MinMaxScaler(), StandardScaler()],
    "poly__degree": [1,2,4],
    "reg__alpha": [0.01, 0.1, 1]
}

%time grid_test(model_ridge, grid_ridge, "Ridge regression")
models_result_gs_kf["Ridge regression"]
```

```
{'poly__degree': 4, 'reg__alpha': 0.01, 'scaler': MinMaxScaler()}
           MSE: 1902322.5710
Relative error:       0.2105
     R-squared:       0.8784
         RMSPE:       0.2767
CPU times: total: 13.9 s
Wall time: 3.49 s
```

Out[44]:

	Media	Deviazione standard
Ridge regression	0.871162	0.004724

In [45]:
```
warnings.filterwarnings(action='ignore', category=ConvergenceWarning)
```

Regressione Lasso

In [46]:
```
model_lasso = Pipeline([
    ("scaler", None),
    ("poly", PolynomialFeatures(include_bias=False)),
    ("reg", Lasso())
])

grid_lasso = {
    "scaler": [None, MinMaxScaler(), StandardScaler()],
    "poly__degree": [1,2,4],
    "reg__alpha": [0.01, 0.1, 1]
}

%time grid_test(model_lasso, grid_lasso, "Lasso regression")

models_result_gs_kf["Lasso regression"]
```

```
{'poly__degree': 4, 'reg__alpha': 0.01, 'scaler': MinMaxScaler()}
           MSE: 1930497.0935
Relative error:       0.2221
     R-squared:       0.8766
         RMSPE:       0.2944
CPU times: total: 4min 2s
Wall time: 1min
```

Out[46]:

	Media	Deviazione standard
Lasso regression	0.868718	0.005115

Elastic Net

In [47]:
```
model_elastic = Pipeline([
    ("scaler", None),
    ("poly", PolynomialFeatures(include_bias=False)),
    ("reg", ElasticNet())
])

grid_elastic = {
    "scaler": [None, MinMaxScaler(), StandardScaler()],
    "poly__degree": [1,2,4],
    "reg__alpha": [0.1, 1, 10],
    "reg__l1_ratio": [0.1, 0.25, 0.5]
}

%time grid_test(model_elastic, grid_elastic, "Elastic net")

models_result_gs_kf["Lasso regression"]
```

```
{'poly__degree': 4, 'reg__alpha': 0.1, 'reg__l1_ratio': 0.5, 'scaler': StandardS
caler()}
           MSE: 1956882.2280
Relative error:       0.2203
     R-squared:       0.8750
         RMSPE:       0.2937
CPU times: total: 8min 58s
Wall time: 2min 15s
```

Out[47]:

	Media	Deviazione standard
Lasso regression	0.868718	0.005115

I risultati ottenuti fin'ora sono molto simili, la combinazione migliore è quella che usa la **Regression Ridge** con i parametri:

- Grado di `PolynomialFeatures` = 4,
- Grado alpha = 0.01
- MinMaxScaler()

I risultati sono: MSE: 1902322.5710
Relative error: 0.2105
R-squared: 0.8784
RMSPE: 0.2767

Decision tree

In [48]:
```
model_tree = DecisionTreeRegressor(max_depth=10, random_state=42)
%time model_tree.fit(x_train, y_train)
```

```
CPU times: total: 250 ms
Wall time: 53.3 ms
```

Out[48]:
```
▾                DecisionTreeRegressor

DecisionTreeRegressor(max_depth=10, random_state=42)
```

Nel `DecisionTreeRegressor` i coefficenti delle feature sono le seguenti:

In [49]:
```
print(pd.Series(model_tree.feature_importances_, x_train.columns).sort_values(as
      "\nDepth: ", model_tree.get_depth(),
      "\nLeaves: ", model_tree.get_n_leaves())
```

```
carat    0.730579
y        0.255306
z        0.009114
x        0.005001
dtype: float64
Depth:  10
Leaves:  800
```

Per quanto riguarde le prestazioni, sono praticamente uguali a quelle della **Regressione Ridge**

In [50]:
```
print_eval(x_val, y_val, model_tree, "Decision tree")
models_feature_coeff["Decision tree"] = pd.Series(model_tree.feature_importances
models_hyperparameter["Decision tree"] = model_tree.get_params()

score = cross_val_score(model_tree, x_val, y_val, cv=kf)
mean = score.mean()
std = score.std()
models_result_gs_kf["Decision tree"] = pd.DataFrame(data={"Media":mean, "Deviazi
models_result_gs_kf["Decision tree"]
```

