

TRABAJO PRACTICO

PARADIGMAS DE PROGRAMACION

Lorenzo Martinez

Informacion de los Datos

Para empezar hablemos de las distintas columnas que tiene el dataframe y que representa cada uno para los diamantes...

Antes que nada . La fuente del dataframe es

<https://www.kaggle.com/datasets/shivam2503/diamonds>

Descripción:

Conjunto de datos que contiene los precios y otros atributos de casi 54.000 diamantes. Las variables son las siguientes:

Formato :

Un dataframe con 53940 filas y 10 variables:

Price (precio):

precio en dólares estadounidenses (\\$326-\\$18.823)

Carat (quilate):

peso del diamante (0,2-5,01)

Cut (talla):

calidad de la talla (Fair/regular, Good/buena, Very Good/muy buena, Premium/superior, Ideal)

Color:

color del diamante, de D (mejor) a J (peor)

Clarity (claridad)

medida de la claridad del diamante (I1 (peor), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (mejor))

X:

longitud en mm (0-10,74)

Y:

anchura en mm (0-58,9)

Z:

profundidad en mm (0-31,8)

Depth (profundidad)

porcentaje de profundidad total = z / media(x, y) = 2 * z / (x + y) (43-79)

Table (tabla)

anchura de la parte superior del diamante en relación con el punto más ancho (43-95)

Visualicemos el dataframe

Detalles estadísticos del conjunto de datos: (Recopilación de datos)

```
df
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
...
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64

53940 rows × 10 columns

detalles estadísticos del conjunto de datos:

```
df.describe()
```

	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	0.000000
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	2.910000
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	3.530000
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	4.040000
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	31.800000

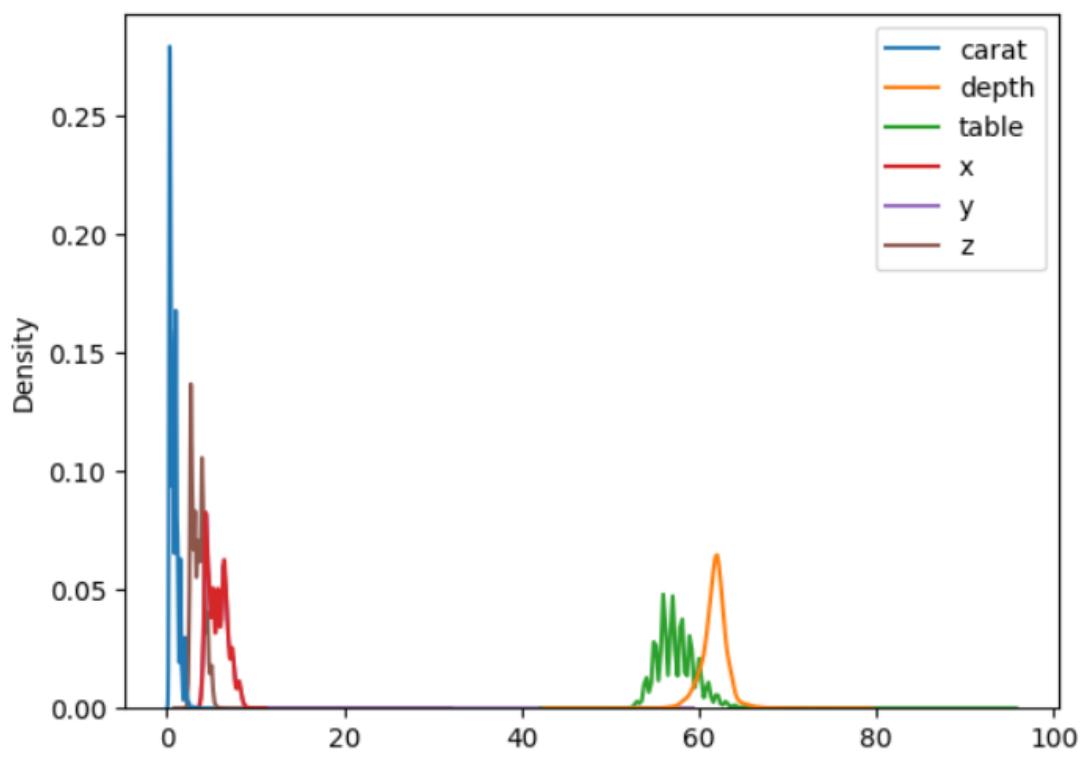
Antes de continuar observemos el grafico de densidad

```
sns.kdeplot(data=df[['carat', 'depth', 'table', 'x', 'y', 'z']])
```

```
# Mostrar el gráfico
```

```
plt.show()
```

✓ 1.4s



Ahora se continua revisando que tipos de datos se guardan en cada columna y de paso comprobamos que todas las filas contienen datos no nulos

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
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: 4.1+ MB
```

```
df.isnull().any()
```

[5]

```
...   carat      False
      cut        False
      color      False
      clarity    False
      depth      False
      table      False
      price      False
      x          False
      y          False
      z          False
dtype: bool
```

No tenemos NANS /nulos en el dataset ,pero podriamos tener 0 que podrian ser como NAns camuflados

Revisemos si existe campos iguales a 0 en algunas columnas

```
cols = df[df == 0].count(axis=0)  
cols[cols > 0]
```

```
5]
```

```
x      8  
y      7  
z     20  
dtype: int64
```

X y z tienen valores igual a 0. Un diamante es un objeto tridimensional , de ninguna forma alguna una coordenada puede valer 0 en un diamante . Algo hay que hacer efectivamente. Pero antes observemos aquellos diamantes que tienen 0 en sus coordenadas.

```
df.loc[(df['x'] == 0) | (df['y'] == 0) | (df['z'] == 0)]
```

	carat	cut	color	clarity	depth	table	price	x	y	z
2207	1.00	Premium	G	SI2	59.1	59.0	3142	6.55	6.48	0.0
2314	1.01	Premium	H	I1	58.1	59.0	3167	6.66	6.60	0.0
4791	1.10	Premium	G	SI2	63.0	59.0	3696	6.50	6.47	0.0
5471	1.01	Premium	F	SI2	59.2	58.0	3837	6.50	6.47	0.0
10167	1.50	Good	G	I1	64.0	61.0	4731	7.15	7.04	0.0
11182	1.07	Ideal	F	SI2	61.6	56.0	4954	0.00	6.62	0.0
11963	1.00	Very Good	H	VS2	63.3	53.0	5139	0.00	0.00	0.0
13601	1.15	Ideal	G	VS2	59.2	56.0	5564	6.88	6.83	0.0
15951	1.14	Fair	G	VS1	57.5	67.0	6381	0.00	0.00	0.0
24394	2.18	Premium	H	SI2	59.4	61.0	12631	8.49	8.45	0.0
24520	1.56	Ideal	G	VS2	62.2	54.0	12800	0.00	0.00	0.0
26123	2.25	Premium	I	SI1	61.3	58.0	15397	8.52	8.42	0.0
26243	1.20	Premium	D	VVS1	62.1	59.0	15686	0.00	0.00	0.0
27112	2.20	Premium	H	SI1	61.2	59.0	17265	8.42	8.37	0.0
27429	2.25	Premium	H	SI2	62.8	59.0	18034	0.00	0.00	0.0
27503	2.02	Premium	H	VS2	62.7	53.0	18207	8.02	7.95	0.0
27739	2.80	Good	G	SI2	63.8	58.0	18788	8.90	8.85	0.0
49556	0.71	Good	F	SI2	64.1	60.0	2130	0.00	0.00	0.0
49557	0.71	Good	F	SI2	64.1	60.0	2130	0.00	0.00	0.0
51506	1.12	Premium	G	I1	60.4	59.0	2383	6.71	6.67	0.0

Se observa incluso que existen casos en donde las 3 coordenadas son 0!! Por suerte solo son 20 casos con anomalías en sus coordenadas aun así que se va aplicar alguna estrategia para contrarrestar esto

```
df.replace(0, np.nan, inplace=True)  
df.isnull().sum()
```

✓ 0.0s

```
carat      0  
cut        0  
color      0  
clarity    0  
depth      0  
table      0  
price      0  
x           8  
y           7  
z          20  
dtype: int64
```

Como solo X y z tiene 0 ,podemos indicar directamente que se remplacen todos los 0 por un Nan

```
df.replace(0, np.nan, inplace=True)
mean = df['x'].mean() # imputando x con media
df['x'].fillna(mean, inplace =True)
mode = df['y'].mode()
df['y'].fillna(mode, inplace =True)
mode = df['z'].mode()
df['z'].fillna(mode, inplace =True)

df.isnull().sum()

6] ✓ 0.0s

carat      0
cut        0
color      0
clarity    0
depth      0
table      0
price      0
x          0
y          0
z          0
dtype: int64
```

Imputamos todos los nan por el valor medio de cada columna y finalmente comprobamos que ya no estan los nan. De paso comprobamos que ya no existen columnas con x y z con 0 en sus filas

```
cols = df[df == 0].count(axis=0)
cols[cols > 0]

✓ 0.0s

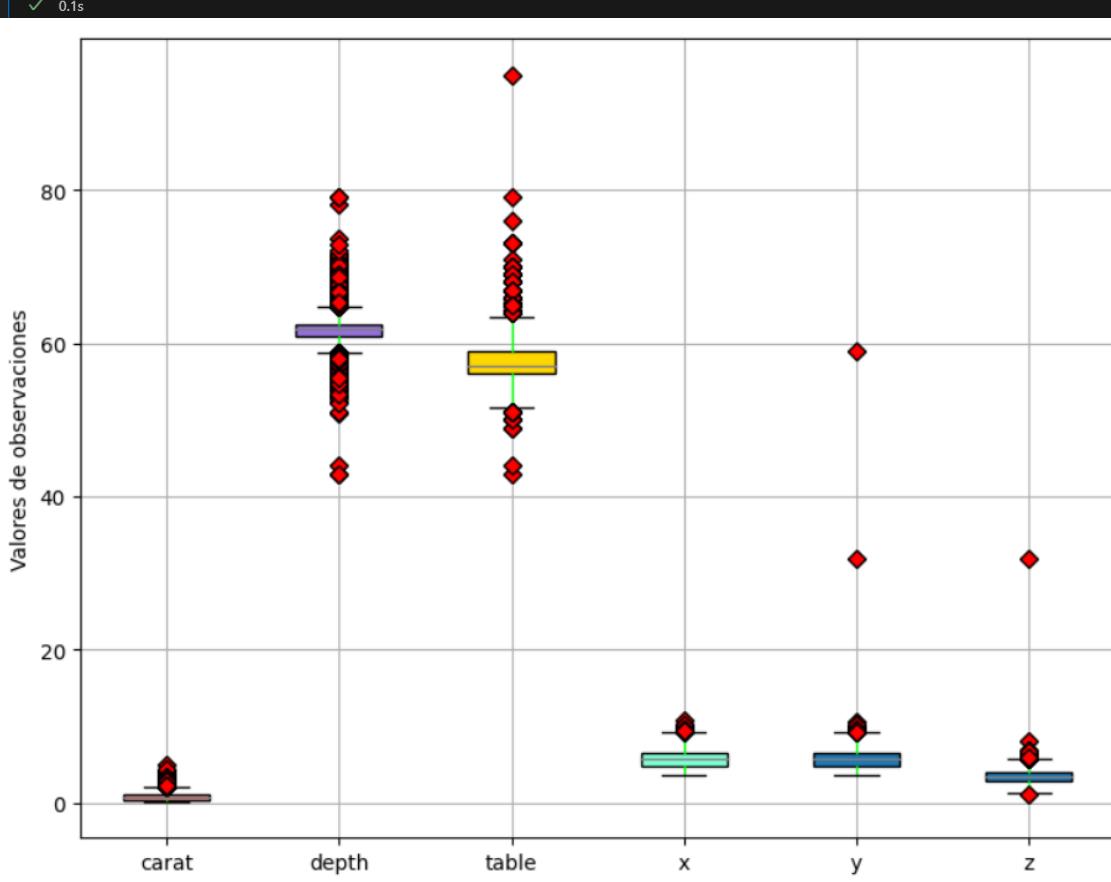
Series([], dtype: int64)
```

A continuacion analicemos los Outliers que tiene el dataframe

Obtenemos primero las columnas numéricas

```
columnas_numericas = df.select_dtypes(include=['int', 'float']).columns.tolist()
columnas_numericas.remove('price')
columnas_numericas
] ✓ 0.0s
['carat', 'depth', 'table', 'x', 'y', 'z']

dataframe = df.loc[:, ['carat', 'depth', 'table', 'x', 'y', 'z']]
# Primer acercamiento a la detección de outliers mediante boxplot utilizando todas las columnas numéricas...
fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(9, 7))
labels = ['carat', 'depth', 'table', 'x', 'y', 'z']
red_diamond = dict(markerfacecolor='r', marker='D')
bplot = ax1.boxplot(dataframe.select_dtypes(include=["int16", "int32", "int64", "float16", "float32", "float64"]),
                     vert=True,
                     patch_artist=True,
                     labels=labels,
                     capprops=dict(color="black"),
                     medianprops=dict(color="grey"),
                     whiskerprops=dict(color="lime"),
                     flierprops=red_diamond)
colors = ['lightcoral', 'mediumpurple', 'gold', 'aquamarine']
for patch, color in zip(bplot['boxes'], colors):
    patch.set_facecolor(color)
for ax in [ax1]:
    ax.yaxis.grid(True)
    ax.xaxis.grid(True)
    ax.set_ylabel('Valores de observaciones')
plt.show()
```

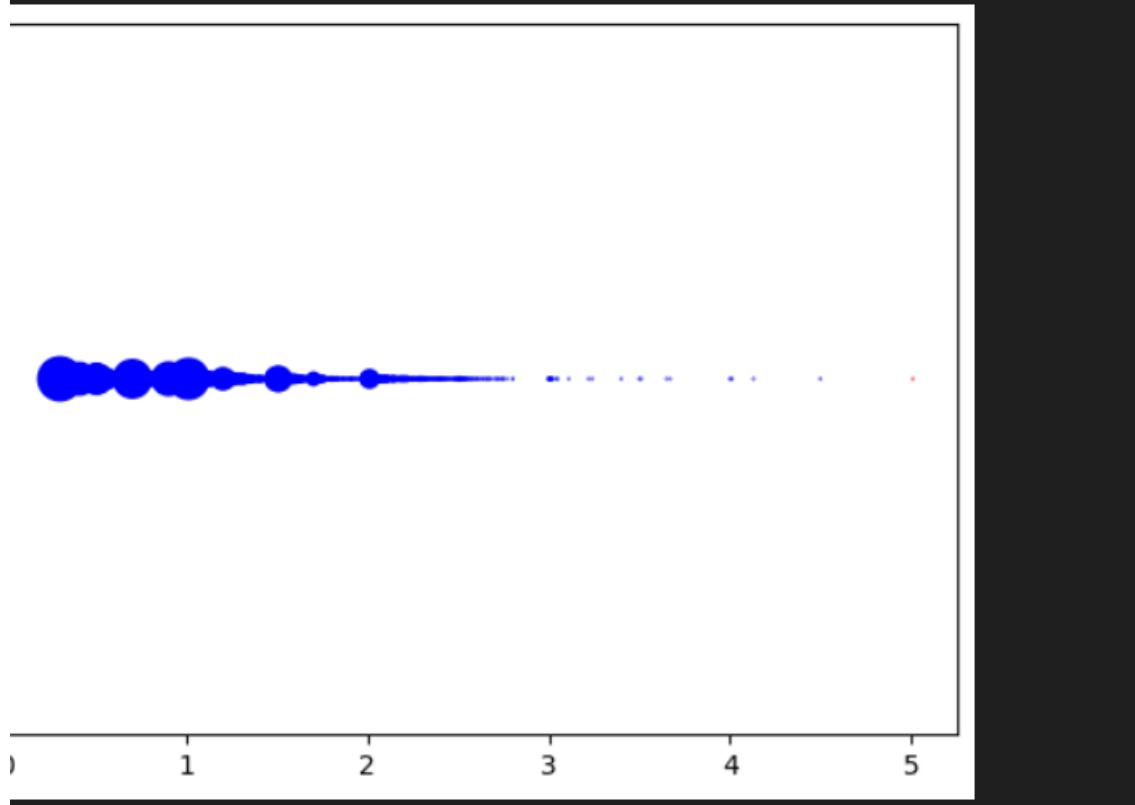


los rombos rojos representan los valores atípicos (outliers) en los datos. A simple vista pareciera que por cada columna contamos con varios valores atípicos no? Vamos a seguir con en analisis uno por uno para determinar que hacer con ellos

Otro grafico mas. En este caso solo analizamos para la columna carat

```
def outliersVisualPuntos(columna) :  
    ... variable, counts = np.unique(df[columna], return_counts=True)  
    ... sizes = counts*0.10  
    ... colors = ['blue']*len(variable)  
    ... colors[-1] = 'red'  
    ... plt.axhline(1, color='k', linestyle=' ')  
    ... plt.scatter(variable, np.ones(len(variable)), s=sizes, color=colors)  
    ... plt.yticks([])  
    ... plt.show()  
  
outliersVisualPuntos('carat')
```

0.0s



Veamos la Puntuación Z

```

# Puntuación Z = (punto_datos - media) / desviación estándar
# ayuda a comprender qué tan lejos está el punto de datos de la media. Y después de configurar un valor
# de umbral, se pueden utilizar los valores de puntuación z de los puntos de datos para definir los valores atípicos

def puntuacionZdeUnaVariableInd(variable):
    z = np.abs(stats.zscore(df[variable]))
    return z

zCarat = puntuacionZdeUnaVariableInd('carat')
zCarat

0      1.198168
1      1.240361
2      1.198168
3      1.071587
4      1.029394
...
53935   0.164427
53936   0.164427
53937   0.206621
53938   0.130927
53939   0.101137
Name: carat, Length: 53940, dtype: float64

```

Contemos el numero de outliers en cada columna.

```

# para definir un valor de umbral atípico, se elige que generalmente es 3.0.
def cantidadDeValoresAtipicosEnLaColumna(z,nombreColumna) :
    threshold = 3
    a = np.where(z > threshold)
    cantidad_elementos = len(a[0])
    print("Cantidad de valores atípicos en la columna "+nombreColumna+" :", cantidad_elementos)
    return cantidad_elementos

cantidadDeValoresAtipicosEnLaColumna(zCarat,'carat')

7] Cantidad de valores atípicos en la columna carat : 439
439

```

Veamos los valores atípicos de todas las variables independientes

```

lista = ['carat', 'depth', 'table', 'x', 'y', 'z']
cantidadDeOutliers = 0
for element in lista :
    z= puntuacionZdeUnaVariableInd(element)
    cantidadDeValoresAtipicos = cantidadDeValoresAtipicosEnLaColumna(z,element)
    cantidadDeOutliers= cantidadDeOutliers + cantidadDeValoresAtipicos
    print("cantidad total de valores atípicos en el df :" , cantidadDeOutliers)

8] Cantidad de valores atípicos en la columna carat : 439
Cantidad de valores atípicos en la columna depth : 685
Cantidad de valores atípicos en la columna table : 336
Cantidad de valores atípicos en la columna x : 35
Cantidad de valores atípicos en la columna y : 27
Cantidad de valores atípicos en la columna z : 55
cantidad total de valores atípicos en el df : 1577

```

Para hacer el siguiente análisis tenemos que saber de antemano lo siguiente ;
 1ero : ¿cuantos registros cuenta nuestros dataframe? (Recordarlo no viene mal)

```
num_filas = df.shape[0] (variable) num_filas  
print("Número de filas:", num_filas)
```

Número de filas: 53940

Lo almacenamos en una variable porque la vamos a usar mas adelante

2do : ahora veamos la cantidad de variables independientes que consideramos para hacer el analisis de datos outliers

```
lista = ['carat', 'depth', 'table', 'x', 'y', 'z']
```

Aclaracion : la columna price no se tiene en cuenta porque es la que vamos a usar para predecir. Por eso no esta en la lista. El resto son columnas/variables categoricas que directamente no se tienen en cuenta para el analisis

```
len(lista)
```

6

3ero :

¿Cuantos datos hay en el df en total teniendo en cuenta solo la lista ['carat', 'depth', 'table', 'x', 'y', 'z']? Osea la sumatoria de todos los datos que tiene el dataframe en las columnas 'carat', 'depth', 'table', 'x', 'y', 'z' . Para saberlo hacemos una cuenta sencilla (numero de filas * cantidad de filas que consideramos en este caso es 6)

```
numeroDeDatos = num_filas*len(lista)  
numeroDeDatos
```

4to:

Lo que sigue ahora es determinar el porcentaje sobre el total de datos que son outliers

```
porcentajeDeOutliers= (cantidadDeOutliers*100)/numeroDeDatos  
print("el porcentaje de outliers es del ",porcentajeDeOutliers,"%")
```

el porcentaje de outliers es del 0.48726980595723646 %

El porcentaje de valores atípicos (outliers) en las columnas ['carat', 'depth', 'table', 'x', 'y', 'z'] es solo del 0.49%. Esto significa que es posible que estos valores atípicos no tengan un impacto significativo en el rendimiento de nuestro modelo. Por lo tanto, se ha decidido dejarlos como están y no realizar ninguna acción específica

sobre ellos . Sin embargo, es importante destacar que previamente se ha tenido que reemplazar los valores igual a cero en las columnas 'x' y 'z', lo que claramente nos indica o nos da cierta sospecha de que existen anomalías en el dataframe. Esta podría ser una razón válida para considerar la eliminación de filas que contengan valores atípicos en algunas de las columnas, ya que podrían ser sospechosos de ser anomalías. No obstante, dado que los valores atípicos solo representan el 0.49% de todos los datos en las columnas numéricas, hemos decidido dejarlos como están debido a la duda sobre su impacto en el modelo.