```
           MSE: 2019954.7575
Relative error:       0.2026
     R-squared:       0.8709
         RMSPE:       0.2711
```

Out[50]:

	Media	Deviazione standard
Decision tree	0.862788	0.006625

Random forest

In [51]:
```
model_random_forest = RandomForestRegressor(max_samples=0.1, max_features="sqrt"
%time model_random_forest.fit(x_train, y_train)
```

```
CPU times: total: 7.5 s
Wall time: 1.39 s
```

Out[51]:
```
RandomForestRegressor(max_features='sqrt', max_samples=0.1, n_estimator
s=600,
                      n_jobs=-1)
```

In [52]:
```
print_eval(x_val, y_val, model_random_forest, "Random forest")
models_feature_coeff["Random forest"] = pd.Series(model_random_forest.feature_im
models_hyperparameter["Random forest"] = model_random_forest.get_params()

mean = cross_val_score(model_random_forest, x_val, y_val, cv=kf).mean()
std = cross_val_score(model_random_forest, x_val, y_val, cv=kf).std()
models_result_gs_kf["Random forest"] = pd.DataFrame(data={"Media":mean, "Deviazi
models_result_gs_kf["Random forest"]
```

```
           MSE: 1828117.8697
Relative error:       0.1989
     R-squared:       0.8832
         RMSPE:       0.2643
```

Out[52]:

	Media	Deviazione standard
Random forest	0.8809	0.004383

Con il `RandomForestRegressor` **abbiamo avuto i migliori risultati fin'ora.**

Nella cella presente vengono mostrati i coefficenti delle feature usate.

In [53]:
```
pd.Series(model_random_forest.feature_importances_, model_random_forest.feature_
```

Out[53]:
```
y        0.399008
carat    0.320091
x        0.212491
z        0.068410
dtype: float64
```

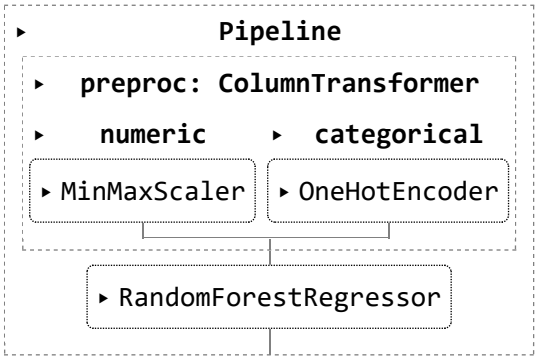
Random forest (usando tutte le variabili, anche quelle categoriche)

Proviamo ad usare il `RandomForestRegressor` con tutte le variabili (anche quelle categoriche).

In [54]:
```
model_random_forest_complete = Pipeline ([
    ("preproc", ColumnTransformer([
        ("numeric", MinMaxScaler(), numerical_var_not_price),
        ("categorical", OneHotEncoder(), categorical_var)
    ])),
    ("reg", RandomForestRegressor(max_samples=0.1, max_features="sqrt", n_estima
    ])
%time model_random_forest_complete.fit(x_complete_train, y_complete_train)
```

```
CPU times: total: 7.31 s
Wall time: 1.57 s
```

Out[54]:



In [55]:
```
print_eval(x_complete_val, y_complete_val, model_random_forest_complete, "Random
models_feature_coeff["Random forest complete"] = pd.Series(model_random_forest_c
                                                           model_random_forest_co
models_hyperparameter["Random forest complete"] = model_random_forest_complete.n

mean = cross_val_score(model_random_forest_complete, x_complete_val, y_complete_
std = cross_val_score(model_random_forest_complete, x_complete_val, y_complete_v
models_result_gs_kf["Random forest complete"] = pd.DataFrame(data={"Media":mean,
models_result_gs_kf["Random forest complete"]
```

```
           MSE:  497621.0423
Relative error:       0.0945
     R-squared:       0.9682
         RMSPE:       0.1324
```

Out[55]:

	Media	Deviazione standard
Random forest complete	0.957463	0.002181

I risultati ottenuti con il `RandomForestRegressor`, **utilizzando tutte le feature** (anche quelle categoriche), sono i **migliori in assoluto**.

Nella cella seguente sono mostrati i coefficenti.

In [56]:
```
models_feature_coeff["Random forest complete"]
```

Out[56]:
```
numeric__y                    0.271360
numeric__carat                0.228147
numeric__x                    0.213402
numeric__z                    0.195016
categorical__clarity_SI2      0.012893
numeric__table                0.008271
categorical__clarity_I1       0.008102
categorical__clarity_VVS2     0.006876
categorical__color_J          0.006737
categorical__clarity_SI1      0.005670
categorical__clarity_IF       0.005303
categorical__color_I          0.004712
categorical__clarity_VVS1     0.004654
categorical__clarity_VS2      0.003452
categorical__clarity_VS1      0.003375
categorical__color_F          0.003053
categorical__color_G          0.003043
categorical__color_D          0.002828
categorical__color_E          0.002797
categorical__color_H          0.002755
categorical__cut_Ideal        0.002372
categorical__cut_Premium      0.001772
categorical__cut_Fair         0.001354
categorical__cut_Very Good    0.001069
categorical__cut_Good         0.000988
dtype: float64
```

CatBoost

Infine, utilizziamo `CatBoost` che permette di usare direttamente le variabli categoriche senza dover'usare l' `OneHotEncoder` .

In [57]:
```
#pip install catboost
from catboost import CatBoostRegressor
```

```
C:\Users\simon\anaconda3\lib\site-packages\xgboost\compat.py:36: FutureWarning:
pandas.Int64Index is deprecated and will be removed from pandas in a future vers
ion. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
```

In [58]:
```
catbm = CatBoostRegressor(n_estimators=500, cat_features=["cut", "color", "clari
catbm.fit(x_complete_train, y_complete_train, verbose=False) #verbose=False nasc
```

Out[58]:
```
<catboost.core.CatBoostRegressor at 0x1a738157970>
```

I punteggi di questo algoritmo sono migliori anche del `RandomForest` con tutte le variabili.

In [59]:
```
print_eval(x_complete_val, y_complete_val, catbm, "Catboost")
```

```
           MSE:  287504.0852
Relative error:       0.0783
     R-squared:       0.9816
         RMSPE:       0.1055
```

Calcolo i coefficenti delle feature, e lo score, con la sua deviazione standard.

In [66]:
```
models_feature_coeff["Catboost"]= pd.Series(catbm.get_feature_importance(), x_co
score = cross_val_score(catbm, x_complete_val, y_complete_val, cv=kf)
mean = score.mean()
std = score.std()
models_result_gs_kf["Catboost"] = pd.DataFrame(data={"Media":mean, "Deviazione s
```

Questi sono i coefficenti delle feature usate, la media dell'R^2 in k cross fold validation e la sua deviazione standard.

In [67]:
```
print(models_feature_coeff["Catboost"])
models_result_gs_kf["Catboost"]
```

```
y          27.143682
carat      18.367820
clarity    17.273151
x          14.247607
color      11.669575
z          10.322363
cut         0.608405
table       0.367397
dtype: float64
```

Out[67]:

	Media	Deviazione standard
Catboost	0.980266	0.001779

4) Valutazione dei modelli di regressione

Utilizziamo le strutture dati (dictionary) create in precedenza per mostrare i dettagli sugli score, sugli iperparametri e sulle feature dei vari algoritmi utilizzati per valutarli.

In [68]:
```
def print_all_about_model(model, coeff=False):
    print("\n\t",model, "\nMse:\t\t", f"{models_mse.get(model):.4f}",
          "\nRelative error: ", f"{models_re.get(model):.4f}",
          "\nR-squared:\t", f"{models_r2.get(model):.4f}")
    if key in models_result_gs_kf:
        print("R-squared medio:", f"{models_result_gs_kf[model].loc[model,'Media
          "\nDeviazione standard degli score:", f"{models_result_gs_kf[model].loc
    print("RMSPE:\t\t", f"{models_rmspe.get(model):.4f}",
          "\nIperparametri: ", models_hyperparameter.get(model))
    if coeff== True:
        print("Coefficienti:\n", models_feature_coeff[model])
    print("\n")
```

In [69]:
```
for key in models_mse.keys():
    print_all_about_model(key)
```

```
         LinearRegression
Mse:             1967546.2722
Relative error:  0.2136
R-squared:       0.8743
R-squared medio: 0.8698
Deviazione standard degli score: 0.0083
RMSPE:           0.2832
Iperparametri:  {'poly__degree': 2, 'scaler': None}



         Ridge regression
Mse:             1902322.5710
Relative error:  0.2105
R-squared:       0.8784
R-squared medio: 0.8712
Deviazione standard degli score: 0.0047
RMSPE:           0.2767
Iperparametri:  {'poly__degree': 4, 'reg__alpha': 0.01, 'scaler': MinMaxScaler
()}



         Lasso regression
Mse:             1930497.0935
Relative error:  0.2221
R-squared:       0.8766
R-squared medio: 0.8687
Deviazione standard degli score: 0.0051
RMSPE:           0.2944
Iperparametri:  {'poly__degree': 4, 'reg__alpha': 0.01, 'scaler': MinMaxScaler
()}