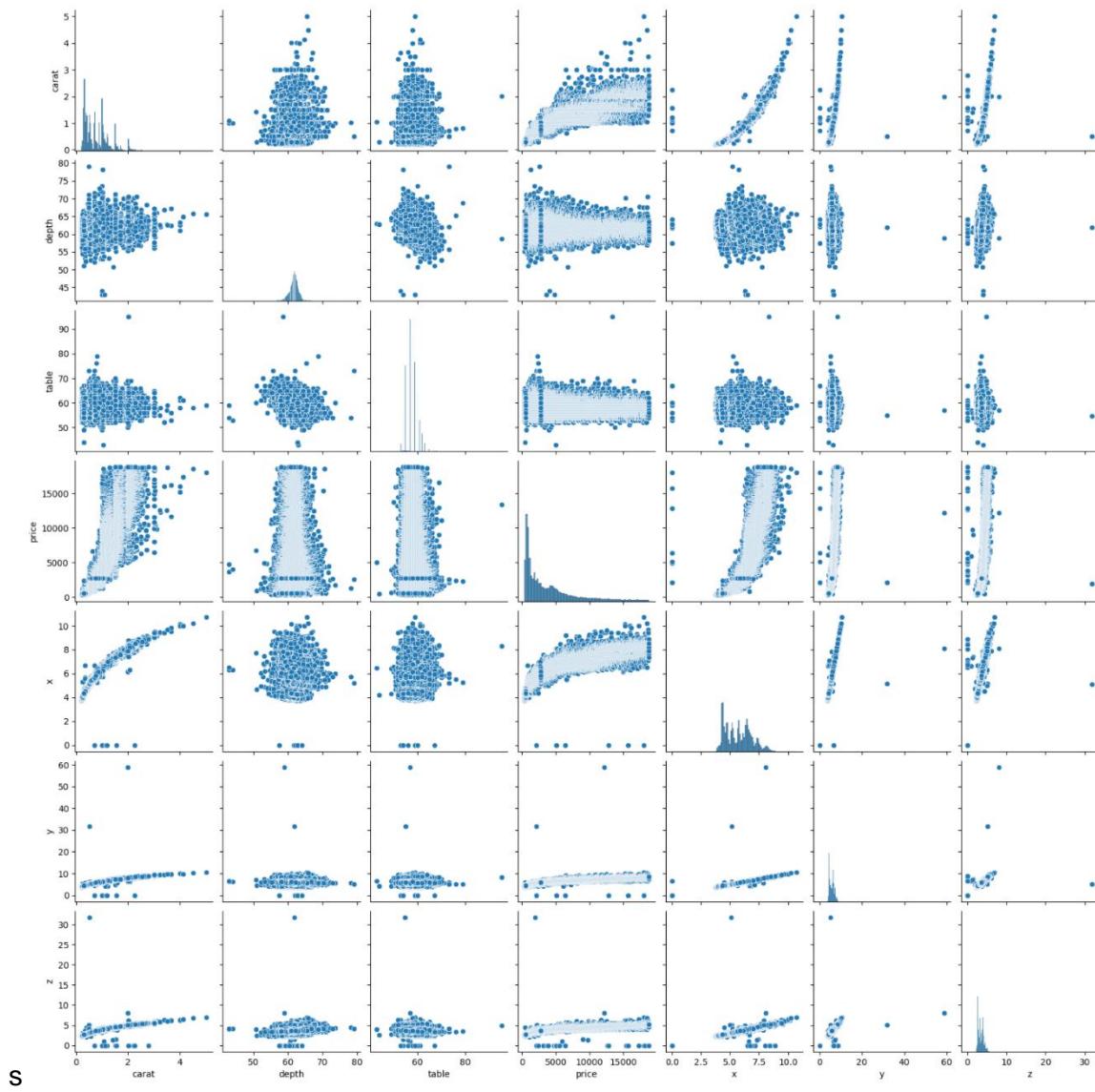
Un poco de análisis bivariado/multivariado...

Cuando comparamos pares de variables, estamos tratando de encontrar conexiones o relaciones entre ellas. Esta información es muy importante porque nos ayuda a identificar cuáles variables pueden ser las mejores para predecir en nuestro modelo. Además, nos permite detectar si existen relaciones no lineales entre las variables, es decir, si la relación entre ellas no sigue una línea recta.

Observemoslo con una imagen

```
#gráfico de pares: es una matriz de diagramas de dispersión que permite comprender la relación por pares  
# entre diferentes variables en un conjunto de datos.
```

```
pairplot = sns.pairplot(df)  
  
#Mostrar el gráfico  
pairplot.fig.show()
```



en el gráfico de dispersión generado por `sns.pairplot(df)`, se puede ver una relación lineal entre las variables "x" y "y" con respecto a la variable "price", esto significa que existe una asociación lineal entre esas variables. En otras palabras, a medida que los valores de "x" aumentan, los valores de "y" también tienden a aumentar o disminuir de manera lineal, y esto a su vez tiene un impacto en los valores de "price".

Este tipo de relación lineal entre "x", "y" y "price" puede ser útil en el análisis de datos, ya que indica que "x" y "y" pueden ser buenos predictores de "price" en un modelo de regresión lineal. Esto significa que se puede utilizar una ecuación lineal para predecir los valores de "price" en función de los valores de "x" y "y".

Matriz de correlacion

Cuando queremos ver cómo se relacionan entre sí las diferentes variables en nuestros datos, utilizamos algo llamado correlación. La correlación nos muestra qué tan fuerte es la relación entre dos variables. Para representar estas correlaciones se debe observar los cuadros o cajas de diferentes colores. Cuanto más intenso sea el color del cuadro, mayor será la magnitud de la correlación.

También tenemos números dentro de los cuadros. Estos números nos indican qué tan cerca está la correlación de 1. Si el número es positivo, significa que hay una relación positiva entre las variables. Si es negativo, significa que hay una relación negativa.

Cuando la correlación es igual a 1 o -1, eso significa que la relación entre las variables es perfecta.

```
corr = df.corr()
plt.subplots(figsize=(12,8))
sns.heatmap(corr, xticklabels=corr.columns, yticklabels=corr.columns, annot=True, fmt='%.0%', cmap=sns.diverging_palette(240, 10, as_cmap=True))
```



Se observa que las variables "table" y "depth" tienen una correlación baja con la variable dependiente "price" y podríamos considerar que su capacidad predictiva es limitada o pobre según los numeros que se ven. Entonces sería factible la eliminación de estas variables del dataframe. Al eliminar estas variables, nos estaríamos deshaciendo de aquella información que se considera que no aporta de manera significativa a la predicción del precio. Aun así hay que tener en cuenta algo ,el precio del diamante lo determina sus características y esos 2 atributos por mas baja correlación son determinantes para el precio de un diamante, solo que en menor medida según los resultados. Vamos a proceder a dejarlas así ,para que mas adelante sea el algoritmo de eliminación hacia atrás el que decida que columnas deben desaparecer.

Ya terminamos con la parte de análisis de datos.

Como ya se fue adelantando previamente, lo que se busca predecir es el precio del diamante. Se entiende que el precio está determinado según las características que el diamante posea.

Por lo tanto el precio va a ser la variable dependiente mientras que el resto van a ser las independientes.

Separacion de variables dependientes de las independientes

Antes de comenzar con la separación de los archivos, es importante destacar que se ha optado por utilizar el paradigma de objetos en este trabajo práctico. Esto se hace con el objetivo de facilitar la reutilización de métodos y funciones en diferentes notebooks de Jupyter, como se puede observar en la imagen a la izquierda.

Existen varios archivos .ipynb, siendo el primero "01 - Análisis de datos" el que ya ha sido explicado. A continuación, se encuentra el archivo "02..." y así sucesivamente.

En la mayoría de estos archivos, excepto el último llamado "Conclusiones.ipynb", se hace uso constante de la clase "RegresionModelo". Esta clase se encarga de la separación de las variables dependientes e independientes, la eliminación de variables que afectan negativamente al rendimiento del algoritmo y la asignación del algoritmo de predicción a ejecutar, ya sea lineal, de árboles, polinómico, etc. También se encarga de dar la orden al algoritmo indicado de realizar la predicción para así finalmente obtener los resultados para mostrar en pantalla mediante graficos. Además, esta clase también cuenta con algunas funciones menores que se irán explicando gradualmente. No obstante, se intentará explicar la funcionalidad completa de este objeto en su totalidad. Esto solo es una pequeña introducción.

Se ha decidido separar las predicciones en varios archivos de Jupyter Notebook, que van desde el "02" al "07", y así sucesivamente. La razón detrás de esta decisión es evitar que un solo archivo sea demasiado grande y más difícil de mantener, así como también reducir la posibilidad de cometer errores. Además, hay algoritmos que requieren un tiempo considerable para realizar sus predicciones, como es el caso de la regresión con máquinas vectoriales, y no es conveniente tener que esperar a que se complete la ejecución de ese algoritmo para poder ejecutar otro.

The screenshot shows a Jupyter Notebook environment. On the left, a sidebar displays a file tree:

- PARADIGMASDEPROGRAMACIONUNSAM-TPS
- > __pycache__
- > svrLinealCapturas
- > TP-1 Diamantes (Algoritmos de regresion)
- > __pycache__
- 01 - Analisis de datos.ipynb
- 02 - Separacion y eliminacion de datos + Regresion Li... M
- 03 - Regresion Polinomica.ipynb
- 04 - Regresion con mquinas de soporte vectorial (SVR).ipy...
- 05 - Regresion con Arboles de desicion copy.ipynb
- 06 - Regresion con RANDOM FOREST.ipynb
- 07 - Regresion con KNN.ipynb
- 08 - Conclusiones.ipynb
- algoritmos.py
- diamonds.csv
- output.png
- predicción.py
- variable.pkl

The main notebook area shows a Python script named 'predicción.py' with the following code:

```
algoPrediccion = predicción.RegresionModelo(df)
algoPrediccion.df
algoPrediccion.definirConjuntoDeVariablesIndependientes('price')
algoPrediccion.X
```

Below this, another code block shows the 'RegresionModelo' class definition:

```
TP-1 Diamantes (Algoritmos de regresion) > predicción.py > RegresionModelo
from algoritmos import RegresionLineal
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from math import sqrt

class RegresionModelo:
    X=None
    Y=None
    df= None
    X_train = None
    X_test = None
    y_train = None
    y_test = None
    SL = 0.05

    listaColumnaCategorica = None
    algoritmo = RegresionLineal()
    y_pred=None

    mean_absolute_errorDic = None
    mean_squared_errorDic = None
    root_mean_squared_errorDic = None
    meanAbsoluteErrorPorcentajeDic = None
```

El objeto Regresion modelo recibe como parámetro al df y luego se ejecuta el método definirConjuntoDeVariables... que es un método que recibe un string que indica que del dataframe que le asignamos en el constructor al objeto todas las columnas salvo price son variables independientes y justamente price es la dependiente y la que se intenta predecir.

```

algoPrediccion = prediccion.RegresionModelo(df)
algoPrediccion.df
algoPrediccion.definirConjuntoDeVariablesIndependientesYDependientes('price')
algoPrediccion.X

```

Por lo general se trato de darle nombre a los metodo los mas descriptivos posibles

```

def obtenerListaDeColumnasDF(self):
    return self.df.columns.tolist()

def obtenerSoloListaDeVariablesIndependientes(self, dependiente : str) -> list:

    columnas = self.obtenerListaDeColumnasDF()
    columnas.remove(dependiente)
    return columnas

##2
def definirConjuntoDeVariablesIndependientesYDependientes(self, varDependiente: str):

    listaDeVariablesIndependientes : list = self.obtenerSoloListaDeVariablesIndependientes(varDependiente)
    self.X= self.df.loc[:,listaDeVariablesIndependientes]
    self.Y = self.df.loc[:,[varDependiente]]

```

algoPrediccion.X

✓ 0.0s

	carat	cut	color	clarity	depth	table	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	4.34	4.35	2.75
...
53935	0.72	Ideal	D	SI1	60.8	57.0	5.75	5.76	3.50
53936	0.72	Good	D	SI1	63.1	55.0	5.69	5.75	3.61
53937	0.70	Very Good	D	SI1	62.8	60.0	5.66	5.68	3.56
53938	0.86	Premium	H	SI2	61.0	58.0	6.15	6.12	3.74
53939	0.75	Ideal	D	SI2	62.2	55.0	5.83	5.87	3.64

53940 rows × 9 columns

Ya tenemos el conjunto de variables dependientes e independientes

```
algoPrediccion.Y
✓ 0.0s

  price
0    326
1    326
2    327
3    334
4    335
...
53935  2757
53936  2757
53937  2757
53938  2757
53939  2757

53940 rows × 1 columns
```

De categoricos a valores numericos

Ya de antemano contamos con variables independientes categoricos.
El siguiente paso es convertir los datos de aquellas columnas en datos numericos
Las variables independientes categoricas son

```
algoPrediccion.columnasCategorica()
✓ 0.0s

['cut', 'color', 'clarity']
```

Metodo que solo retorna solo los nombres de aquellas columnas que no almacen numeros osea las que son categoricas.

```

def columnasCategorica(self) :
    columnasNoNumericas = self.X.select_dtypes(exclude=['number'])
    columnasNoNumericas = columnasNoNumericas.columns.tolist()
    self.listaColumnaCategorica= columnasNoNumericas
    return columnasNoNumericas

```

Aca abajo vemos la ejecucion del metodo que recibe por parametro el nombre de la columna que va a pasar de categorico a numerico

Vallamos paso por paso

	algoPrediccion.deCategoricoANumerico('cut')																		
✓	0.0s																		
<table border="1"> <thead> <tr> <th></th> <th>cut</th> <th>cut</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>2</td> <td>Ideal</td> </tr> <tr> <td>1</td> <td>3</td> <td>Premium</td> </tr> <tr> <td>2</td> <td>1</td> <td>Good</td> </tr> <tr> <td>3</td> <td>3</td> <td>Premium</td> </tr> <tr> <td>4</td> <td>1</td> <td>Good</td> </tr> </tbody> </table>			cut	cut	0	2	Ideal	1	3	Premium	2	1	Good	3	3	Premium	4	1	Good
	cut	cut																	
0	2	Ideal																	
1	3	Premium																	
2	1	Good																	
3	3	Premium																	
4	1	Good																	
	algoPrediccion.deCategoricoANumerico('color')																		
✓	0.0s																		
<table border="1"> <thead> <tr> <th></th> <th>color</th> <th>color</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1</td> <td>E</td> </tr> <tr> <td>1</td> <td>1</td> <td>E</td> </tr> <tr> <td>2</td> <td>1</td> <td>E</td> </tr> <tr> <td>3</td> <td>5</td> <td>I</td> </tr> <tr> <td>4</td> <td>6</td> <td>J</td> </tr> </tbody> </table>			color	color	0	1	E	1	1	E	2	1	E	3	5	I	4	6	J
	color	color																	
0	1	E																	
1	1	E																	
2	1	E																	
3	5	I																	
4	6	J																	

```
algoPrediccion.deCategoricoANumerico('clarity')
```

✓ 0.0s

	clarity	clarity
0	3	SI2
1	2	SI1
2	4	VS1
3	5	VS2
4	3	SI2

La tabla que se muestra es como quedo numericamente la tabla. Vemos para este caso que al valor S12 Se le asigno el numero 3 (se comprueba en la fila 0 y 4)

```
def deCategoricoANumerico(self, columna):
    labelencoder_X = LabelEncoder()
    self.X[columna] = labelencoder_X.fit_transform(self.X[columna])
    # Se emiten ambos grupos de datos para comparar
    return pd.concat([pd.DataFrame(self.X[columna]),self.df[columna] ], axis=1).head()
```

Las variables categóricas, como la talla (Cut), el color (Color) y la claridad (Clarity), se podrían beneficiados de la codificación one-hot, que implica crear variables dummy para cada categoría en lugar de escalar los datos. La razón principal es que estas variables no tienen un orden inherente y no se puede establecer una relación numérica directa entre las diferentes categorías.

```
algoPrediccion.conversionDeCategorioADummieNumerico()
```

✓ 0.0s

```
array([[0. , 0. , 1. , ..., 3.95, 3.98, 2.43],
       [0. , 0. , 0. , ..., 3.89, 3.84, 2.31],
       [0. , 1. , 0. , ..., 4.05, 4.07, 2.31],
       ...,
       [0. , 0. , 0. , ..., 5.66, 5.68, 3.56],
       [0. , 0. , 0. , ..., 6.15, 6.12, 3.74],
       [0. , 0. , 1. , ..., 5.83, 5.87, 3.64]])
```

```
algoPrediccion.evitarTrampa()
```

```
def conversionDeCategoricoADummieNumerico(self) :  
    onehotencoder = make_column_transformer((OneHotEncoder(), self.listaColumnaCategorica), remainder = "passthrough")  
    self.X = onehotencoder.fit_transform(self.X)  
  
    return self.X
```

Evitamos la trampa de los valores dummies

```
def evitarTrampa(self) :  
    self.X = self.X[:,1:]  
    ##2
```

A continuación, dividimos el 80% de los datos para el conjunto de entrenamiento y el 20% de los datos al conjunto de pruebas usando el código de abajo. Y por ultimo mostramos su dimension

```
algoPrediccion.divisionDeConjuntos()  
algoPrediccion.X_train.shape
```

✓ 0.0s

(43152, 25)

```
algoPrediccion.X_test.shape
```

✓ 0.0s

(10788, 25)

```
algoPrediccion.y_test.shape
```

✓ 0.0s

(10788, 1)

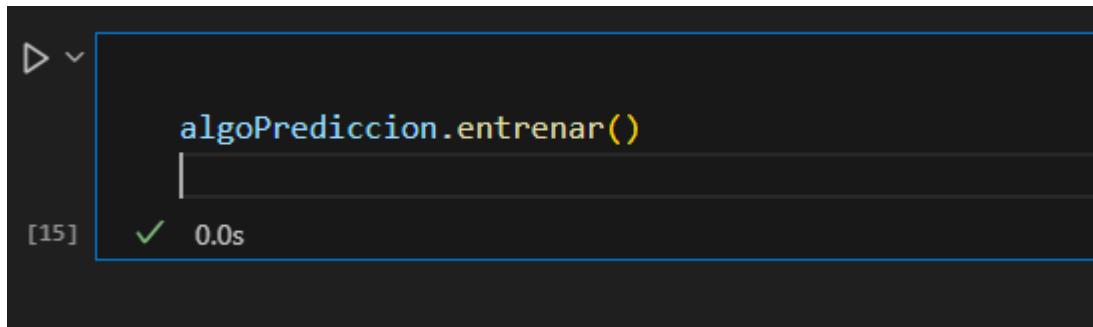
```
algoPrediccion.y_train.shape
```

✓ 0.0s

(43152, 1)

```
def divisionDeConjuntos(self) :  
    self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(self.X, self.Y, test_size=0.2, random_state=0)
```

El siguiente paso es el entrenamiento del modelo.



The screenshot shows a Jupyter Notebook cell with the following code:

```
[15] algoPrediccion.entrenar()
```

The cell has a green checkmark icon and a time stamp of "0.0s".

```
def entrenar(self) :  
    self.algoritmo.entrenar(self)
```

Es importante destacar que el objeto "RegresionModelo" tiene predefinido el modelo de regresión lineal como algoritmo por defecto. Como mencioné anteriormente, el objeto "RegresionModelo" delega la tarea de hacer las predicciones y devolver los resultados a otro objeto, en este caso, "RegresionLineal". Esto permite que el objeto "RegresionModelo" pueda generar los gráficos correspondientes.

Cuando pasamos al siguiente algoritmo, simplemente necesitamos cambiar el atributo "algoritmo" del objeto "RegresionModelo" por otro algoritmo específico. Es importante destacar que se está utilizando el patrón de diseño "strategy", el cual nos permite intercambiar diferentes algoritmos en tiempo de ejecución.

En resumen, el objeto "RegresionModelo" tiene una configuración predeterminada con el modelo de regresión lineal, pero puede cambiar de algoritmo fácilmente gracias al patrón "strategy". Esto brinda flexibilidad en la elección y evaluación de diferentes algoritmos de regresión.

```

class RegresionModelo:

    X=None
    Y=None
    df= None
    X_train = None
    X_test = None
    y_train = None
    y_test = None
    SL = 0.05

    (variable) algoritmo: RegresionLineal

💡 algoritmo := RegresionLineal()
y_pred=None

mean_absolute_errorDic = None
mean_squared_errorDic = None
root_mean_squared_errorDic = None

```

Implementacion del patron de diseño.: Python no tiene interfaces pero si tiene Herencia multiple. Así que tranquilamente una clase puede hacer indirectamente de rol de interfaz.

En este caso el strategy es RegresionLineal y debe implementar obligatoriamente como si fuera un contrato los metodos de TipoDeRegresion. Como podemos ver RegresionLineal tiene el metodo entrenar que justamente hace eso, entrenar el modelo utilizando el conjunto de entrenamiento del objeto Regresion Modelo que recibe por parametro todo el objeto RegresionModelo para así poder hacer uso de sus atributos para realizar el entrenamiento y luego retornar la predicción .

```

class TipoDeRegresion():
    def entrenar(self,RegresionModelo ):
        pass

    def prediccion(self,RegresionModelo):
        pass

    def nombreDeRegresion(self) :
        pass

class RegresionLineal(TipoDeRegresion):
    regressor = LinearRegression()
    def entrenar(self,RegresionModelo):
        self.regressor.fit(RegresionModelo.X_train, RegresionModelo.y_train)

    def prediccion(self,RegresionModelo):
        y_pred = self.regressor.predict(RegresionModelo.X_test).flatten()
        return y_pred
    def nombreDeRegresion(self):
        return "Regresion Lineal"

```

Obviamente hay mas strategys que representan a cada algoritmo visto en la materia. Aca vemos el caso del Polinomio. Tiene los mismos metodos que regresionLineal solo que responde a cada uno de manera diferente. Osea existe el polimorfismo y por eso en este tp es muy facil cambiar de algoritmo de regresion . Simplemente basta con modificar el atributo algoritmo del objeto RegresorModelo

```

class RegresionPolinomica(TipoDeRegresion) :

    regressor = LinearRegression()
    grado = None
    X_poly = None

    def __init__(self, grado ):
        self = self
        self.grado = grado

    def nombreDeRegresion(self):
        return "Regresion Polinomica"

    def crearPolinomio(self,RegresionModelo) :
        poly_reg = PolynomialFeatures(degree=self.grado)
        self.X_poly = poly_reg.fit_transform(RegresionModelo.X_train)

    def entrenar(self, RegresionModelo):
        self.crearPolinomio(RegresionModelo)
        self.regressor.fit(self.X_poly, RegresionModelo.y_train)

    def prediccion(self, RegresionModelo):
        poly_reg = PolynomialFeatures(degree=self.grado)
        X_poly_test = poly_reg.fit_transform(RegresionModelo.X_test)
        y_pred = self.regressor.predict(X_poly_test).flatten()

        return y_pred

```

Predicción sobre los datos de la prueba:

Despues del entrenamiento viene los resultados de la prediccion y de eso se encarga el strategy,

```

algoPrediccion.prediccion()
✓ 0.0s

```

```

def prediccion(self) :
    self.y_pred= self.algoritmo.prediccion(self)

```

Por supuesto los resultado se guardan en el atributo y_pred para asi no tener que volver a realizar el llamado

```

    def prediccion(self, RegresionModelo):
        y_pred = self.regressor.predict(RegresionModelo.X_test).flatten()
        return y_pred
    def nombreDeRegresion(self):
        return "Regresion Lineal"

```

Aclaracion : nombreDeRegresion es un metodo relevante para el final . Tampoco es importante entender su razon de existir

RESULTADOS DE LA PREDICCIÓN

algoPrediccion.resultadoDeEntrenamiento()

✓ 0.0s

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4910.11	177.11	3.74
1	6424	7589.96	1165.96	18.15
2	5510	6138.47	628.47	11.41
3	8770	10293.72	1523.72	17.37
4	4493	5248.09	755.09	16.81
...
10783	1289	1126.88	162.12	12.58
10784	3435	3515.69	80.69	2.35
10785	3847	5266.79	1419.79	36.91
10786	8168	6975.63	1192.37	14.60
10787	1917	2312.25	395.25	20.62

```

def resultadoDeEntrenamiento(self) :
    y_test = np.ravel(self.y_test) ##EVITA ERRORES.

    pd.set_option('display.float_format', lambda x: '{:.2f}'.format(x)) ##PARA EVITAR LA NOTACION CIENTIFICA
    dff = pd.DataFrame({'Actual': y_test, 'Prediccion':self.y_pred})
    diferencia = abs(dff['Prediccion']-dff['Actual'])
    diferenciaPorcentual=abs((dff['Prediccion']*100)/(dff['Actual']))-100

    dff = pd.DataFrame({'Actual': y_test,
                        'Prediccion':self.y_pred,
                        "Error Absoluto": diferencia,
                        "Error porcentual absoluto %":diferenciaPorcentual
                      })
    return dff

```

Como se puede ver este metodo tiene como objetivo mostrar los resultados de la predicción en formato de tablas. Y hace uso del atributo y_pred obtenido en el método entrenar y prediccion() que hace uso del strategy.