         Elastic net
Mse:             1956882.2280
Relative error:  0.2203
R-squared:       0.8750
R-squared medio: 0.8521
Deviazione standard degli score: 0.0062
RMSPE:           0.2937
Iperparametri:  {'poly__degree': 4, 'reg__alpha': 0.1, 'reg__l1_ratio': 0.5, 'sc
aler': StandardScaler()}



         Decision tree
Mse:             2019954.7575
Relative error:  0.2026
R-squared:       0.8709
R-squared medio: 0.8628
Deviazione standard degli score: 0.0066
RMSPE:           0.2711
Iperparametri:  {'ccp_alpha': 0.0, 'criterion': 'squared_error', 'max_depth': 1
0, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, '
min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, '
random_state': 42, 'splitter': 'best'}



         Random forest
Mse:             1828117.8697
```

```
Relative error:  0.1989
R-squared:       0.8832
R-squared medio: 0.8809
Deviazione standard degli score: 0.0044
RMSPE:           0.2643
Iperparametri:  {'bootstrap': True, 'ccp_alpha': 0.0, 'criterion': 'squared_erro
r', 'max_depth': None, 'max_features': 'sqrt', 'max_leaf_nodes': None, 'max_samp
les': 0.1, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_spl
it': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 600, 'n_jobs': -1, 'oob
_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
```

```
         Random forest complete
Mse:             497621.0423
Relative error:  0.0945
R-squared:       0.9682
R-squared medio: 0.9575
Deviazione standard degli score: 0.0022
RMSPE:           0.1324
Iperparametri:  {'bootstrap': True, 'ccp_alpha': 0.0, 'criterion': 'squared_erro
r', 'max_depth': None, 'max_features': 'sqrt', 'max_leaf_nodes': None, 'max_samp
les': 0.1, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_spl
it': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 600, 'n_jobs': -1, 'oob
_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
```

```
         Catboost
Mse:             287504.0852
Relative error:  0.0783
R-squared:       0.9816
R-squared medio: 0.9803
Deviazione standard degli score: 0.0018
RMSPE:           0.1055
Iperparametri:  None
```

1. Tra i modelli classici basati su equazioni, nonostante non ci fossero grosse differenze, tenendo conto dell' R^2, dell'**errore relativo** e dell' **RMSPE**, il modello migliore è quello che utilizza la **Ridge regression**.

1. Se invece utilizziamo gli alberi di regressione, il modello migliore è il **Random forest** (quello con tutte le feature), che ha un R^2 superiore al 95%, e un errore relativo inferiore al 10%.

1. Il miglior modello di regressione è il **CatBoost** (leggermente migliore rispetto a **Random forest**, basato su *gradient boosting* che supporta anche le feature categoriche con
   - R^2 medio = **98,03%**
   - *Errore relativo* = **7,8%**
   - *RMSPE* = **10,5%**

5) Conclusioni

Prendendo in considerazione il `Random forest` : Le feature più importanti sono quelle riferite alla **dimensione** del diamante (`x` , `y` e `z`) e anche il **peso** del diamante stesso `carat` . Queste variabili sono quelle che influenzano POSITIVAMENTE e in modo maggiore il `price` .

In [70]:
```
models_feature_coeff["Random forest complete"]
```

Out[70]:
```
numeric__y                    0.271360
numeric__carat                0.228147
numeric__x                    0.213402
numeric__z                    0.195016
categorical__clarity_SI2      0.012893
numeric__table                0.008271
categorical__clarity_I1       0.008102
categorical__clarity_VVS2     0.006876
categorical__color_J          0.006737
categorical__clarity_SI1      0.005670
categorical__clarity_IF       0.005303
categorical__color_I          0.004712
categorical__clarity_VVS1     0.004654
categorical__clarity_VS2      0.003452
categorical__clarity_VS1      0.003375
categorical__color_F          0.003053
categorical__color_G          0.003043
categorical__color_D          0.002828
categorical__color_E          0.002797
categorical__color_H          0.002755
categorical__cut_Ideal        0.002372
categorical__cut_Premium      0.001772
categorical__cut_Fair         0.001354
categorical__cut_Very Good    0.001069
categorical__cut_Good         0.000988
dtype: float64
```

Se prendiamo in considerazione `Catboost` , le variabili che influenzano maggiormente e POSITIVAMENTE il `price` sono sempre `x` , `y` , `z` , `carat` , ma anche la purezza del diamante `clarity` e il colore `color` .

Non ci sono feature correlate NEGATIVAMENTE

In [71]:
```
models_feature_coeff["Catboost"]
```

Out[71]:
```
y          27.143682
carat      18.367820
clarity    17.273151
x          14.247607
color      11.669575
z          10.322363
cut         0.608405
table       0.367397
dtype: float64
```