Veamos que se muestra

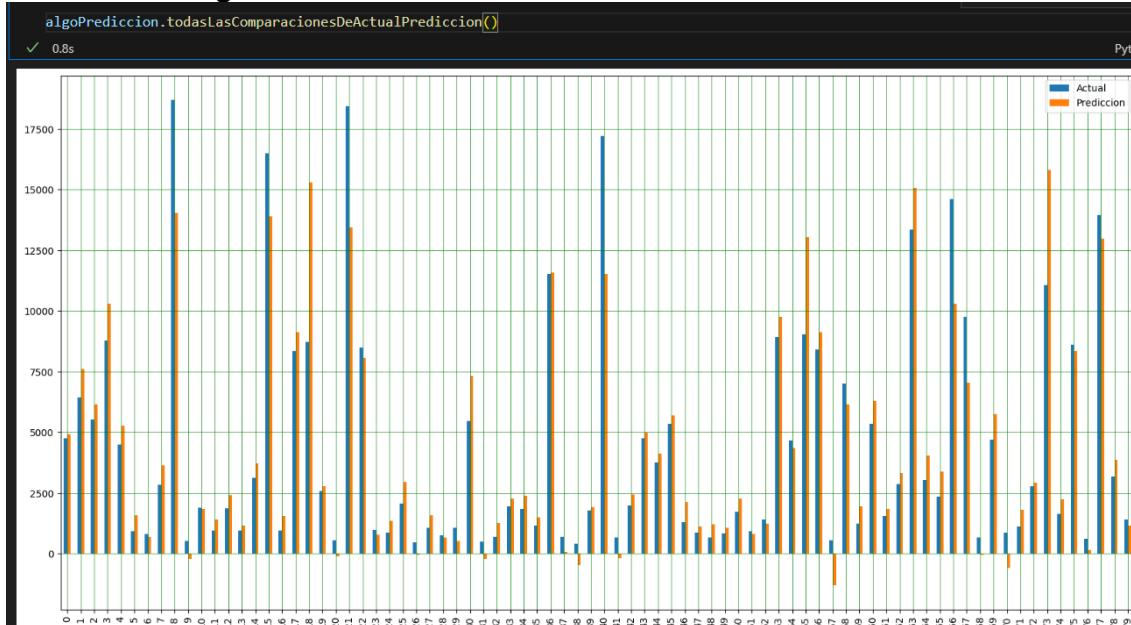
algoPrediccion.resultadoDeEntrenamiento()

✓ 0.0s

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4910.11	177.11	3.74
1	6424	7589.96	1165.96	18.15
2	5510	6138.47	628.47	11.41
3	8770	10293.72	1523.72	17.37
4	4493	5248.09	755.09	16.81
...
10783	1289	1126.88	162.12	12.58
10784	3435	3515.69	80.69	2.35
10785	3847	5266.79	1419.79	36.91
10786	8168	6975.63	1192.37	14.60
10787	1917	2312.25	395.25	20.62

Por supuesto el Error absoluto es para cada caso (Diamante predecido en particular) . Vemos que el primer caso el error solo fue de un 3.74 % con una diferencia de 177.11 unidades (o dolares si tenemos en cuenta que estamos intentando predecir precios) . 3.74 % de error un numero muy bueno pero al parecer no es normal que suceda eso como se puede ver en el resto de las filas. Por las dudas aclaro que el error absoluto es la diferencia entre Actual y predicción. Esta tabla se va a repetir en el resto de los algoritmos de predicción.

Es hora de los graficos



Se muestra una comparacion visual entre el valor actual y el predecido. Solo en 80 casos de lo contrario es mas dificil comprender el dibujo. En lineas generales no esta tan mal.

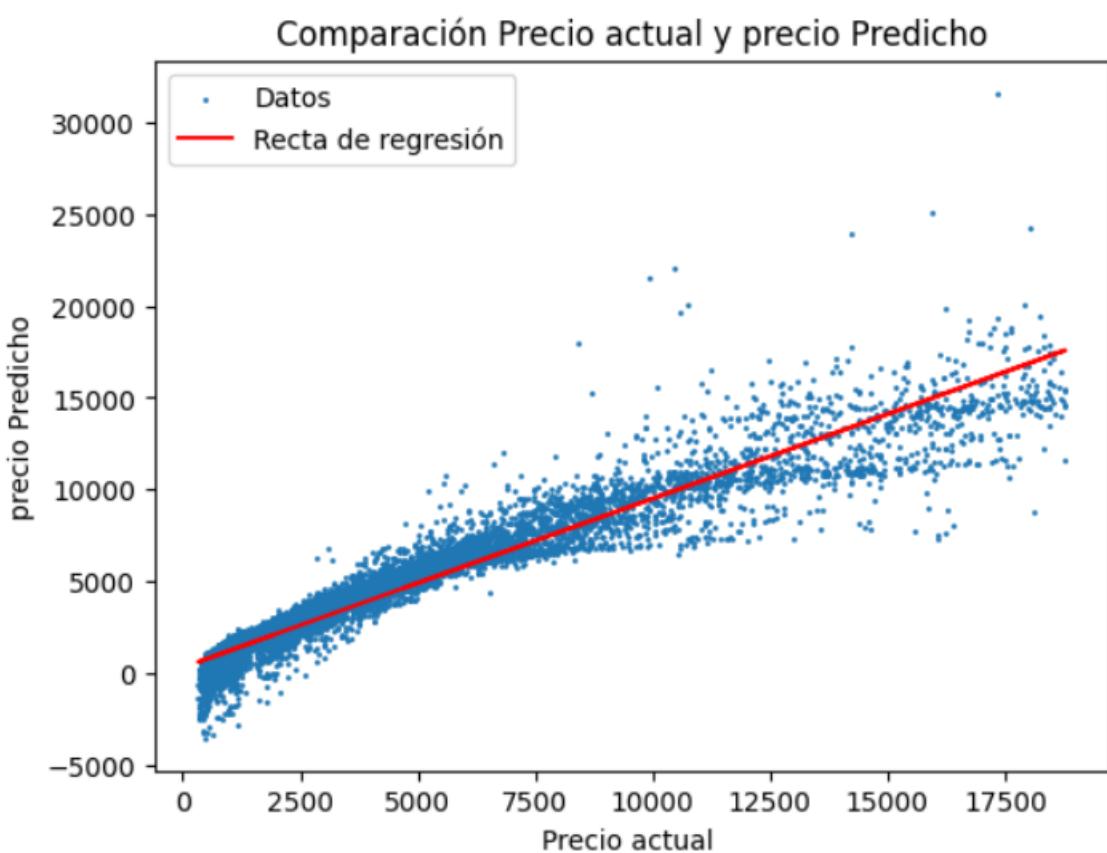
Antes de continuar veamos el codigo del metodo que se ejecuto en la imagen de arriba

```
def todasLasComparacionesDeActualPrediccion(self) :
    self.graficoActualPrediccion()
    self.graficoRegresionPrediccion()
    self.graficoPrediccion2()

def graficoActualPrediccion(self):
    df1 = self.resultadoDeEntrenamiento()
    df1 = df1.loc[:, ['Actual', 'Prediccion']].head(80)
    df1.plot(kind='bar', figsize=(20,10))
    plt.grid(which='major', linestyle='-', linewidth='0.5', color='green')
    plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')
    plt.show()

def graficoRegresionPrediccion(self) :
    actual = np.squeeze(self.y_test)
    predicción = np.squeeze(self.y_pred)
    # Ajustar la recta de regresión lineal
    coefficients = np.polyfit(actual, predicción, 1)
    regression_line = np.polyval(coefficients, actual)
    # Gráfico de dispersión
    plt.scatter(actual, predicción, label='Datos', s=1)
    # Línea de regresión
    plt.plot(actual, regression_line, color='red', label='Recta de regresión')
    # Etiquetas de los ejes y título del gráfico
    plt.xlabel('Precio actual ')
    plt.ylabel('precio Predicho')
    plt.title('Comparación Precio actual y precio Predicho')
    # Mostrar la leyenda
    plt.legend()
    # Mostrar el gráfico
    plt.show()
```

Los nombres de los métodos no son los más apropiados, sin embargo, cambiarlos en este punto podría ser riesgoso. Lo más relevante aquí es que al ejecutar el método "todasLasComparacionesDel..." se muestran automáticamente tres gráficos en la pantalla del Jupyter Notebook.

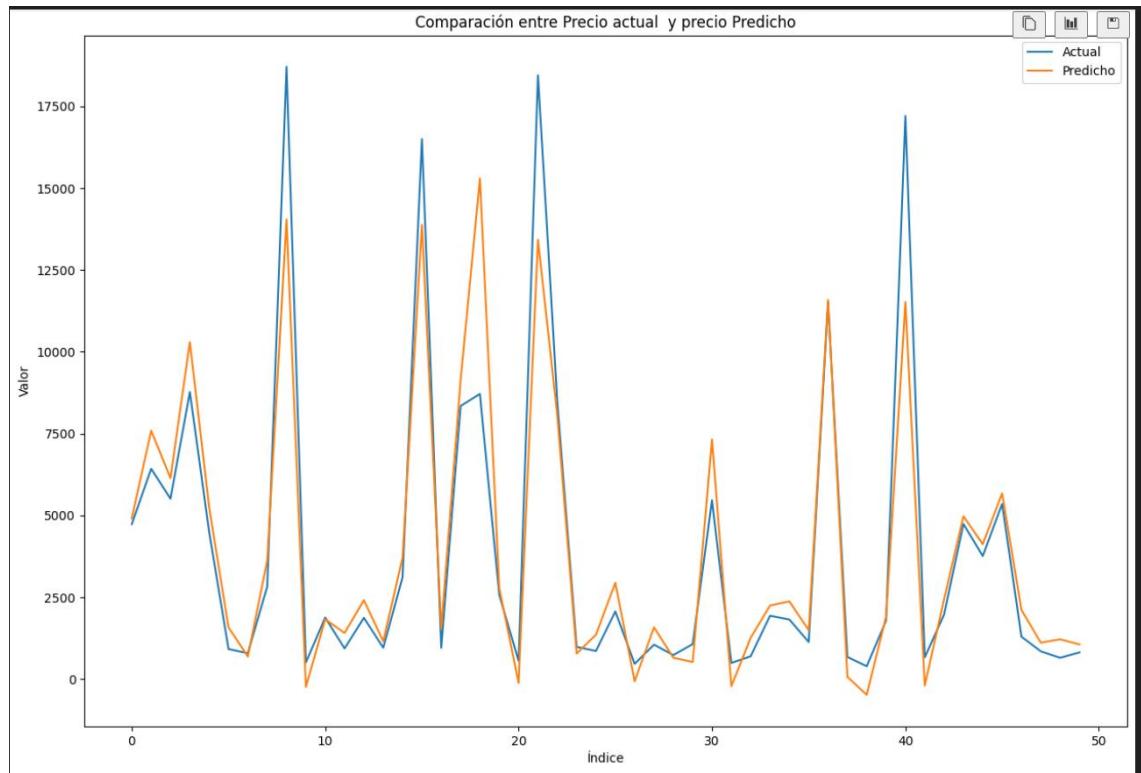


En el gráfico resultante, los puntos dispersos representan los pares de valores reales y predichos. Si observamos puntos que están lejos de la recta de regresión generada (la línea roja), podría significar que hay discrepancias considerables entre los valores reales y los valores predichos correspondientes a esos puntos.

Cuando decimos que un punto está "mas lejos" de la recta de regresión generada, significa que hay una mayor diferencia entre el valor predicho y el valor real correspondiente. En la regresión lineal ideal, los puntos de dispersión estarían muy cerca de la recta de regresión, lo que indicaría una buena capacidad de predicción.

Sin embargo, si hay puntos que se notan claramente que están alejados de la recta de regresión, lo que podría indicar una falta de ajuste o un rendimiento ineficiente del modelo en algunos casos.

En el gráfico resultante, los puntos dispersos representan los pares de valores reales y predichos. Si hay puntos que están lejos de la recta de regresión generada (la línea roja), significa que hay discrepancias considerables entre los valores reales y los valores predichos correspondientes a esos puntos.



El conjunto de datos utilizado abarca 50 casos, lo cual proporciona una muestra más amplia para comprender mejor los resultados. La idea principal es evaluar la coincidencia entre dos líneas en un punto específico (x, y), y se considera que cuanto mayor sea la superposición entre ambas líneas, mejor será la coincidencia.

Por ultimo la conclusion

```
algoPrediccion.resultadoDeEntrenamiento().describe()
```

✓ 0.0s

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
count	10788.00	10788.00	10788.00	10788.00
mean	3929.21	3914.43	737.74	39.46
std	3981.60	3815.38	839.18	65.48
min	326.00	-3558.25	0.09	0.00
25%	947.50	1040.45	259.89	9.13
50%	2398.00	2828.63	532.77	20.28
75%	5311.25	5892.33	880.55	41.37
max	18787.00	31561.38	14232.38	893.24

```
algoPrediccion.conclusiones()
```

✓ 0.0s

	Media	Error %	Efectividad %
Mean Absolute Error	737.74	18.76	81.24
Root Mean Squared Error	1117.32	28.41	71.59
Mean Squared Error	1248413.36	NaN	NaN

Primero, mostramos los valores estadísticos, los cuales utilizan un método previamente explicado. Al analizar estos resultados, es interesante observar, por ejemplo, que el mejor resultado presenta un error de 0.09 dólares, lo cual indica un error porcentual muy bajo, cercano al 0%. Esto sugiere que, a pesar de los resultados de la tabla de conclusiones, nuestro algoritmo es capaz de realizar predicciones precisas en varios casos.

Ahora, procedemos con el método de conclusiones. Este método nos permite evaluar qué tan buenas o malas fueron nuestras predicciones. Antes de continuar, revisemos el código.

```

def conclusiones(self) :

    mean_absolute_error = metrics.mean_absolute_error(self.y_test, self.y_pred)
    mean_squared_error = metrics.mean_squared_error(self.y_test, self.y_pred)
    root_mean_squared_error = np.sqrt(mean_squared_error)
    promedioDePrecios = self.df['price'].mean()
    errorCuadraticoPorcentaje = (root_mean_squared_error * 100) / promedioDePrecios
    efectividadCuadraticaPrediccion = 100 - errorCuadraticoPorcentaje

    errorAbsolutoPorcentaje = (mean_absolute_error * 100) / promedioDePrecios
    efectividadAbsolutaPrediccion = 100 - errorAbsolutoPorcentaje

    # Crea el DataFrame con los resultados
    data = {
        'Media': [mean_absolute_error,root_mean_squared_error,mean_squared_error] ,
        'Error %' : [errorAbsolutoPorcentaje,errorCuadraticoPorcentaje,np.nan] ,
        'Efectividad %' : [efectividadAbsolutaPrediccion,efectividadCuadraticaPrediccion,np.nan]
    }

    nombreDeRegresion = self.algoritmo.nombreDeRegresion()

    self.mean_absolute_errorDic[nombreDeRegresion] = mean_absolute_error
    self.mean_squared_errorDic[nombreDeRegresion] = mean_squared_error
    self.root_mean_squared_errorDic[nombreDeRegresion] = root_mean_squared_error

    self.meanAbsoluteErrorPorcentajeDic[nombreDeRegresion]=errorAbsolutoPorcentaje
    self.meanSquaredErrorPorcentajeDic[nombreDeRegresion]=errorCuadraticoPorcentaje
    self.promedioAbsolutoEfectividadPorcentajeDic[nombreDeRegresion]=efectividadAbsolutaPrediccion
    self.promedioCuadraticoEfectividadPorcentajeDic[nombreDeRegresion]=efectividadCuadraticaPrediccion

    self.guardarArchivo()

    df = pd.DataFrame(data, index=['Mean Absolute Error','Root Mean Squared Error', 'Mean Squared Error'])
    return df

```

Por ahora, es suficiente con saber que el método CONCLUSION() devuelve una tabla y, además, va guardando valores en cuatro atributos que son un diccionarios del objeto RegresionModelo. Estos atributos se encuentran por encima del método guardarArchivo() y van a ser extremadamente útiles para comparar el rendimiento de todos los algoritmos en la conclusión final. Sobre el método guardarArchivo() no es necesario prestarle demasiada atención ya que su función principal es guardar el objeto RegresionModelo, junto con sus estados, en un archivo para poder ejecutarlo en otro Jupiter Notebook.

```

def guardarArchivo(self) :
    with open('variable.pkl', 'wb') as f:
        pickle.dump(self, f)

```

Bien veamos la tabla que se retorna en el método CONCLUSION() nuevamente

	Media	Error %	Efectividad %
Mean Absolute Error	737.78	18.76	81.24
Root Mean Squared Error	1117.33	28.41	71.59
Mean Squared Error	1248435.26	NaN	NaN

Para la fila que abarca Mean Absolute Error (Error Absoluto Medio): El valor de 737.78 indica que, en promedio, las predicciones del modelo tienen una diferencia absoluta de 737.78 unidades con respecto a los valores reales. Un menor valor de MAE indicaría un mejor resultado y por lo tanto nos diría que existe una mayor precisión del modelo. El error porcentual asociado es del 18.76%, lo que implica que el algoritmo de regresión lineal se equivoca en un promedio del 18.76% al realizar las predicciones. Por lo tanto teniendo en cuenta el error porcentual asociado y haciendo una sencilla cuenta de 100>Error% nos daría que tan efectivo fue nuestro algoritmo .En este caso en particular nos arroja un resultado de un 81.24%

Para la fila que abarca Root Mean Squared Error (Error Cuadrático Medio): El valor de 1117.33 representa la raíz cuadrada del error cuadrático medio, lo que indica una dispersión promedio de 1117.33 unidades entre las predicciones (y_{pred}) y los valores reales (y_{train}). Tal cual como pasa con el MAE, un valor menor de RMSE indica una mayor precisión del modelo. El error porcentual asociado es del 28.41% y la efectividad se encuentra en un 71.59%.

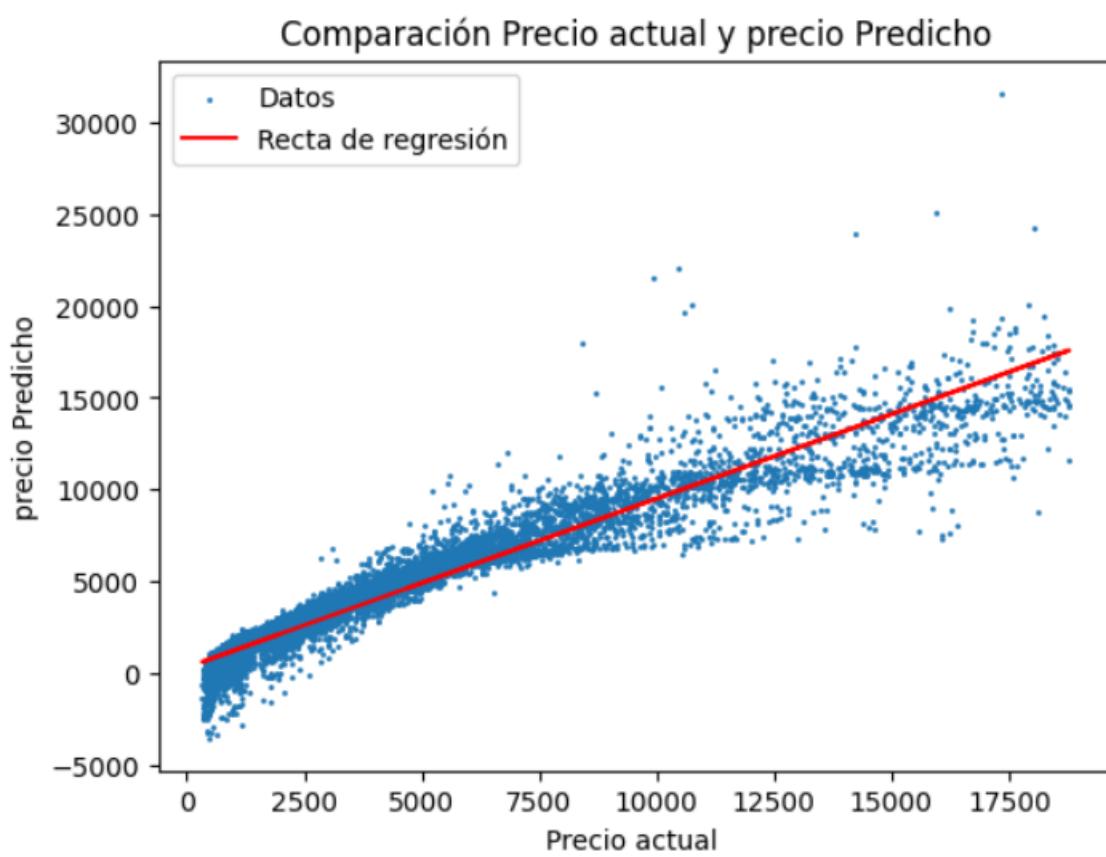
Sobre el Mean Squared Error (Error Cuadrático Medio): El valor de 1248435.26 representa el error cuadrático medio, que es el promedio de los errores al cuadrado entre las predicciones y los valores reales. En este trabajo práctico se decidió de forma arbitraria no proporcionar datos sobre el error porcentual y su efectividad para MSE

Sin embargo de las 3 métricas que se ve en la tabla de arriba la que nos más interesa es el error Cuadrático y su efectividad ya que esta última es más sensible porque penaliza de manera más significativa los errores más grandes, ya que se elevan al cuadrado por lo que la conclusión definitiva para este caso sería que nuestro algoritmo fue efectivo en un 71.59 y tuvo un error de un 28.41%. Aun así el algoritmo puede realizar buenas predicciones de hecho en algunos casos logra muy buenos resultados como se pudo ver en la tabla y en el gráfico de abajo que existen casos donde la predicción es realmente buena

```
algoPrediccion.resultadoDeEntrenamiento().sort_values('Error Absoluto', ascending=True).head(30)
```

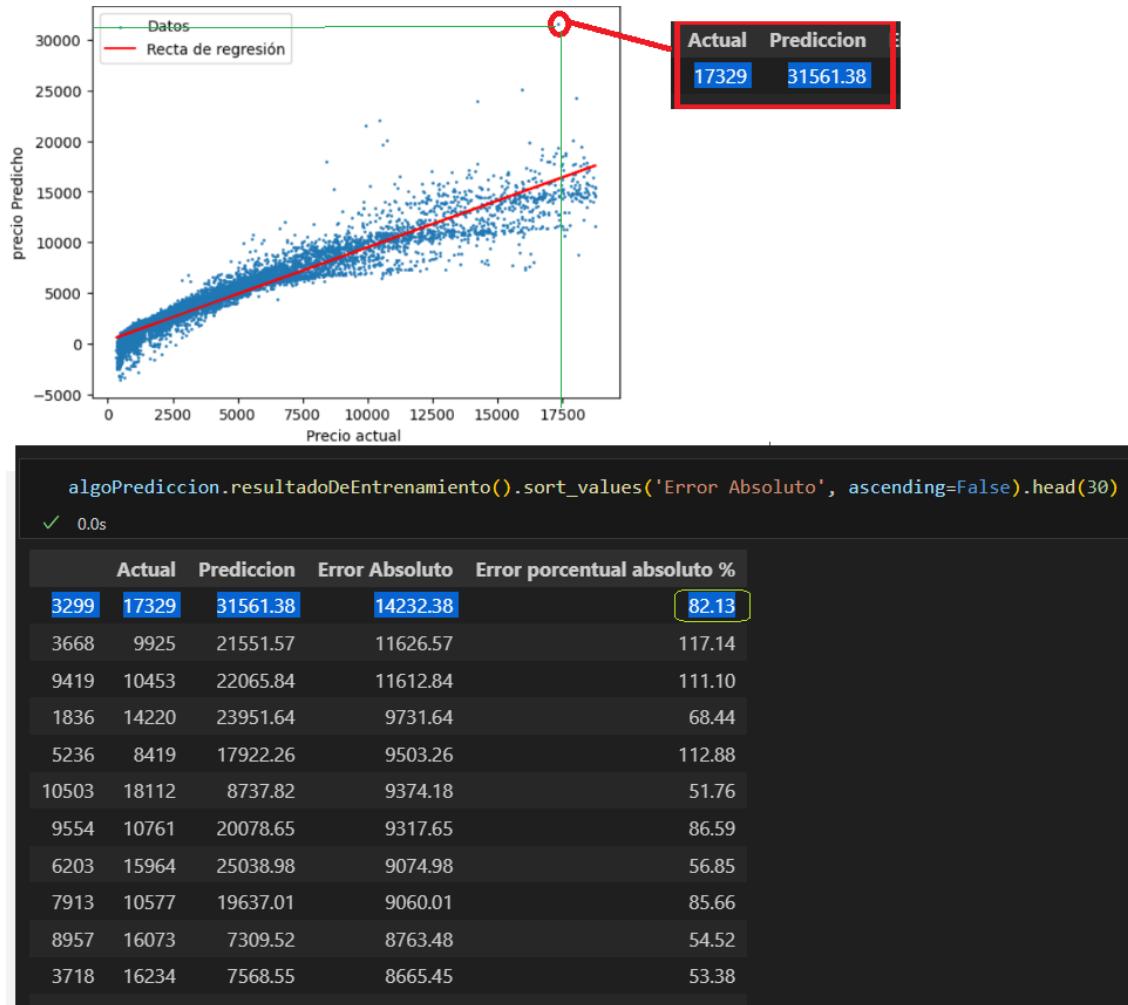
✓ 0.0s

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
4490	627	627.09	0.09	0.01
8056	4690	4689.90	0.10	0.00
1728	1279	1279.30	0.30	0.02
7845	2414	2413.56	0.44	0.02
8260	680	679.55	0.45	0.07
2103	1367	1366.54	0.46	0.03
5599	680	679.53	0.47	0.07
6159	1639	1638.36	0.64	0.04
6276	9892	9891.21	0.79	0.01
725	2061	2061.87	0.87	0.04
7959	816	817.10	1.10	0.13
2681	601	602.12	1.12	0.19
5264	453	451.81	1.19	0.26
9816	2203	2201.77	1.23	0.06
8436	2179	2177.68	1.32	0.06
6054	15710	15708.61	1.39	0.01
2559	3164	3165.45	1.45	0.05
8672	1002	1000.52	1.48	0.15
6485	1219	1217.36	1.64	0.13
8908	6271	6272.69	1.69	0.03
1455	6232	6233.72	1.72	0.03
7399	453	451.20	1.80	0.40



Donde claramente se ve que existen casos en donde se logra un muy buen rendimiento. Sin embargo vemos que a medida que los precios tienden a ser mas alto la dispersion es mucho mayor hasta lograr una una diferencia entre el valor real y predicho muy grande magnitud en terminos relativos. Expliquemos esto ultimo

Observemos primero la tabla y el grafico de abajo



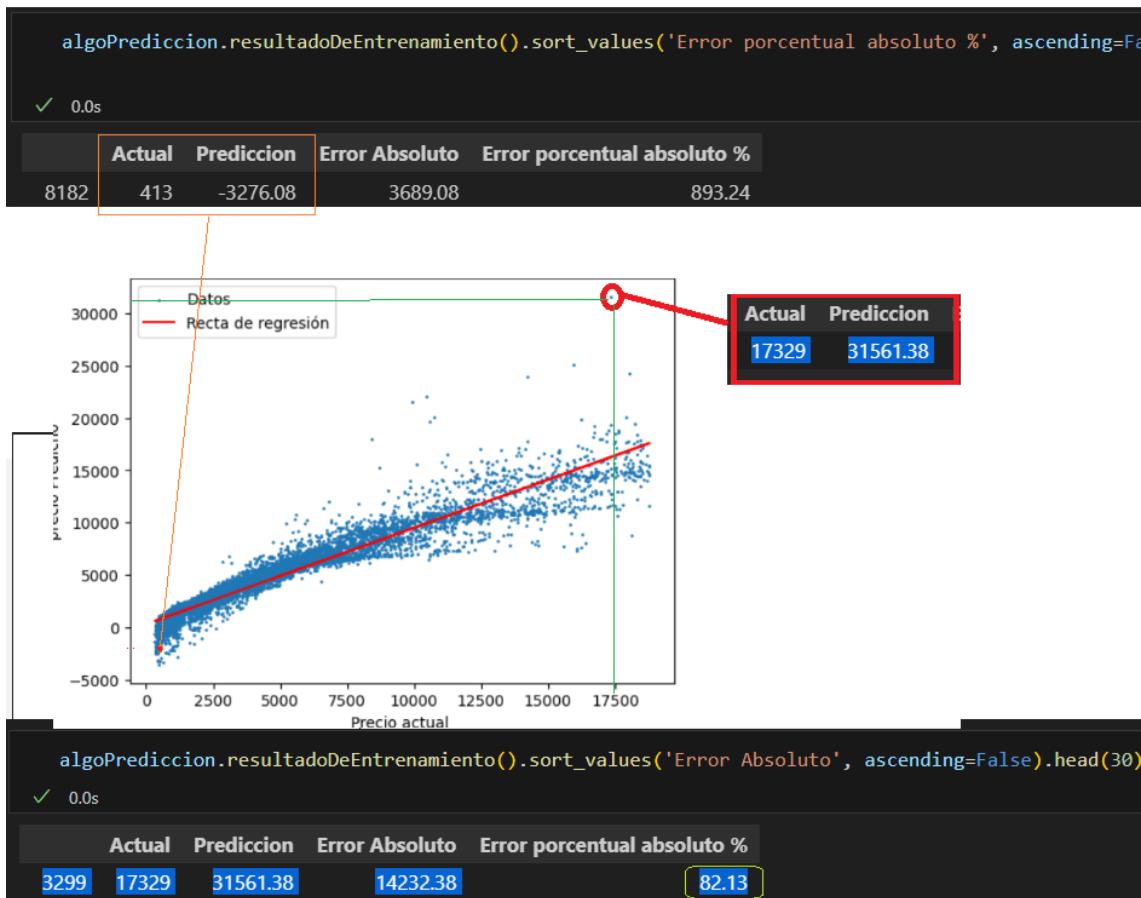
La tabla de arriba esta ordenada de forma descendente teniendo en cuenta los valores de la columna Error Absoluto. Sin embargo si observamos la primera fila vemos que el error absoluto es el mas grande ,aun asi no es el peor resultado de prediccion de nuestro algoritmo. Observemos que su error porcentual es de un 82% sin embargo existe casos peores , simplemente basta con ver la fila de abajo que da un error porcentual absoluto del 117.14. Si no existiese el error porcentual absoluto en la tabla uno al ver los resultados diria que el resultado de la primera fila es la peor prediccion que tuvo el algoritmo pero no es asi. De hecho existe casos peores como el de abajo

```
algoPrediccion.resultadoDeEntrenamiento().sort_values('Error porcentual absoluto %', ascending=False).head(30)
```

✓ 0.0s

	Actual	Predictión	Error Absoluto	Error porcentual absoluto %
8182	413	-3276.08	3689.08	893.24
10496	413	-3173.28	3586.28	868.35
388	490	-3558.25	4048.25	826.17

¿Nos estaria diciendo el grafico de arriba que para valores mas altos la predicciones suelen ser mas dificil? Veamos esta hipotesis con mayor claridad



Viendo el grafico de arriba concluimos lo siguiente : la cercanía a la linea de regresion efectivamente no es un indicador definitivo de la calidad de la prediccion. No debemos asumir directamente que los pares (y_{train} y_{pred}) más cercanos a la linea de regresion tienen mas probabilidades de indicar un porcentaje de predicción mejor , ni que los puntos mas alejados de la recta son aquellos con un resultado de predicción peores. La linea de regresión se utiliza para estimar la relación entre las variables, pero como vemos no captura todos los detalles individuales de cada punto de datos.

En resumen, cuando los valores reales y predichos son grandes, es posible que exista una mayor dispersión entre los puntos y la recta. Esto puede considerarse normal debido a la variabilidad en los datos por lo tanto no debemos hacer suposiciones automáticas sobre la precisión basadas únicamente en la cercanía a la línea de regresión. Si bien el gráfico puede dar indicios de que tan buena fue la predicción , mejor quedarse con la tabla de aca abajo.

	Media	Error %	Efectividad %
Mean Absolute Error	737.78	18.76	81.24
Root Mean Squared Error	1117.33	28.41	71.59
Mean Squared Error	1248435.26	NaN	NaN

Antes de pasar al siguiente modelo de regresión la pregunta es ¿ podriamos obtener un mejor resultado si aplicamos algun algoritmo de eliminación en aquellas variables o columnas menos significativas que no aporten mucho a los resultados de la predicciones? Veámoslo.

```
algoPrediccion.regresionOLSResultados()
```

```
def regresionOLS(self) :  
    return sm.OLS(endog = self.Y, exog = self.X).fit()  
  
def regresionOLSResultados(self):  
    return self.regresionOLS().summary()
```

OLS Regression Results						
Dep. Variable:	price		R-squared:	0.92		
Model:	OLS		Adj. R-squared:	0.92		
Method:	Least Squares		F-statistic:	2.688e+0		
Date:	Wed, 31 May 2023		Prob (F-statistic):	0.0		
Time:	12:42:18		Log-Likelihood:	-4.5573e+0		
No. Observations:	53940		AIC:	9.115e+0		
Df Residuals:	53916		BIC:	9.117e+0		
Df Model:	23					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
x1	579.7514	33.592	17.259	0.000	513.911	645.592
x2	832.9118	33.407	24.932	0.000	767.433	898.391
x3	762.1440	32.228	23.649	0.000	698.978	825.310
x4	726.7826	32.241	22.542	0.000	663.591	789.975
x5	3584.8729	216.253	16.577	0.000	3161.015	4008.731
x6	3375.7549	216.151	15.618	0.000	2952.097	3799.413
x7	3312.0191	216.278	15.314	0.000	2888.112	3735.926
x8	3102.8340	216.417	14.337	0.000	2678.655	3527.013
x9	2604.6063	216.533	12.029	0.000	2180.200	3029.012
x10	2118.6285	216.680	9.778	0.000	1693.934	2543.323
x11	1215.4749	217.078	5.599	0.000	790.001	1640.949
x12	-1400.3956	194.098	-7.215	0.000	-1780.829	-1019.962
x13	3944.7067	189.271	20.842	0.000	3573.734	4315.680
x14	2265.0765	190.203	11.909	0.000	1892.278	2637.875
x15	1302.1907	190.054	6.852	0.000	929.683	1674.699
x16	3178.0023	189.421	16.777	0.000	2806.736	3549.268

x17	2866.8280	189.665	15.115	0.000	2495.083	3238.573
x18	3607.3635	189.072	19.079	0.000	3236.781	3977.946
x19	3550.4185	189.245	18.761	0.000	3179.497	3921.340
x20	1.126e+04	48.628	231.494	0.000	1.12e+04	1.14e+04
x21	-63.8061	4.535	-14.071	0.000	-72.694	-54.918
x22	-26.4741	2.912	-9.092	0.000	-32.181	-20.767
x23	-1008.2611	32.898	-30.648	0.000	-1072.741	-943.781
x24	9.6089	19.333	0.497	0.619	-28.284	47.502
x25	-50.1189	33.486	-1.497	0.134	-115.752	15.515
Omnibus:		14433.356	Durbin-Watson:		1.183	
Prob(Omnibus):		0.000	Jarque-Bera (JB):		565680.446	
Skew:		0.577	Prob(JB):		0.00	
Kurtosis:		18.823	Cond. No.		7.49e+15	

Notes:

Se va a utilizar el algoritmo de eliminacion hacia atrás asi que de todo la tabla que se muestra arriba lo unico que nos interesa son los p valores. Asi que generamos otra tabla mas simplificada

algoPrediccion.todosLosP()

```
def todosLosP(self) :
    # Obtener los valores de p (p-values) de la regresion OLS
    valores = self.regresionOLS().pvalues
    # Formatear los valores de p con tres decimales
    valores_formateados = valores.apply(lambda x: "{:.3f}".format(x))
    # Verificar si los valores de p son mayores que el umbral de significancia (0.05)
    pMayorASL = valores > 0.05
    # Crear un DataFrame con los valores de p formateados y la indicacion de si son mayores que el umbral
    df = pd.DataFrame({'P-Values': valores_formateados, 'P>' + str(self.SL): pMayorASL})
    # NECESITO SABER LOS INDICES DE CADA FILA YA QUE REPRESENTAN A SU VEZ EL INDICE DE COLUMNAS
    # DE LAS VARIABLES INDEPENDIENTES. TENERLOS COMO X1,X2 X3 X4 no me sirve para nada para la el algoritmo de eliminacion
    #
    df.reset_index(inplace=True)
    df.drop('index', axis=1, inplace=True)
    # Reasignar un nuevo indice numerico al DataFrame
    df.index = range(len(df))
    # se devuelve el DataFrame con los valores de p y su comparacion con el umbral de significancia
    return df
```

Con el codigo de arriba logramos obtener una tabla mucho mas simplificada y mas util para lo que queremos hacer . Veamos el resultado

```
algoPrediccion.todosLosP()
```

	P-Values	P>0.05
0	0.000	False
1	0.000	False
2	0.000	False
3	0.000	False
4	0.000	False
5	0.000	False
6	0.000	False
7	0.000	False
8	0.000	False
9	0.000	False
10	0.000	False
11	0.000	False
12	0.000	False
13	0.000	False
14	0.000	False
15	0.000	False
16	0.000	False
17	0.000	False
18	0.000	False
19	0.000	False
20	0.000	False
21	0.000	False
22	0.000	False
23	0.619	True
24	0.134	True

La visualizacion se hace mucho mas sencilla y directa de esta forma . Como podemos ver las 2 ultimas que representan las columnas (o variables independiente) cuentan con P-values que superan al umbral establecido (0.05) . Tambien podemos ver que los indices de las filas no estan reprentados por X1 X2 X3

X4 etc lo cual nos es util luego para indicar que filas eliminar de forma automatica en un solo paso. Continuemos. Lo siguiente que se ejecuto en el jupiter notebook fue

```
> algoPrediccion.todosLosPQueSuperaAlSL()
29] ✓ 0.2s
..
  P-Values  P>0.05
  23      0.619      True
  24      0.134      True
...
def todosLosPQueSuperaAlSL(self) :
    return self.todosLosP().loc[self.todosLosP()['P>'+str(self.SL)] == True]
```

Por supuesto que el metodo de arriba seria mucho mas util si hubiera mas casos . Aun tiene mas utilidades sobre todo al momento de realizar la eliminacion. Pero visualmente sirve igual por mas pocos casos que haya para mostrar directamente aquellas columnas menos significativas para nuestro modelos que van a ser eliminadas.

Del metodo mostrado arriba nos interesa obtener los indices

```
algoPrediccion.obtenerIndicesDeAquellosQueSuperanAllS()
✓ 0.2s
[23, 24]
...
def obtenerIndicesDeAquellosQueSuperanAllS(self):
    return self.todosLosPQueSuperaAlSL().index.tolist()
```

Finalmente procedemos a la eliminacion y vemos los resultados

```
algoPrediccion.eliminarColumnasQueSuperenAlSL()  
algoPrediccion.todosLosP()
```

✓ 0.2s

P-Values P>0.05

0	0.000	False
1	0.000	False
2	0.000	False
3	0.000	False
4	0.000	False
5	0.000	False
6	0.000	False
7	0.000	False
8	0.000	False
9	0.000	False
10	0.000	False
11	0.000	False
12	0.000	False
13	0.000	False
14	0.000	False
15	0.000	False
16	0.000	False
17	0.000	False
18	0.000	False
19	0.000	False
20	0.000	False
21	0.000	False
22	0.000	False

Pasamos de 25 variables a 23. Efectivamente se realizo la eliminacion. Solo para estar seguro comprobemoslo

```
algoPrediccion.todosLosPQueSuperaAlSL()
```

✓ 0.1s

P-Values P>0.05

A continuacion volvemos a realizar el entrenamiento DESDE 0. Esta vez estamos seguro de que todas las variables que tenemos son significativas para nuestro modelo.

```
algoPrediccion.realizarEntrenamientoCompleto()  
algoPrediccion.resultadoDeEntrenamiento()
```

✓ 0.0s

Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
--------	------------	----------------	-----------------------------

0	4733	4910.86	177.86	3.76
1	6424	7590.37	1166.37	18.16
2	5510	6137.91	627.91	11.40
3	8770	10292.28	1522.28	17.36
4	4493	5248.32	755.32	16.81
...
10783	1289	1127.01	161.99	12.57
10784	3435	3516.20	81.20	2.36
10785	3847	5266.55	1419.55	36.90
10786	8168	6976.04	1191.96	14.59
10787	1917	2311.11	394.11	20.56

10788 rows × 4 columns

```

def realizarEntrenamientoCompleto(self):
    self.divisionDeConjuntos()
    self.entrenar()
    self.prediccion()

def divisionDeConjuntos(self):
    self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(self.X, self.Y, test_size=0.2, random_state=0)

def entrenar(self):
    self.algoritmo.entrenar(self)

def prediccion(self):
    self.y_pred = self.algoritmo.prediccion(self)

algoPrediccion.RealizarEntrenamientoCompleto()
algoPrediccion.resultadoDeEntrenamiento()

```

Actual Prediccion Error Absoluto Error porcentual absoluto %

0	4733	4910.86	177.86	3.76
1	6424	7590.27	1166.27	15.16

```

class RegresionLineal(TipoDeRegresion):
    regressor = LinearRegression()
    def entrenar(self, RegresionModelo):
        self.regressor.fit(RegresionModelo.X_train, RegresionModelo.y_train)

    def prediccion(self, RegresionModelo):
        y_pred = self.regressor.predict(RegresionModelo.X_test).flatten()
        return y_pred
    def nombreDeRegresion(self):
        return "Regresion Lineal"

```

```

def resultadoDeEntrenamiento(self):
    y_test = np.ravel(self.y_test) #EVITA ERRORES.

    pd.set_option('display.float_format', lambda x: '{:.2f}'.format(x)) #PARA EVITAR LA NOTACION CIENTIFICA
    dff = pd.DataFrame({'Actual': y_test, 'Prediccion':self.y_pred})
    diferencia = abs(dff['Prediccion']-dff['Actual'])
    diferenciaPorcentual=abs(((dff['Prediccion']*100)/(dff['Actual']))-100)

    dff = pd.DataFrame([{'Actual': y_test,
                        'Prediccion':self.y_pred,
                        'Error Absoluto': diferencia,
                        'Error porcentual absoluto %':diferenciaPorcentual
                       }])
    return dff

```

```

algoPrediccion.conclusiones()

```

✓ 0.0s

	Media	Error %	Efectividad %
Mean Absolute Error	737.78	18.76	81.24
Root Mean Squared Error	1117.33	28.41	71.59
Mean Squared Error	1248435.26	NaN	NaN

Volvemos a obtener los mismos Resultados!!

El objetivo de la eliminación hacia atrás era reducir el modelo de regresión y quedarnos solo con las variables más importantes y relevantes , en simples palabras reducir la complejidad. Inicialmente se contaban 25 columnas que representan las variables independientes (X), pero después de aplicar la eliminación hacia atrás,se eliminaron 2 columnas y finalmente nos quedamos con 23.

Lo interesante es que, a pesar de haber eliminado esas dos columnas, los resultados de predicción no cambiaron. Esto nos indica que las dos columnas que se eliminó no eran tan importantes para hacer predicciones precisas.

Podemos concluir que las variables en esas dos columnas no estaban aportando información significativa para predecir la variable dependiente.

Esto significa que se puede simplificar el modelo al eliminar esas dos columnas, sin comprometer la calidad de las predicciones.

REGRESION POLINOMICA

Recordemos que cada vez que se ejecutaba el metodo conclusion se realizaba un guardado de un archivo que mantenía el estado del objeto RegresionModelo

```
self.guardarArchivo()

df = pd.DataFrame(data, index=['Mean Absolute Error'])
return df

def guardarArchivo(self):
    with open('variable.pkl', 'wb') as f:
        pickle.dump(self, f)
```

Como pasamos a otro archivo jupiter notebook tenemos que recuperar el estado de ese objeto para si evitar realizar los pasos como por ejemplo la separacion de variables independientes de las dependientes , eliminacion hacia atrás etc. Asi que lo siguiente que se hace es cargar ese archivo que previamente se guardo. Recordemos que lo ultimo que se ejecuto en el el archivo “02- Separacion y eliminacion de datos + regresion lineal.ipynb “ fue el metodo conclusion() Asi que ya estamos asegurados que vamos a recibir el objeto con los ultimos cambios realizados.

The screenshot shows a Jupyter Notebook interface. On the left, the 'EXPLORER' sidebar lists several notebooks: 'PARADIGMASDEPROGRAMACIONUNSAM-TPS', '_pycache__', 'svrLinealCapturas', 'TP-1 Diamantes (Algoritmos de regresion)', '01 - Analisis de datos.ipynb', '02 - Separacion y eliminacion de datos + Regresion Lineal.ipynb' (which is currently selected), '03 - Regresion Polinomica.ipynb', '04 - Regresión con máquinas de soporte vectorial (SVR).ipynb', '05 - Regresion con Arboles de decision copy.ipynb', '06 - Regresion con RANDOM FOREST .ipynb', and '07 - Regresion con KNN.ipynb'. On the right, a code cell contains the following Python code:

```
import pickle

# Cargar la variable desde el archivo
with open('variable.pkl', 'rb') as f:
    algoPrediccion = pickle.load(f)
from algoritmos import RegresionConArboles
from sklearn.ensemble import RandomForestRegressor
```

El siguiente paso es cambiar de algoritmo . Hasta ahora sigue con el modelo de regresion lineal. Hacer esto es muy sencillo. Como se puede ver abajo

```

from algoritmos import RegresionPolinomica
algoPrediccion.algoritmo = RegresionPolinomica(2)
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()

```

[51] ✓ 0.5s

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4677.22	55.78	1.18
1	6424	7244.28	820.28	12.77
2	5510	5227.90	282.10	5.12
3	8770	11542.53	2772.53	31.61
4	4493	5094.43	601.43	13.39
...
10783	1289	1196.04	92.96	7.21
10784	3435	3144.38	290.62	8.46
10785	3847	4766.58	919.58	23.90
10786	8168	8011.78	156.22	1.91
10787	1917	2034.19	117.19	6.11

10788 rows × 4 columns

Bueno ya obtuvimos los resultados con solo ejecutar 2 metodos!! Aca se empieza a ver la ventaja de tenerlo todo encapsulado en un objeto usando el paradigma de objetos justamente. La gran ventaja tambien es la limpieza que maneja los archivos jupiter notebook. Si bien podemos llenarlo de codigo a la larga se complica mantenerlo.

Volviendo a lo que importa , De entrada vemos que le estamos asignando un nuevo algoritmo de regresion ,en este caso el modelo de regresion Polinomica y le estamos definiendo en el constructor un Grado 2

```
algoPrediccion.algoritmo = RegresionPolinomica(2)
```

```

class RegresionPolinomica(TipoDeRegresion) :

    regressor = LinearRegression()
    grado = None
    X_poly = None

    def __init__(self, grado ):
        self = self
        self.grado = grado

    def nombreDeRegresion(self):
        return "Regresion Polinomica"

    def crearPolinomio(self,RegresionModelo) :
        poly_reg = PolynomialFeatures(degree=self.grado)
        self.X_poly = poly_reg.fit_transform(RegresionModelo.X_train)

    def entrenar(self, RegresionModelo):
        self.crearPolinomio(RegresionModelo)
        self.regressor.fit(self.X_poly, RegresionModelo.y_train)

    def prediccion(self, RegresionModelo):
        poly_reg = PolynomialFeatures(degree=self.grado)
        X_poly_test = poly_reg.fit_transform(RegresionModelo.X_test)
        y_pred = self.regressor.predict(X_poly_test).flatten()

        return y_pred

```

La elección del grado no fue trivial sino que se utilizó un método estadístico de validación cruzada para determinar con qué grado el modelo debería trabajar. En este TP se escogió por el de Shuffle Splits. Sin embargo el tiempo de respuesta fue exageradamente lento. No se con exactitud en qué momento se generó el resultado para el MSE de grado 3 pero lo que si estoy seguro es que en 34 minutos no fue capaz de generarme todos los resultados. Esto se debe a que se está trabajando con un conjunto grande de datos que contiene más de 50.000 filas.

```

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

ss = ShuffleSplit(n_splits=40, test_size=0.2, random_state=14)

for d in range(1, 9):
    poly = PolynomialFeatures(degree=d)
    X_now = poly.fit_transform(algoPrediccion.X)
    model = LinearRegression()
    scores = cross_val_score(model, X_now, algoPrediccion.Y, scoring='neg_mean_squared_error', cv=ss, n_jobs=1)
    mse = np.mean(np.abs(scores))
    print(f'Polinomio de Grado {d} MSE: {mse:.5f}, STD: {np.std(scores):.5f}')

```

34m 23.6s

```

Polinomio de Grado 1 MSE: 1274747.57402, STD: 51802.13526
Polinomio de Grado 2 MSE: 491616.12096, STD: 26858.80764
Polinomio de Grado 3 MSE: 47391925945444.68750, STD: 101408189505716.50000

```

Sin embargo con los resultados mostrados ya nos basta. Podemos ver que obtenemos el minimo con un grado igual a 2. Con un grado 3 igual a el MSE comienza aumentar demasiado. Asi que se opta por el grado 2 para la regresion Polinomica porque ademas es el que mejor resultado nos dio.

Volvamos a lo importante

Regresion Polinomica

```

from algoritmos import RegresionPolinomica
algoPrediccion.algoritmo = RegresionPolinomica(2)
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()

```

✓ 0.5s

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4677.22	55.78	1.18
1	6424	7244.28	820.28	12.77

Primero se asigna el modelo de regresion Polinomica de grado 2

Luego se ejecuta el metodo realizarEntrenamientoSinDivisionDeConjuntos()

```
def realizarEntrenamientoCompleto(self) :  
    self.divisionDeConjuntos()  
    self.entrenar()  
    self.prediccion()  
  
    def realizarEntrenamientoSinDivisionDeConjuntos(self):  
        self.entrenar()  
        self.prediccion()
```

Como se puede ver es muy similar al realizarEntrenamientoCompleto() que ya se explico cuando se termino de realizar la eliminacion hacia atrás. La diferencia es que este metodo no hace division de conjuntos porque ya fue realizado previamente !! No hace falta hacerla devuelta. Sin embargo como cambiamos de algoritmos obviamente tenemos que ejecutar los metodos entrenar y prediccion devuelta para asi obtener los resultados del nuevo algoritmo que se va a utilizar.

Veamos los resultados

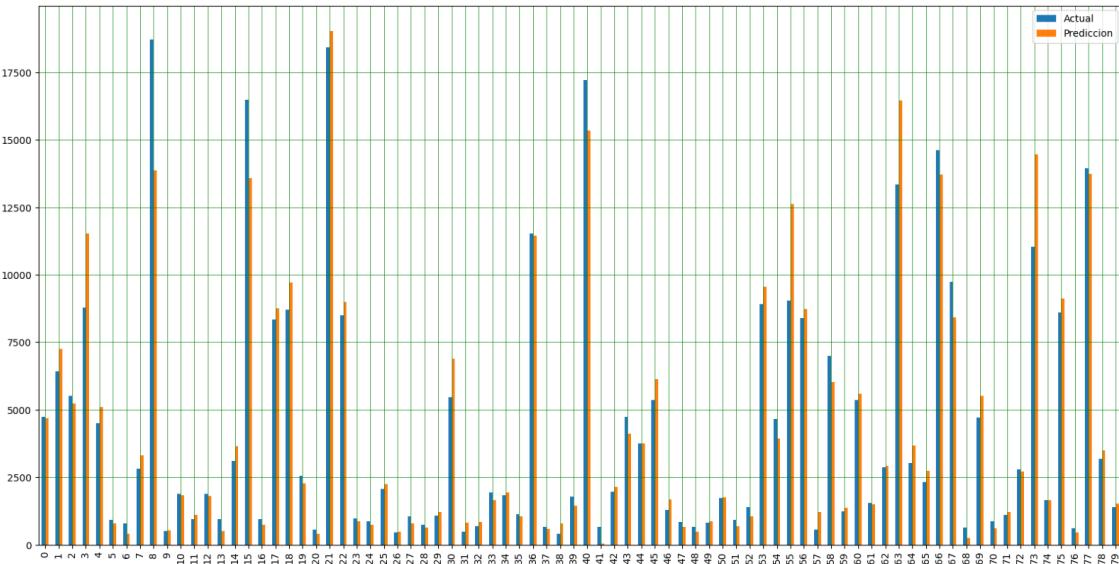
```
algoPrediccion.resultadoDeEntrenamiento().head(20)
```

✓ 0.6s

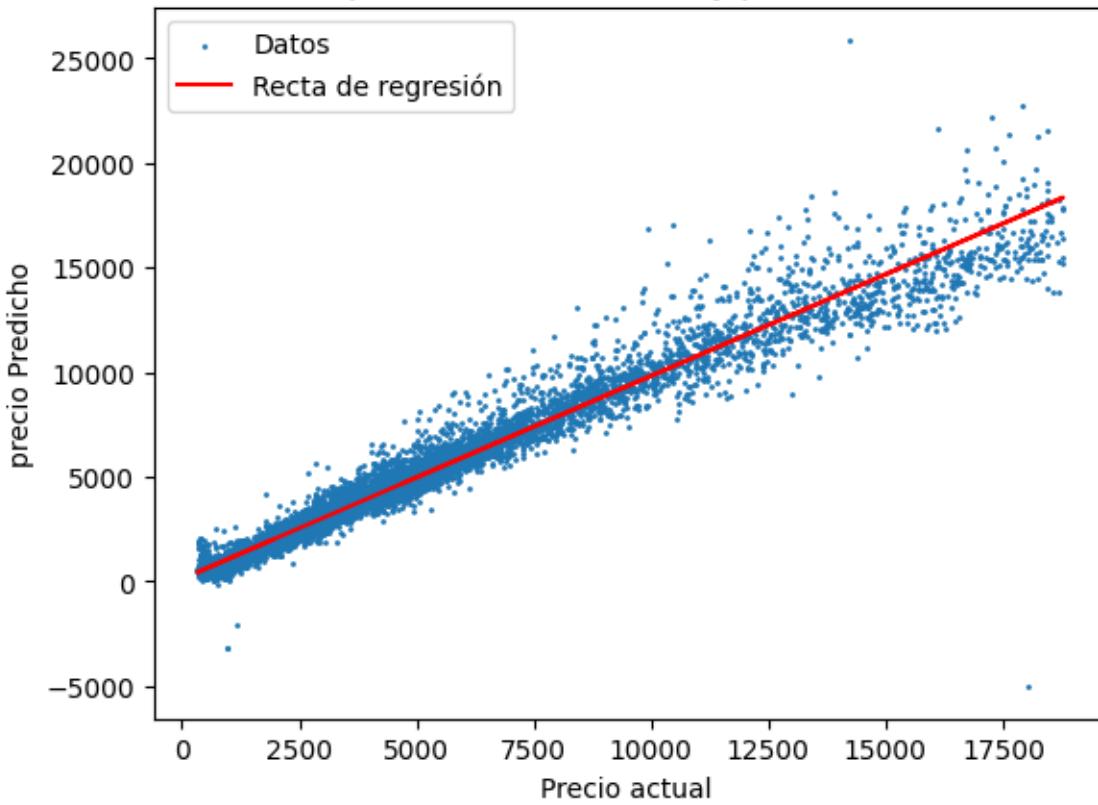
	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4677.22	55.78	1.18
1	6424	7244.28	820.28	12.77
2	5510	5227.90	282.10	5.12
3	8770	11542.53	2772.53	31.61
4	4493	5094.43	601.43	13.39
5	918	801.83	116.17	12.65
6	789	397.18	391.82	49.66
7	2823	3311.03	488.03	17.29
8	18705	13853.99	4851.01	25.93
9	507	524.08	17.08	3.37
10	1880	1831.72	48.28	2.57
11	935	1094.38	159.38	17.05
12	1872	1798.71	73.29	3.92
13	956	517.02	438.98	45.92
14	3111	3647.05	536.05	17.23
15	16499	13585.55	2913.45	17.66
16	954	750.48	203.52	21.33

```
algoPrediccion.todasLasComparacionesDeActualPrediccion()
```

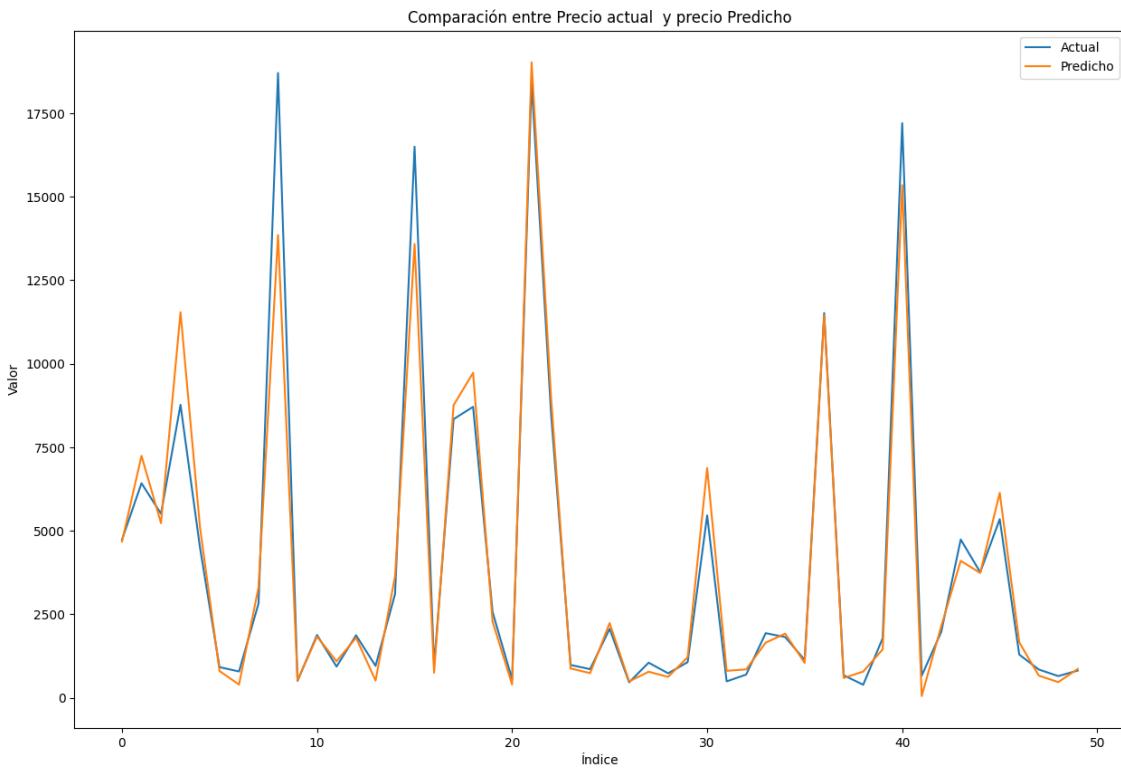
Observemos los resultados graficos



Comparación Precio actual y precio Predicho



Vemos aca que a diferencia del modelo lineal la dispersión aquí es menos “violenta”. Incluso con aquellos precios actuales y predichos mas grandes. Esto ya nos da indicios de que efectivamente se obtuvieron mejores resultados y que la conclusion deberia ser notablemente mejor



Por ultimo veamos la conclusion

```
algoPrediccion.conclusiones()
```

✓ 0.3s

	Media	Error %	Efectividad %
Mean Absolute Error	412.10	10.48	89.52
Root Mean Squared Error	729.42	18.55	81.45
Mean Squared Error	532057.54	NaN	NaN

Como vemos aca los resultados son muchisimo mejores!!! Por supuesto la diferencia es notable , incluso con el RMSE el resultado es bueno!

Pasemos al siguiente modelo de regresion :

REGRESION CON MAQUINAS DE SOPORTE VECTORIAL

```
import pickle

# Cargar la variable desde el archivo
with open('variable.pkl', 'rb') as f:
    algoPrediccion = pickle.load(f)
from algoritmos import RegresionConArboles
from sklearn.ensemble import RandomForestRegressor

✓ 1.3s
```

Regresión con máquinas de soporte vectorial (SVR)

Kernel : rbf

```
from sklearn.svm import SVR
from algoritmos import RegresionSVR

svr_rbf = SVR(kernel="rbf", epsilon=0.1)
svr_lin = SVR(kernel="linear", gamma="auto")
svr_poly = SVR(kernel="poly", gamma="auto", degree=2, epsilon=0.1, coef0=1)

algoPrediccion.algoritmo = RegresionSVR(svr_rbf)
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()

print(svr_poly.kernel)
✓ 40.1s
```

Comenzamos con el kernel rbf

```

class RegresionSVR(TipoDeRegresion):
    sc_X = StandardScaler()
    sc_y = StandardScaler()

    regression = None

    def __init__(self, kernel ):
        self = self
        self.regression = kernel

    def entrenar(self, RegresionModelo):
        X = self.sc_X.fit_transform(RegresionModelo.X_train)
        y = self.sc_y.fit_transform(RegresionModelo.y_train.values.reshape(-1, 1))
        self.regression.fit(X, y)

    def prediccion(self, RegresionModelo):
        X_test = self.sc_X.transform(RegresionModelo.X_test)
        y_pred = self.regression.predict(X_test)
        y_pred = self.sc_y.inverse_transform(y_pred.reshape(-1, 1))
        return y_pred.flatten()

    def nombreDeRegresion(self):
        return f"Regresion SVR con el Kernel : {self.regression.kernel}"

```

C

El strategy de RegresionSVR es valido para los 3 tipos de kernel.

Veamos los resultados :

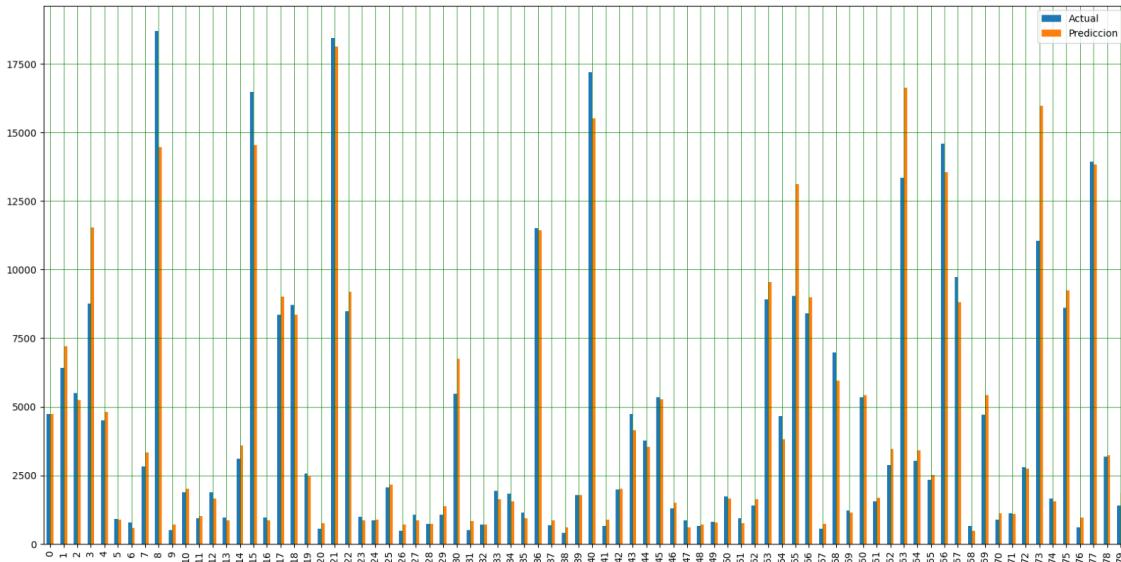
```
algoPrediccion.resultadoDeEntrenamiento()
```

✓ 0.0s

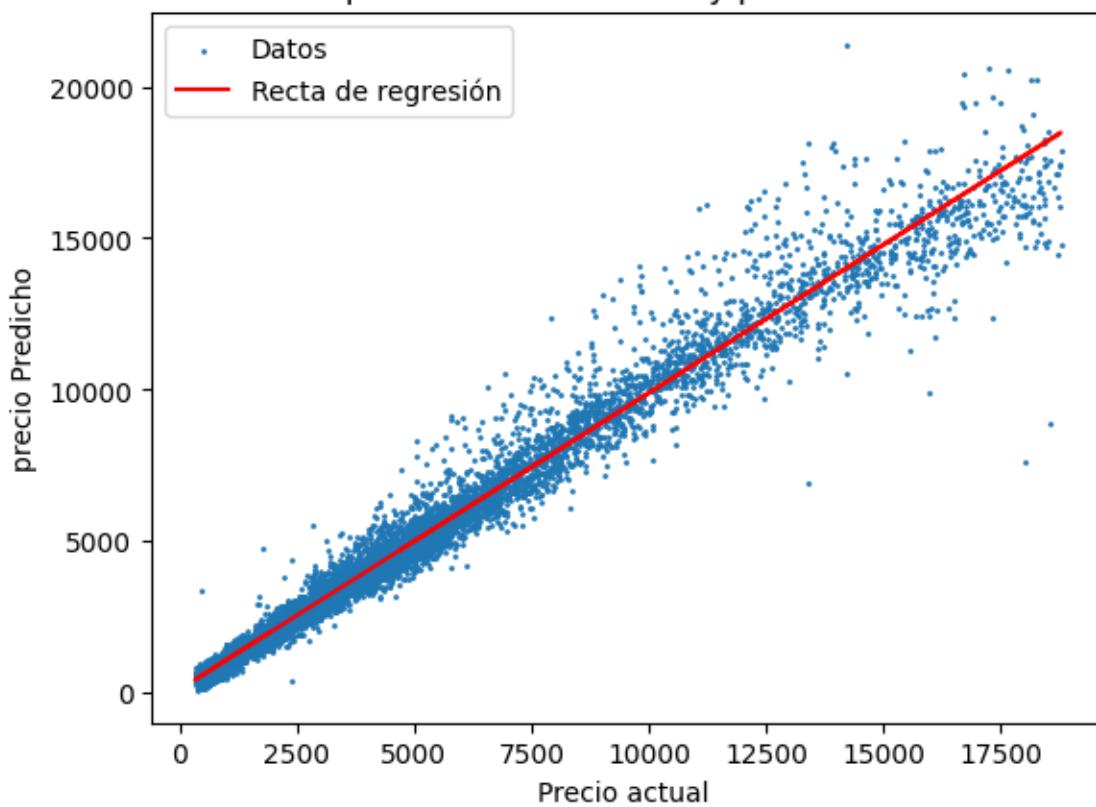
	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4757.15	24.15	0.51
1	6424	7317.62	893.62	13.91
2	5510	5311.71	198.29	3.60
3	8770	11381.24	2611.24	29.77
4	4493	4820.99	327.99	7.30
...
10783	1289	1328.52	39.52	3.07
10784	3435	3290.36	144.64	4.21
10785	3847	4618.08	771.08	20.04
10786	8168	7666.99	501.01	6.13
10787	1917	1776.81	140.19	7.31

10788 rows × 4 columns

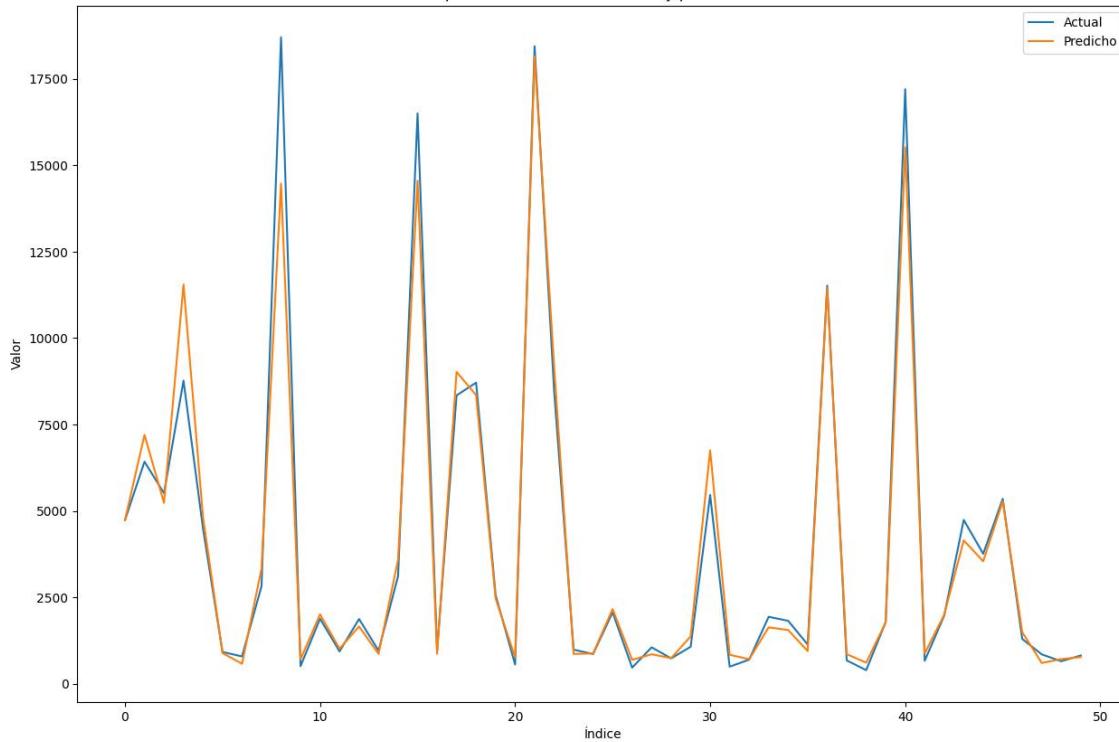
Veamos los graficos



Comparación Precio actual y precio Predicho



Comparación entre Precio actual y precio Predicho



```
algoPrediccion.conclusiones()
```

✓ 0.0s

	Media	Error %	Efectividad %
Mean Absolute Error	348.75	8.87	91.13
Root Mean Squared Error	623.04	15.84	84.16
Mean Squared Error	388176.90	NaN	NaN

SVR : Kernel : Poly

```
algoPrediccion.algoritmo = RegresionSVR(svr_poly)
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()
```

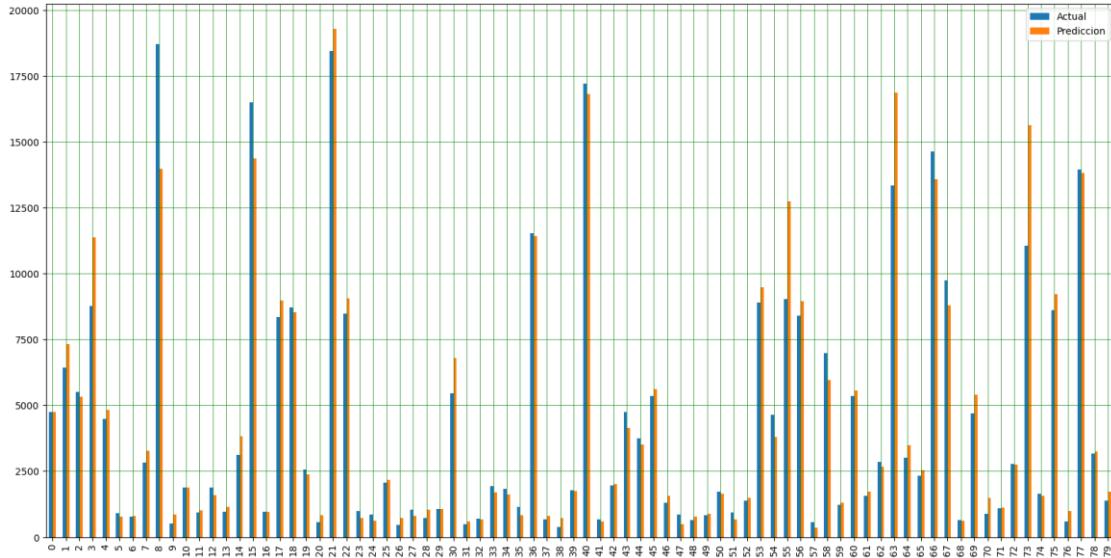
	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4757.15	24.15	0.51
1	6424	7317.62	893.62	13.91
2	5510	5311.71	198.29	3.60
3	8770	11381.24	2611.24	29.77
4	4493	4820.99	327.99	7.30
...
10783	1289	1328.52	39.52	3.07
10784	3435	3290.36	144.64	4.21
10785	3847	4618.08	771.08	20.04
10786	8168	7666.99	501.01	6.13
10787	1917	1776.81	140.19	7.31

[10788 rows x 4 columns]

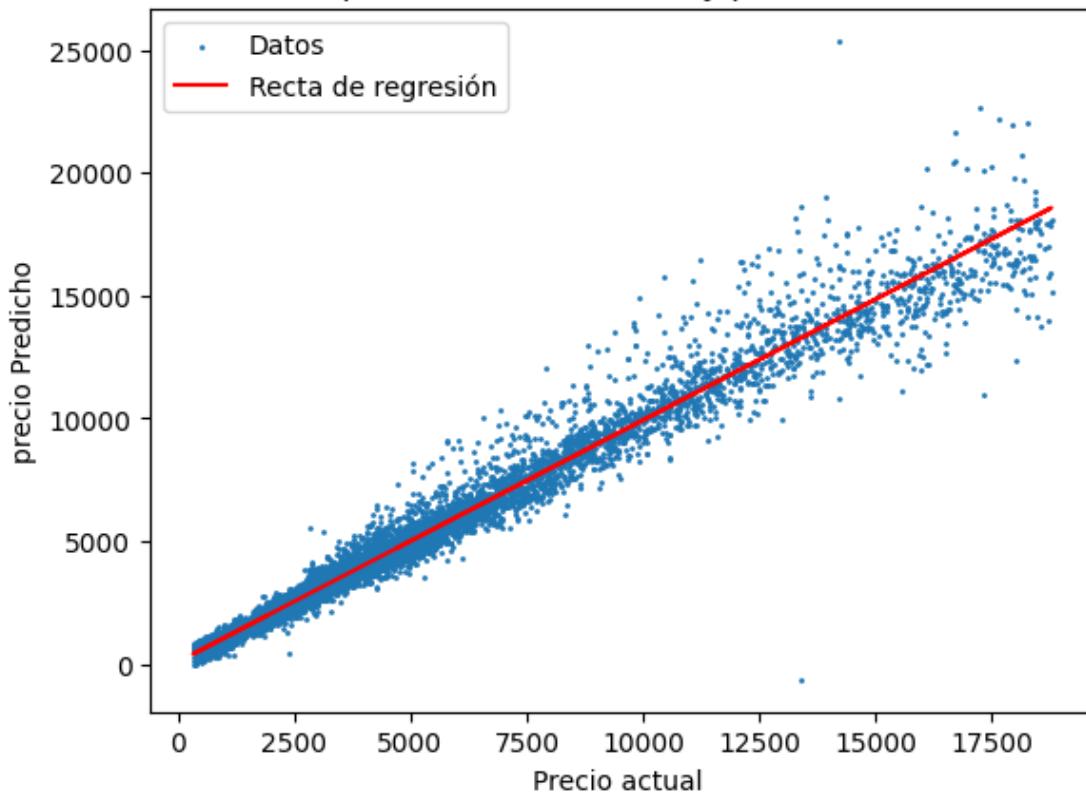
```
algoPrediccion.resultadoDeEntrenamiento().describe()
```

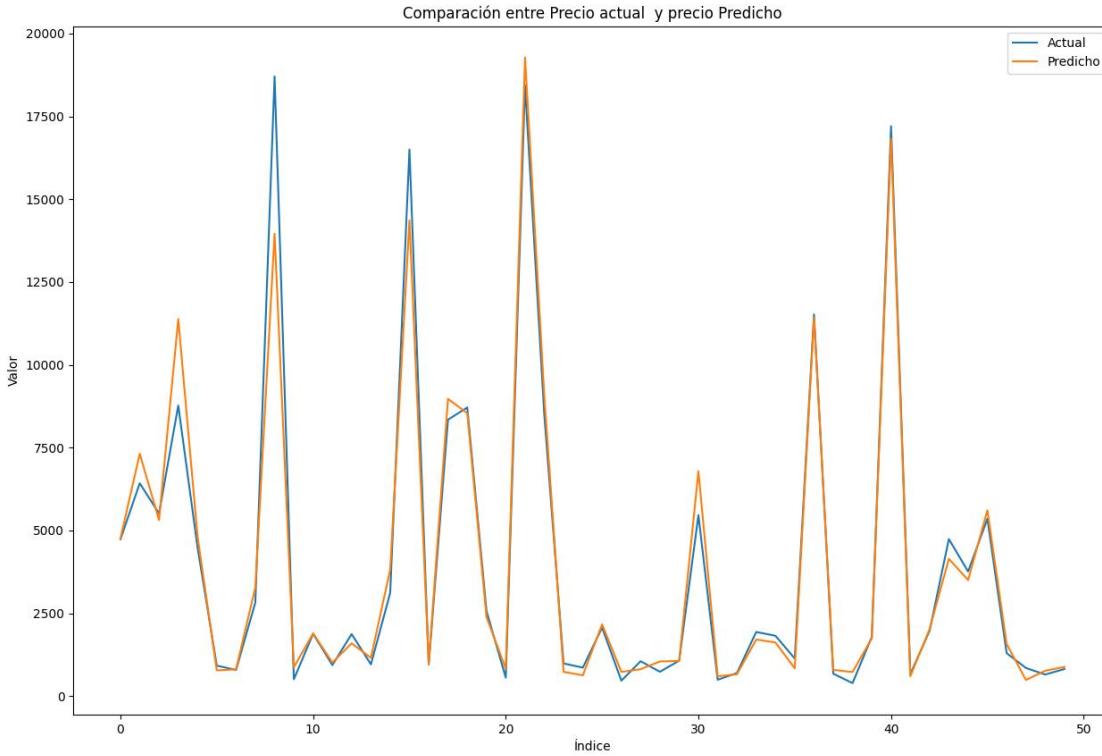
	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
count	10788.00	10788.00	10788.00	10788.00
mean	3929.21	3950.49	350.25	13.58
std	3981.60	3969.61	528.82	15.10
min	326.00	-647.20	0.00	0.00
25%	947.50	955.45	92.95	3.82
50%	2398.00	2405.16	206.03	8.64
75%	5311.25	5417.35	381.79	17.36
max	18787.00	25370.72	14034.20	162.61

```
algoPrediccion.todasLasComparacionesDeActualPrediccion()
```



Comparación Precio actual y precio Predicho





```
algoPrediccion.conclusiones()
```

	Media	Error %	Efectividad %
Mean Absolute Error	350.25	8.91	91.09
Root Mean Squared Error	634.27	16.13	83.87
Mean Squared Error	402293.89	NaN	NaN

SVR :Kernel Linear

```
from sklearn.svm import SVR
from algoritmos import RegresionSVR

algoPrediccion.algoritmo = RegresionSVR(svr_lin)
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()
```

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4944.17	211.17	4.46
1	6424	7410.62	986.62	15.36
2	5510	6013.92	503.92	9.15
3	8770	9851.66	1081.66	12.33
4	4493	4795.63	302.63	6.74

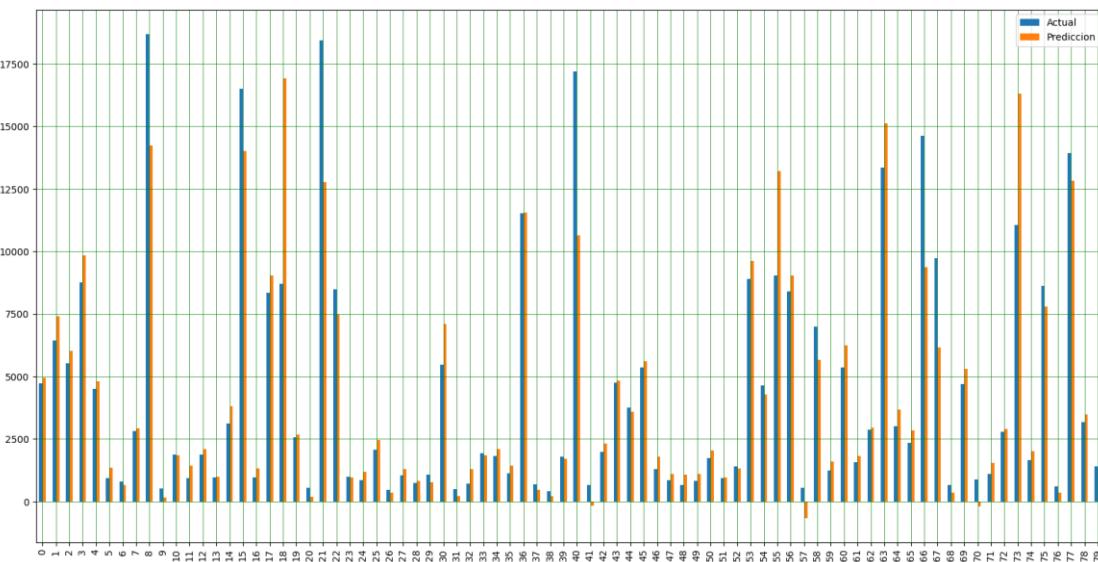
...
10783	1289	1296.92	7.92		0.61
10784	3435	3525.56	90.56		2.64
10785	3847	5062.60	1215.60		31.60
10786	8168	6259.81	1908.19		23.36
10787	1917	1903.30	13.70		0.71

[10788 rows x 4 columns]

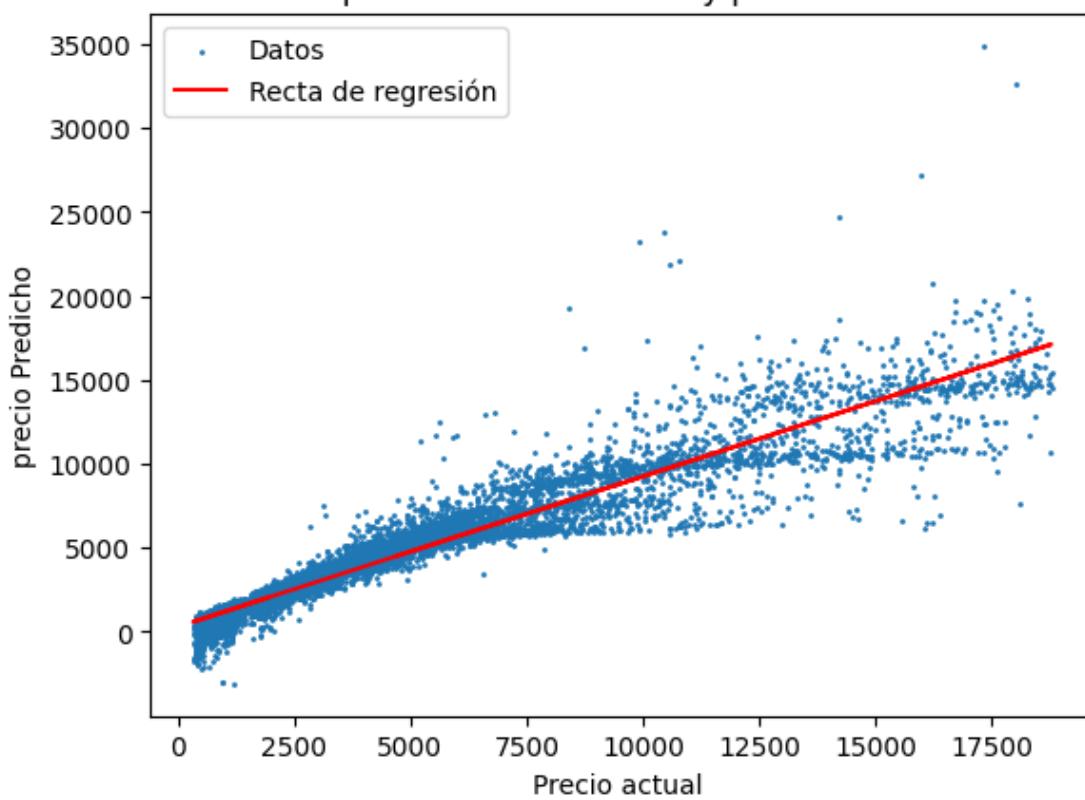
algoPrediccion.resultadoDeEntrenamiento().describe()

	Actual	Prediccion	Error Absoluto	Error porcentual	absoluto %
count	10788.00	10788.00	10788.00		10788.00
mean	3929.21	3821.98	658.69		28.10
std	3981.60	3725.19	966.50		43.67
min	326.00	-3100.12	0.04		0.00
25%	947.50	1066.60	186.57		7.15
50%	2398.00	2580.35	365.89		15.83
75%	5311.25	5620.78	694.20		31.91
max	18787.00	34884.32	17555.32		591.26

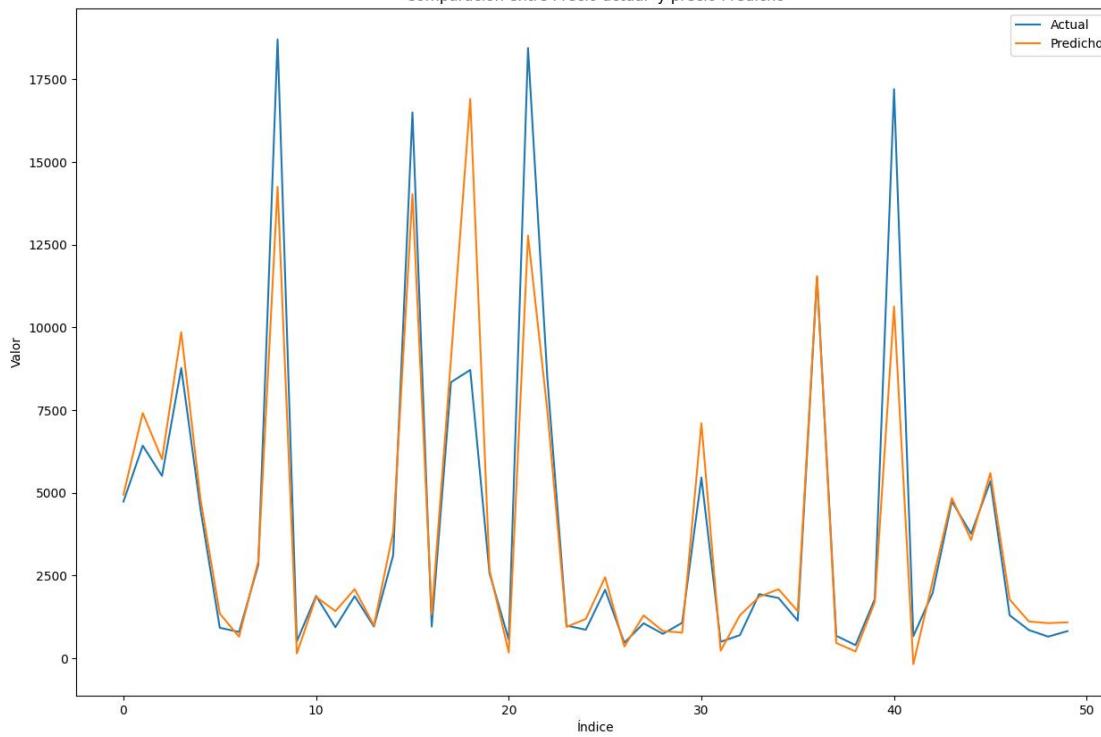
algoPrediccion.todasLasComparacionesDeActualPrediccion()



Comparación Precio actual y precio Predicho



Comparación entre Precio actual y precio Predicho



```
algoPrediccion.conclusiones()
```

	Media	Error %	Efectividad %
Mean Absolute Error	658.69	16.75	83.25
Root Mean Squared Error	1169.58	29.74	70.26
Mean Squared Error	1367909.55	NaN	NaN

REGRESION CON ARBOLES DE DESICION

```
import pickle

# Cargar la variable desde el archivo
with open('variable.pkl', 'rb') as f:
    algoPrediccion = pickle.load(f)
from algoritmos import RegresionConArboles
from sklearn.ensemble import RandomForestRegressor
```

Regresión con Árboles de Decisión

```
from algoritmos import RegresionConArboles
from sklearn.tree import DecisionTreeRegressor
algoPrediccion.algoritmo = RegresionConArboles(DecisionTreeRegressor(random_state=0))
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento().head(30)
```

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4620.00	113.00	2.39
1	6424	7036.00	612.00	9.53
2	5510	4404.00	1106.00	20.07
3	8770	11688.00	2918.00	33.27
4	4493	4678.00	185.00	4.12
5	918	918.00	0.00	0.00
6	789	764.00	25.00	3.17
7	2823	3011.00	188.00	6.66
8	18705	13267.00	5438.00	29.07
9	507	513.00	6.00	1.18
10	1880	1628.00	252.00	13.40
11	935	848.00	87.00	9.30
12	1872	1767.00	105.00	5.61
13	956	956.00	0.00	0.00
14	3111	3959.00	848.00	27.26
15	16499	12617.00	3882.00	23.53
16	954	918.00	36.00	3.77
17	8342	10232.00	1890.00	22.66
18	8711	7854.00	857.00	9.84
19	2567	2310.00	257.00	10.01
20	555	501.00	54.00	9.73

21	18445	15941.00	2504.00	13.58
22	8486	9478.00	992.00	11.69
23	984	767.00	217.00	22.05
24	858	844.00	14.00	1.63
25	2066	2510.00	444.00	21.49
26	464	500.00	36.00	7.76
27	1052	946.00	106.00	10.08
28	733	701.00	32.00	4.37
29	1068	1089.00	21.00	1.97
30	5461	6250.00	789.00	14.45

Esta clase es reutilizada tanto para la Regresión con árboles de decisión como Para el Random Forest . Acá lo que importa es el tipo de regresor que se le declare a este objeto al momento de asignar el modelo de regresión al objeto RegresiónModel o . Por Ejemplo

```
algoPrediccion.algoritmo =
    RegresionConArboles(DecisionTreeRegressor(random_state = 0))
```

Para este caso va el objeto RegresionConArboles va a realizar el entrenamiento y predicciones según el modelo de regresión con árboles de decisión

```
algoPrediccion.algoritmo =
    RegresionConArboles(RandomForestRegressor(n_estimators = 300, random_state = 0))
```

Para este caso va el objeto RegresionConArboles va a realizar el entrenamiento y predicciones según el modelo de regresión forest regresor

```

class RegresionConArboles(TipoDeRegresion):

    regression = None

    def __init__(self, tipoDeArbolDeRegression):
        self = self
        self.regression = tipoDeArbolDeRegression

    regression = DecisionTreeRegressor(random_state = 0)

    def entrenar(self, RegresionModelo):
        self.regression.fit(RegresionModelo.X_train, RegresionModelo.y_train)

    def prediccion(self, RegresionModelo):
        y_pred = self.regression.predict(RegresionModelo.X_test)
        return y_pred

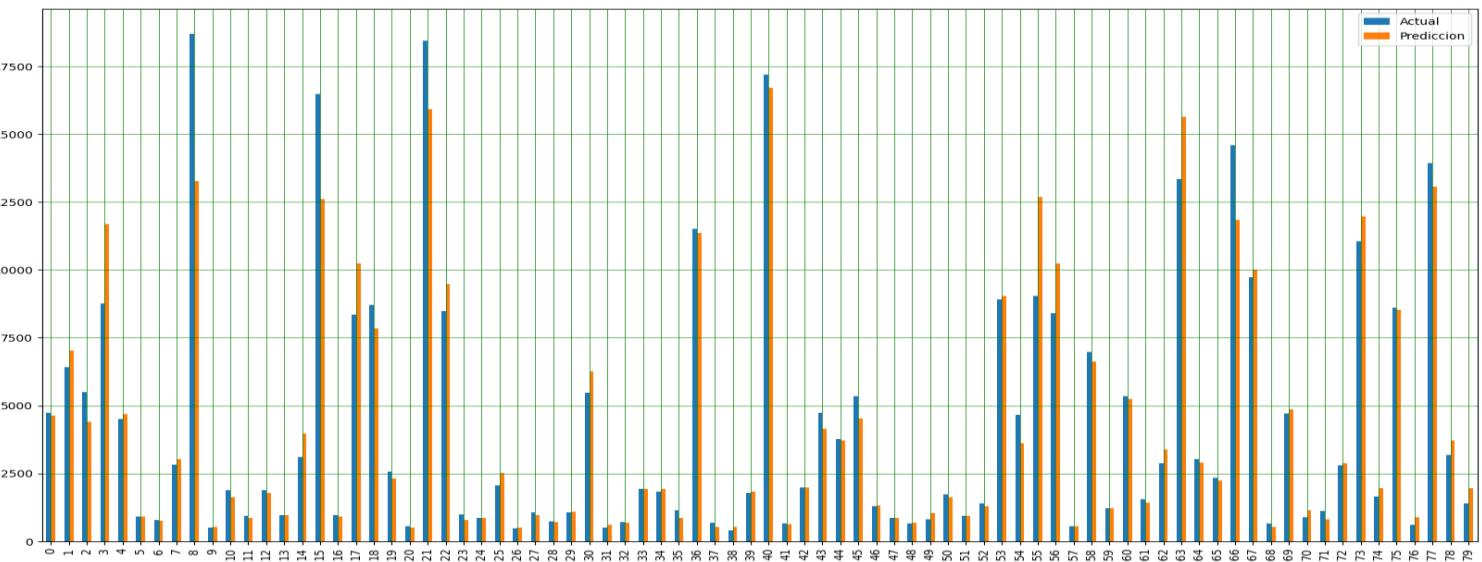
    def nombreDeRegresion(self):
        tipoDeArbolDiccionario = {
            "RandomForestRegressor": "Regresion con Random forest",
            "DecisionTreeRegressor": "Regresion con Arboles de desicion"}
        tipo = self.regression.__class__.__name__
        return tipoDeArbolDiccionario[tipo]

```

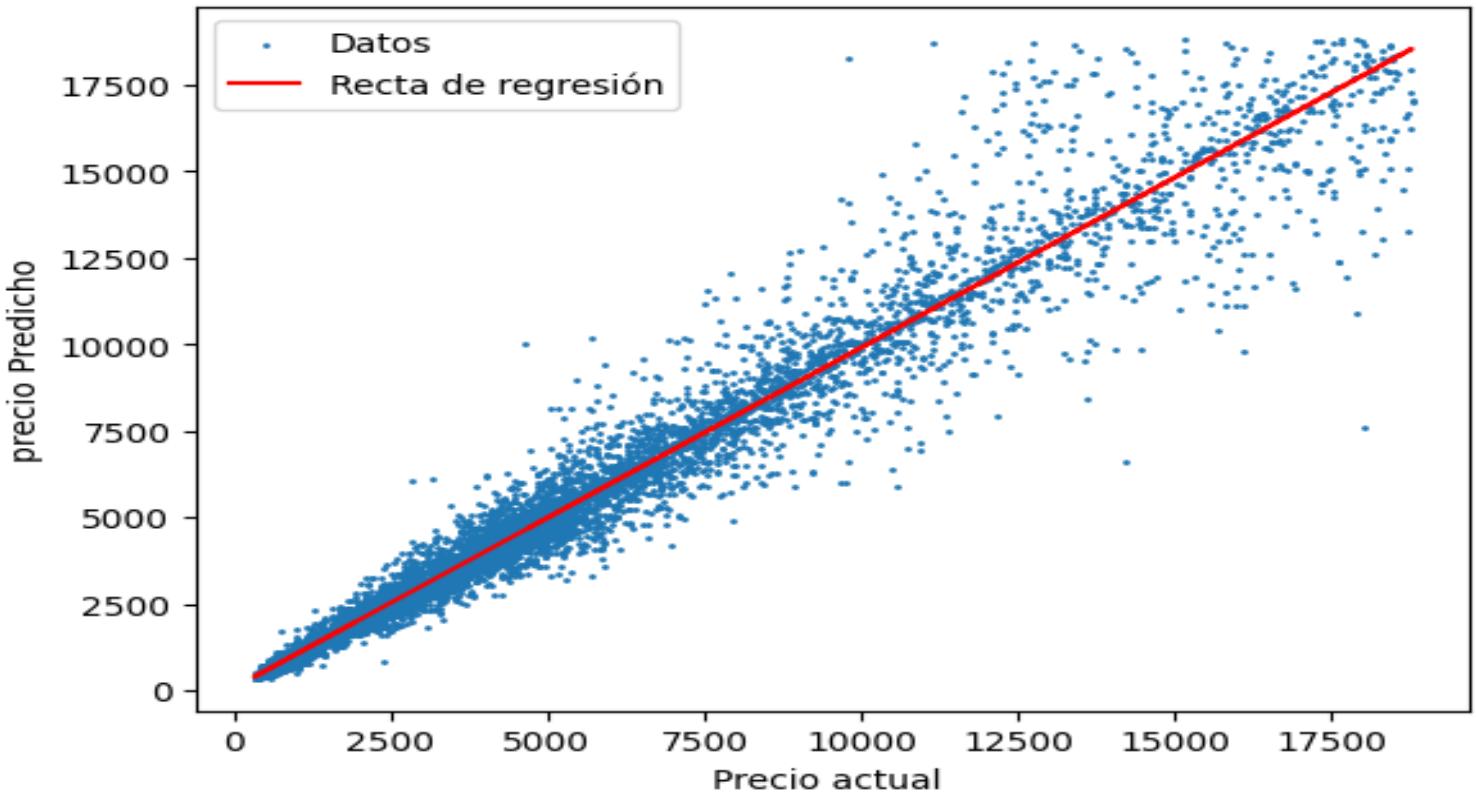
```
algoPrediccion.resultadoDeEntrenamiento().describe()
```

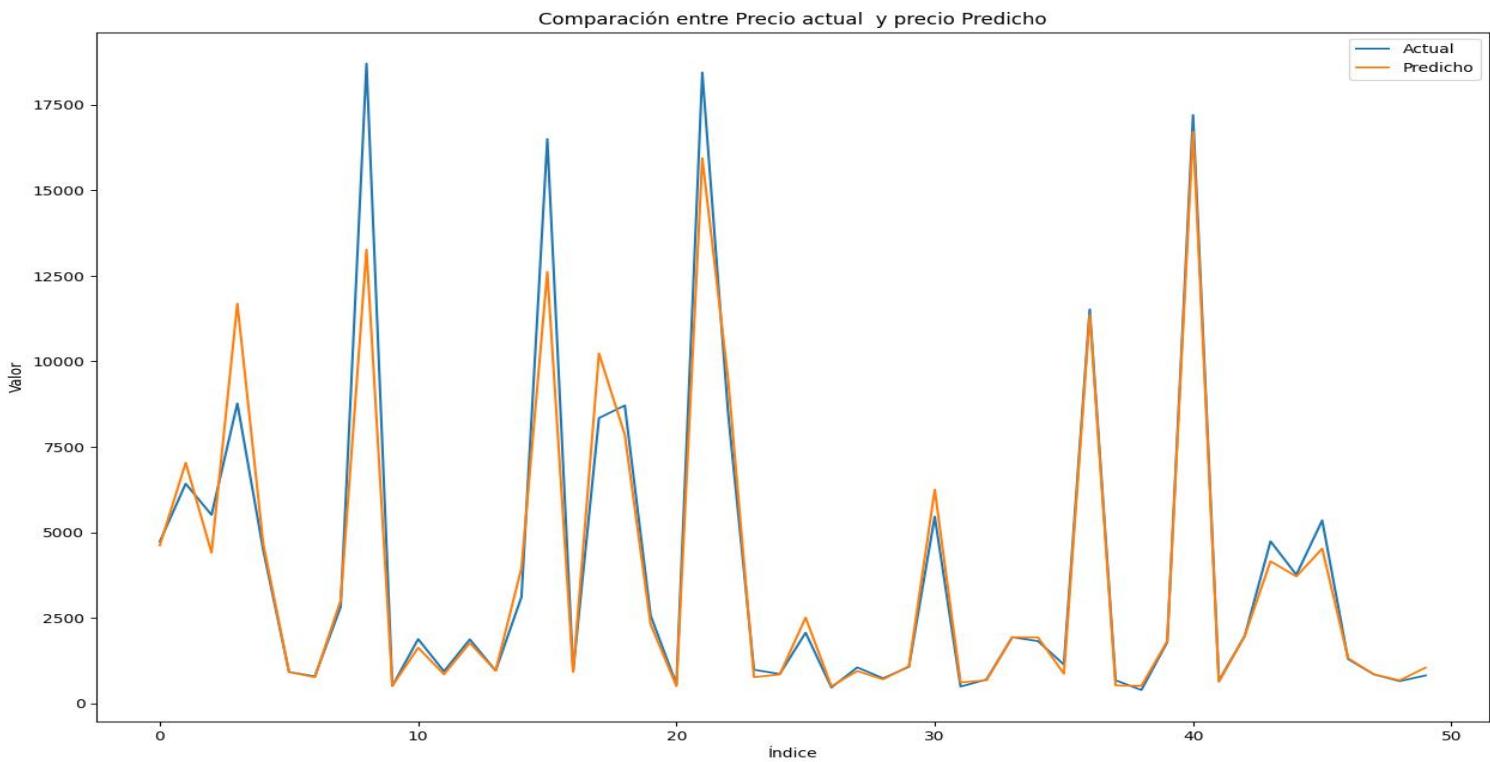
	Actual	Prediccion	Error	Absoluto	Error porcentual	absoluto %
count	10788.00	10788.00		10788.00		10788.00
mean	3929.21	3933.01		363.37		9.55
std	3981.60	3985.64		657.12		9.98
min	326.00	338.00		0.00		0.00
25%	947.50	945.00		48.00		2.08
50%	2398.00	2398.00		146.00		6.46
75%	5311.25	5321.75		376.00		13.96
max	18787.00	18823.00		10424.00		135.25

```
algoPrediccion.todasLasComparacionesDeActualPrediccion()
```



Comparación Precio actual y precio Predicho





```
algoPrediccion.conclusiones()
```

	Media	Error %	Efectividad %
Mean Absolute Error	363.37	9.24	90.76
Root Mean Squared Error	750.87	19.09	80.91
Mean Squared Error	563800.02	NaN	NaN

Regresion con RANDOM FOREST

```
import pickle

# Cargar la variable desde el archivo
with open('variable.pkl', 'rb') as f:
    algoPrediccion = pickle.load(f)
from algoritmos import RegresionConArboles
from sklearn.ensemble import RandomForestRegressor
```

Regresión con ForestRegressor

```

algoPrediccion.algoritmo = RegresionConArboles(RandomForestRegressor(n_estimators = 300, random_state = 0))
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()

```

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4669.43	63.57	1.34
1	6424	7053.27	629.27	9.80
2	5510	5223.09	286.91	5.21
3	8770	11088.87	2318.87	26.44
4	4493	4709.23	216.23	4.81
...
10783	1289	1330.44	41.44	3.21
10784	3435	3346.93	88.07	2.56
10785	3847	4282.07	435.07	11.31
10786	8168	8500.22	332.22	4.07
10787	1917	2129.50	212.50	11.09

[10788 rows x 4 columns]

```

algoPrediccion.resultadoDeEntrenamiento().describe()

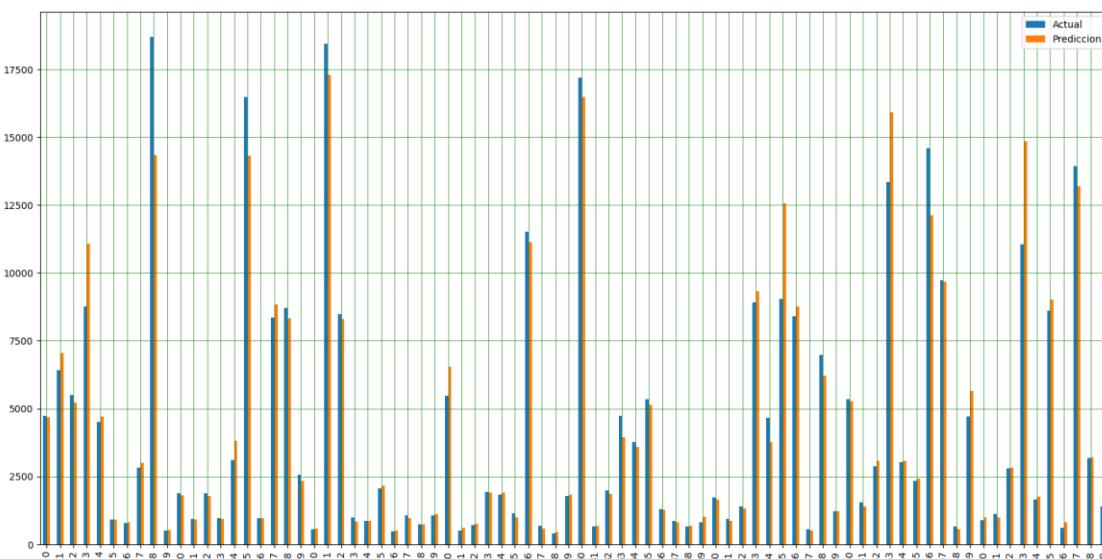
```

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
count	10788.00	10788.00	10788.00	10788.00
mean	3929.21	3933.68	278.89	7.48
std	3981.60	3946.96	474.88	7.17
min	326.00	360.38	0.01	0.00
25%	947.50	931.07	44.76	2.45
50%	2398.00	2433.13	111.00	5.38
75%	5311.25	5305.31	301.33	10.35
max	18787.00	17878.26	5683.08	93.71

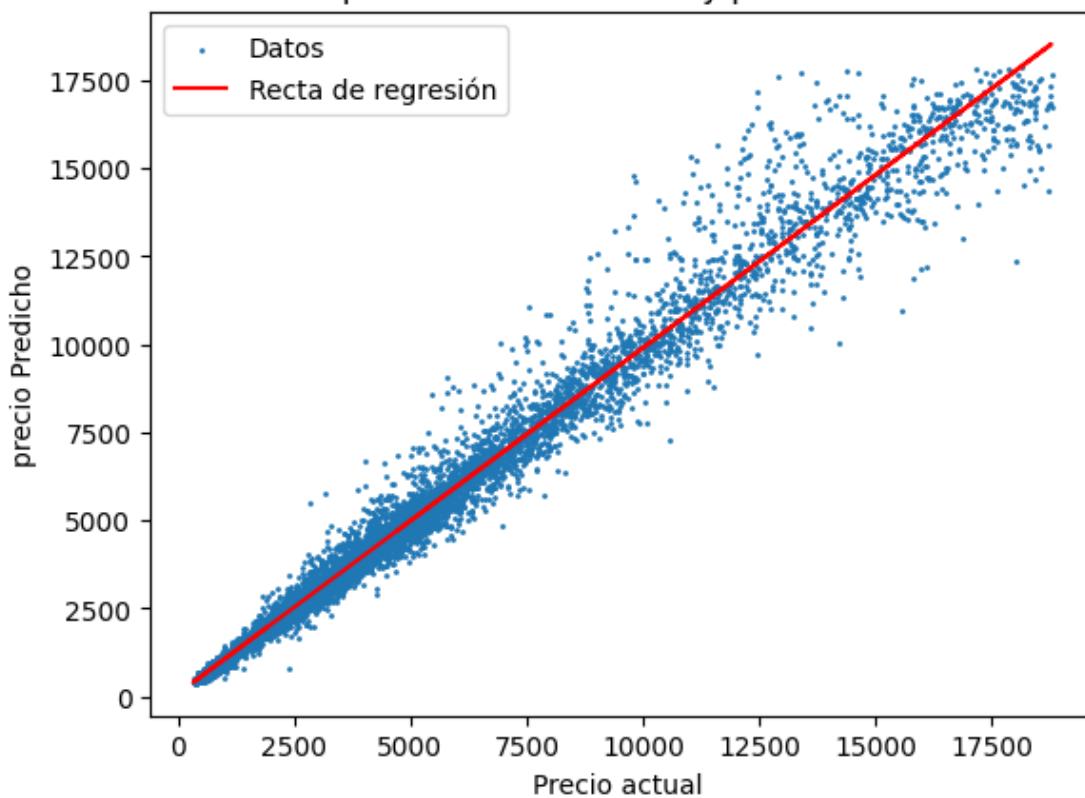
```

algoPrediccion.todasLasComparacionesDeActualPrediccion()

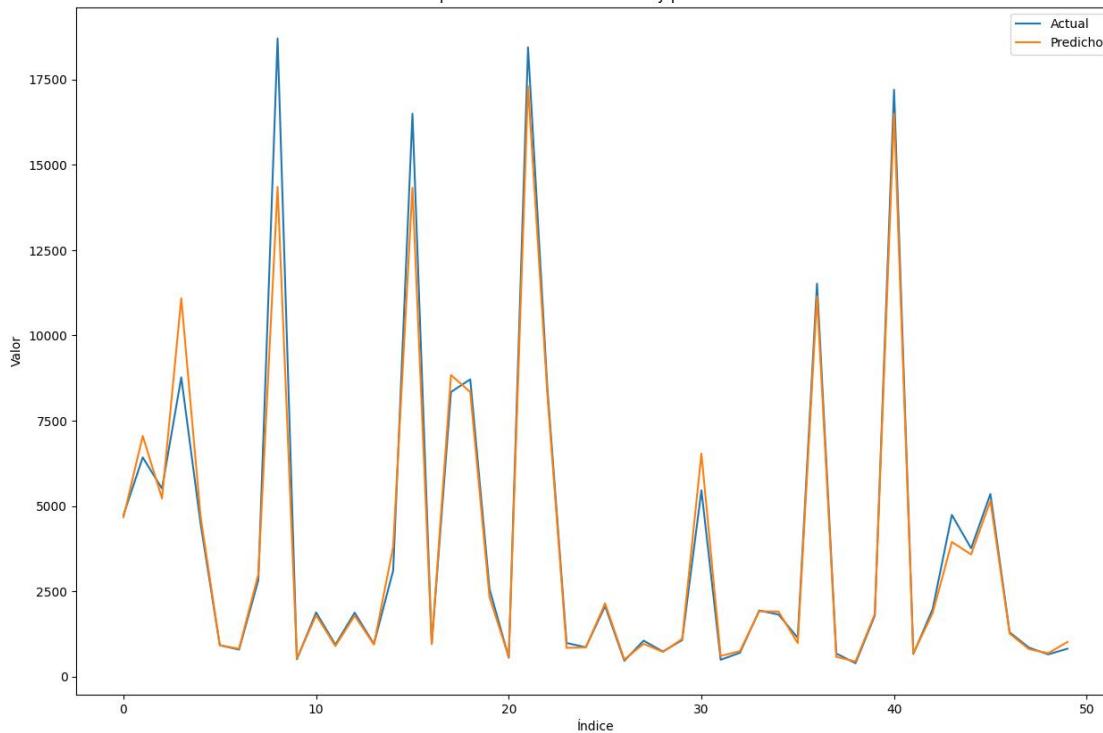
```



Comparación Precio actual y precio Predicho



Comparación entre Precio actual y precio Predicho



```
algoPrediccion.conclusiones()
```

	Media	Error %	Efectividad %
Mean Absolute Error	278.89	7.09	92.91
Root Mean Squared Error	550.70	14.00	86.00
Mean Squared Error	303267.36	NaN	NaN

Regresion con K Nearest Neighbors (KNN)

```
from algoritmos import RegresionKNN
algoPrediccion.algoritmo = RegresionKNN()
algoPrediccion.realizarEntrenamientoSinDivisionDeConjuntos()
algoPrediccion.resultadoDeEntrenamiento()
```

Veamos la clase

```
class DataKNN():
    K = None
    rmseValor = None
    y_pred = None

    def __init__(self, K,rmseValor,y_pred ):
        self = self
        self.K = K
        self.rmseValor = rmseValor
        self.y_pred=y_pred

class RegresionKNN(TipoDeRegresion):

    listaDataKNN = []

    def entrenar(self, RegresionModelo):
        for K in range(20):
            K = K+1
            model = KNeighborsRegressor(n_neighbors = K)
            model.fit(RegresionModelo.X_train, RegresionModelo.y_train) # fit
            y_pred=model.predict(RegresionModelo.X_test).flatten() # hacer predicciones
            en el conjunto de prueba
```

```

        rmseValor = sqrt(mean_squared_error(RegresionModelo.y_test,y_pred)) #
calcular rmse
        self.listaDataKNN.append(DataKNN(K,rmseValor,y_pred))
        print(f'Valor RMSE para k = {K} , es:{rmseValor}')

def dataKNNDefinitivo(self) -> DataKNN :
    dataKNNDefinitivo : DataKNN= min(self.listaDataKNN, key=lambda x: x.rmseValor)
    print("El K seleccionado por tener el valor RMSE minimo es :
",dataKNNDefinitivo.K,"con un RMSE De : ",dataKNNDefinitivo.rmseValor)
    return dataKNNDefinitivo

def prediccion(self, RegresionModelo):
    return self.dataKNNDefinitivo().y_pred

def nombreDeRegresion(self):
    return "Regresion con KNN"

```

Valor RMSE para k = 1 es: 1286.0419825591655
 Valor RMSE para k = 2 es: 1167.0826103902598
 Valor RMSE para k = 3 es: 1155.4326316748816
 Valor RMSE para k = 4 es: 1147.0874627788073
 Valor RMSE para k = 5 es: 1152.3116419213713
 Valor RMSE para k = 6 es: 1160.9909466845484
 Valor RMSE para k = 7 es: 1168.1302649464492
 Valor RMSE para k = 8 es: 1167.5557946209278
 Valor RMSE para k = 9 es: 1174.934946581827
 Valor RMSE para k = 10 es: 1180.236738093619
 Valor RMSE para k = 11 es: 1187.3693319551353
 Valor RMSE para k = 12 es: 1191.8026718211581
 Valor RMSE para k = 13 es: 1195.4980031228713
 Valor RMSE para k = 14 es: 1199.362188507188
 Valor RMSE para k = 15 es: 1207.727587318237
 Valor RMSE para k = 16 es: 1213.4885378848796
 Valor RMSE para k = 17 es: 1221.6283101442814
 Valor RMSE para k = 18 es: 1226.960637761948
 Valor RMSE para k = 19 es: 1231.2493441037193
 Valor RMSE para k = 20 es: 1236.0697164570095
 El K seleccionado por tener el valor RMSE minimo es : 4 con un RMSE De : 1147.0874627788073

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
0	4733	4645.00	88.00	1.86
1	6424	6785.50	361.50	5.63
2	5510	4805.25	704.75	12.79
3	8770	10792.50	2022.50	23.06
4	4493	5926.00	1433.00	31.89
...

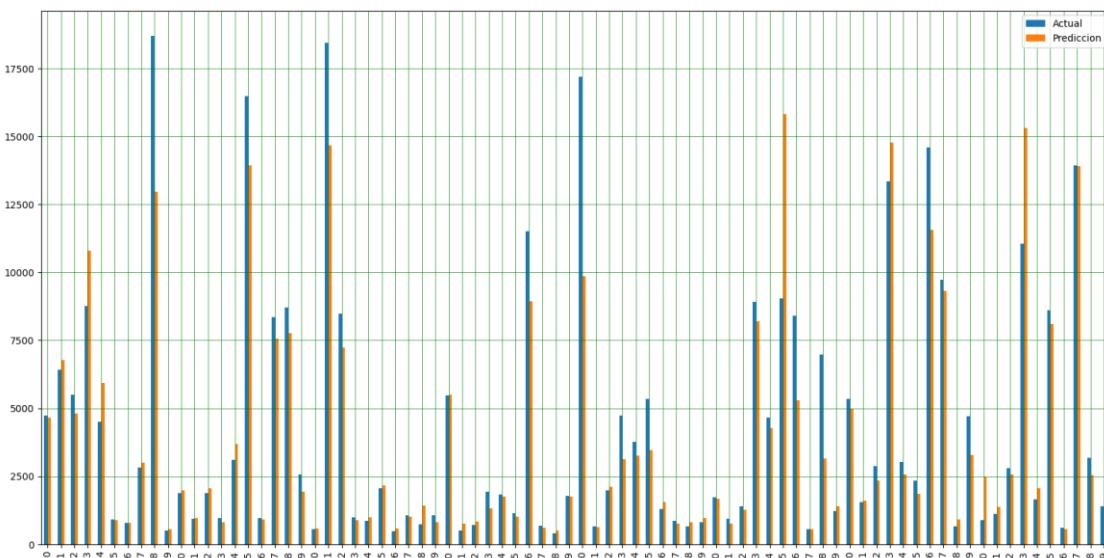
10783	1289	843.75	445.25	34.54
10784	3435	3619.25	184.25	5.36
10785	3847	4993.50	1146.50	29.80
10786	8168	8784.75	616.75	7.55
10787	1917	1001.50	915.50	47.76

[10788 rows x 4 columns]

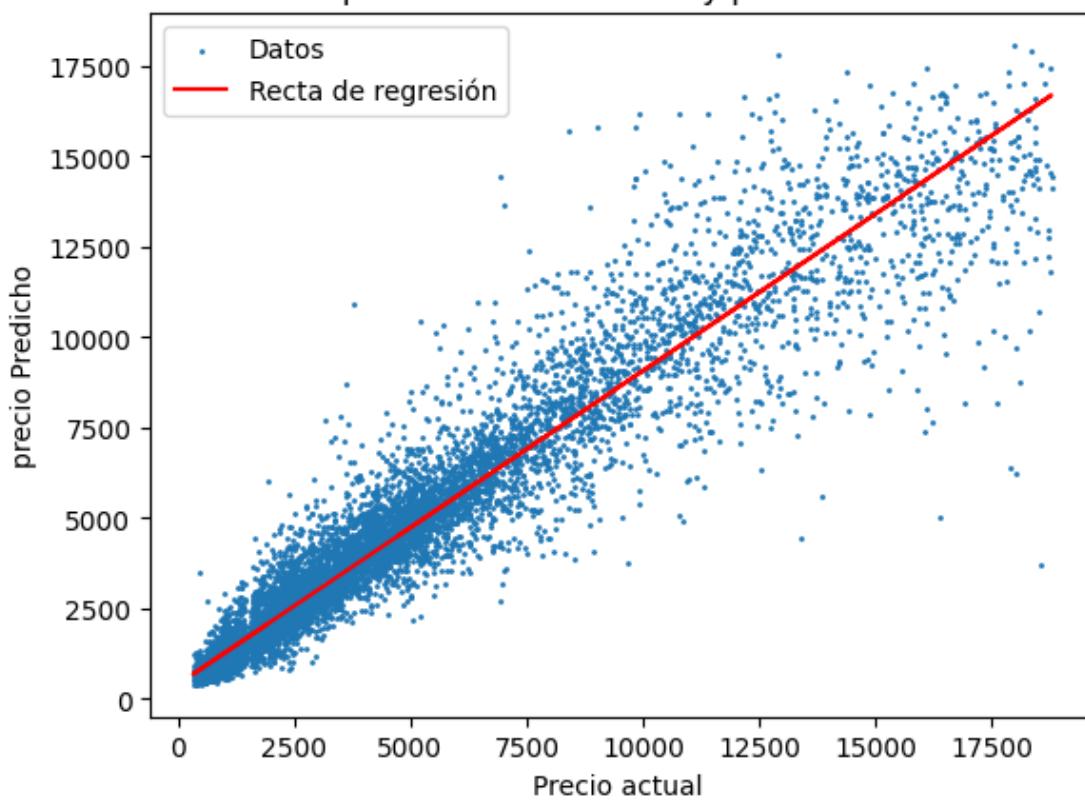
```
algoPrediccion.resultadoDeEntrenamiento().describe()
```

	Actual	Prediccion	Error Absoluto	Error porcentual absoluto %
count	10788.00	10788.00	10788.00	10788.00
mean	3929.21	3809.64	614.23	17.58
std	3981.60	3598.34	968.82	20.90
min	326.00	393.00	0.00	0.00
25%	947.50	960.50	92.00	5.30
50%	2398.00	2549.62	249.75	12.00
75%	5311.25	5221.50	717.56	22.18
max	18787.00	18083.00	14870.00	638.82

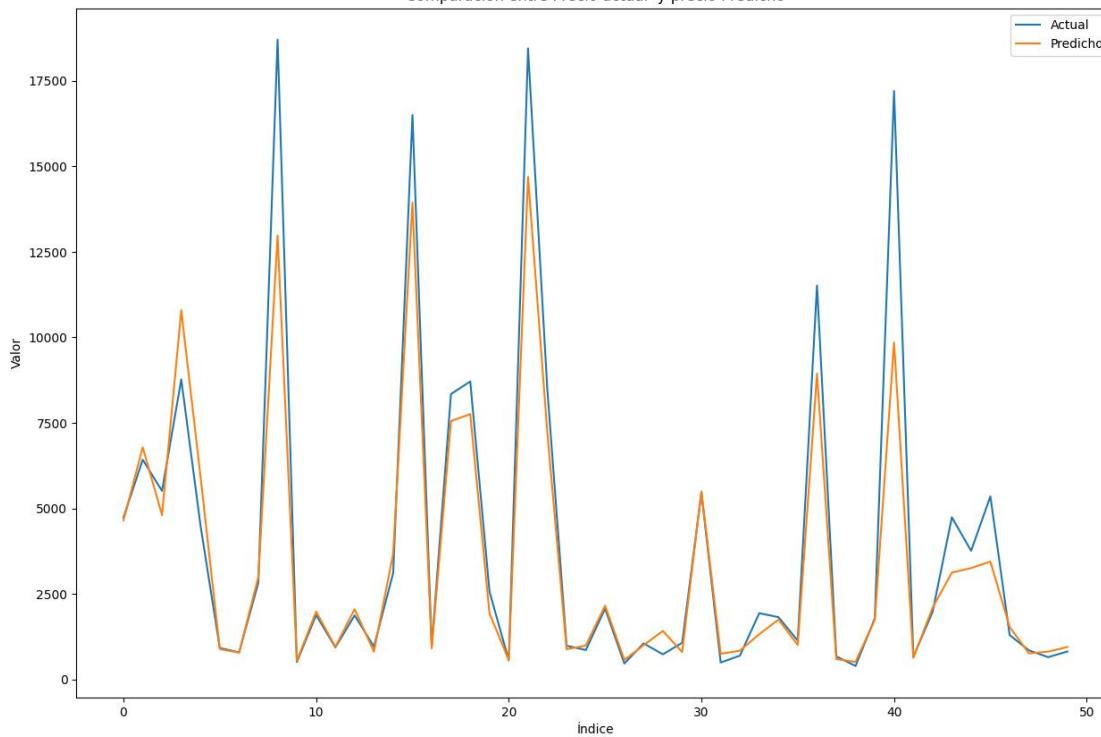
```
algoPrediccion.todasLasComparacionesDeActualPrediccion()
```



Comparación Precio actual y precio Predicho



Comparación entre Precio actual y precio Predicho



```
algoPrediccion.conclusiones()
```

	Media	Error %	Efectividad %
Mean Absolute Error	614.23	15.62	84.38
Root Mean Squared Error	1147.09	29.17	70.83
Mean Squared Error	1315809.65	NaN	NaN

Finalmente se ha mostrado los resultado de todos los algoritmos de regresión considerados en este tp. Pero antes de mostrar los gráficos comparativos de cada uno y ver cual es el algoritmo que tuvo mejores resultados veamos lo siguiente

```

def conclusiones(self):
    mean_absolute_error = metrics.mean_absolute_error(self.y_test, self.y_pred)
    mean_squared_error = metrics.mean_squared_error(self.y_test, self.y_pred)
    root_mean_squared_error = np.sqrt(mean_squared_error)
    promedioDePrecios = self.df['price'].mean()
    errorCuadraticoPorcentaje = (root_mean_squared_error * 100) / promedioDePrecios
    efectividadCuadraticaPrediccion = 100 - errorCuadraticoPorcentaje

    errorAbsolutoPorcentaje = (mean_absolute_error * 100) / promedioDePrecios
    efectividadAbsolutaPrediccion = 100 - errorAbsolutoPorcentaje

    # Crea el DataFrame con los resultados
    data = {
        'Media': [mean_absolute_error, root_mean_squared_error, mean_squared_error],
        'Error %': [errorAbsolutoPorcentaje, errorCuadraticoPorcentaje, np.nan],
        'Efectividad %': [efectividadAbsolutaPrediccion, efectividadCuadraticaPrediccion, np.nan]
    }

    nombreDeRegresion = self.algoritmo.nombreDeRegresion()

    self.mean_absolute_errorDic[nombreDeRegresion] = mean_absolute_error
    self.mean_squared_errorDic[nombreDeRegresion] = mean_squared_error
    self.root_mean_squared_errorDic[nombreDeRegresion] = root_mean_squared_error

    self.meanAbsoluteErrorPorcentajeDic[nombreDeRegresion]=errorAbsolutoPorcentaje
    self.meanSquaredErrorPorcentajeDic[nombreDeRegresion]=errorCuadraticoPorcentaje
    self.promedioAbsolutoEfectividadPorcentajeDic[nombreDeRegresion]=efectividadAbsolutaPrediccion
    self.promedioCuadraticoEfectividadPorcentajeDic[nombreDeRegresion]=efectividadCuadraticaPrediccion

    self.guardarArchivo()

    df = pd.DataFrame(data, index=['Mean Absolute Error', 'Root Mean Squared Error', 'Mean Squared Error'])
    return df

```

Todos los modelos de regresión terminaron sus pruebas ejecutando este método llamado conclusiones que como vimos es propio del objeto RegresionModelo . vemos que cada vez que se ejecuta este método todos los diccionarios que se encuentran adentro del rectángulo reciben una key y un value. La key es determinada por el algoritmo de regresión (strategy) que le corresponda. Por ejemplo el caso mas sencillo es el de regresión lineal

```

class RegresionPolinomica(TipoDeRegresion) :

    def nombreDeRegresion(self):
        return "Regresion Polinomica"

y el caso mas complicado es

class RegresionConArboles(TipoDeRegresion):

    regression =None
    def nombreDeRegresion(self):
        tipoDeArbolDiccionario = {
            "RandomForestRegressor": "Regresion con Random forest",
            "DecisionTreeRegressor": "Regresion con Arboles de desicion"}
        tipo = self.regression.__class__.__name__
        return tipoDeArbolDiccionario[tipo]

```

retomando a la imagen

	Media	Error %	Efectividad %
Mean Absolute Error	348.75	8.87	91.13
Root Mean Squared Error	623.04	15.84	84.16
Mean Squared Error	388176.90	NaN	NaN

```

def conclusiones(self) :
    mean_absolute_error = metrics.mean_absolute_error(self.y_test, self.y_pred)
    mean_squared_error = metrics.mean_squared_error(self.y_test, self.y_pred)
    root_mean_squared_error = np.sqrt(mean_squared_error)
    promedioDePrecios = self.df['price'].mean()
    errorCuadraticoPorcentaje = (root_mean_squared_error * 100) / promedioDePrecios
    efectividadCuadraticaPrediccion = 100 - errorCuadraticoPorcentaje

    errorAbsolutoPorcentaje = (mean_absolute_error * 100) / promedioDePrecios
    efectividadAbsolutaPrediccion = 100 - errorAbsolutoPorcentaje

    # Crea el DataFrame con los resultados
    data = {
        'Media': [mean_absolute_error,root_mean_squared_error,mean_squared_error] ,
        'Error %': [errorAbsolutoPorcentaje,errorCuadraticoPorcentaje,np.nan] ,
        'Efectividad %' : [efectividadAbsolutaPrediccion,efectividadCuadraticaPrediccion,np.nan]
    }

    nombreDeRegresion = self.algoritmo.nombreDeRegresion()

    self.mean_absolute_errorDic[nombreDeRegresion] = mean_absolute_error
    self.mean_squared_errorDic[nombreDeRegresion] = mean_squared_error
    self.root_mean_squared_errorDic[nombreDeRegresion] = root_mean_squared_error

    self.meanAbsoluteErrorPorcentajeDic[nombreDeRegresion]=errorAbsolutoPorcentaje
    self.meanSquaredErrorPorcentajeDic[nombreDeRegresion]=errorCuadraticoPorcentaje
    self.promedioAbsolutoEfectividadPorcentajeDic[nombreDeRegresion]=efectividadAbsolutaPrediccion
    self.promedioCuadraticoEfectividadPorcentajeDic[nombreDeRegresion]=efectividadCuadraticaPrediccion

    self.guardarArchivo()

    df = pd.DataFrame(data, index=['Mean Absolute Error', 'Root Mean Squared Error', 'Mean Squared Error'])
    return df

```

El objetivo es que cada vez que el objeto RegresionModelo ejecute el método conclusión() se guarde los datos mas importante que indican que tan efectivo fue el rendimiento del algoritmo seleccionado. Por supuesto la key representa el nombre del algoritmos en cuestión y las values los resultados que se obtuvieron. Osea en simples palabras se podría decir que se esta guardando los valores de la tabla

conclusión que retorna justamente este metodo que se muestra al final de la prueba de cada algoritmo. Esta es una forma automatizada de hacerlo .Por supuesto se podría haber anotado los resultados en algún lado y luego armar el dataframe final que permite comparar que algoritmo fue mejor.

RESULTADOS FINALES

```
# Cargar la variable desde el archivo
with open('variable.pkl', 'rb') as f:
    algoPrediccion = pickle.load(f)
```

```
algoPrediccion.mean_absolute_errorDic
```

```
{'Regresion Lineal': 737.779013718947,
'Regresion Polinomica': 412.10183615113436,
'Regresion SVR con el Kernel : rbf': 348.7512835754309,
'Regresion SVR con el Kernel : poly': 407.7371556739417,
'Regresion SVR con el Kernel : linear': 658.6852907546541,
'Regresion con Arboles de desicion': 363.36545235446795,
'Regresion con KNN': 614.2302326659251,
'Regresion con Random forest': 278.8850031642791}
```

```
import pandas as pd
import matplotlib.pyplot as plt
import operator

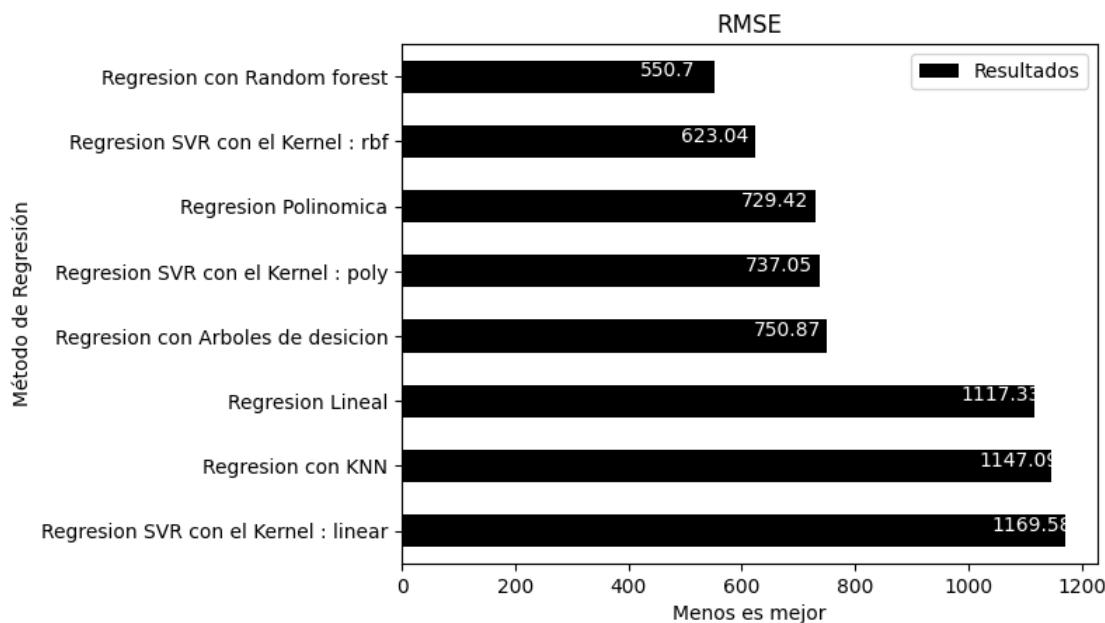
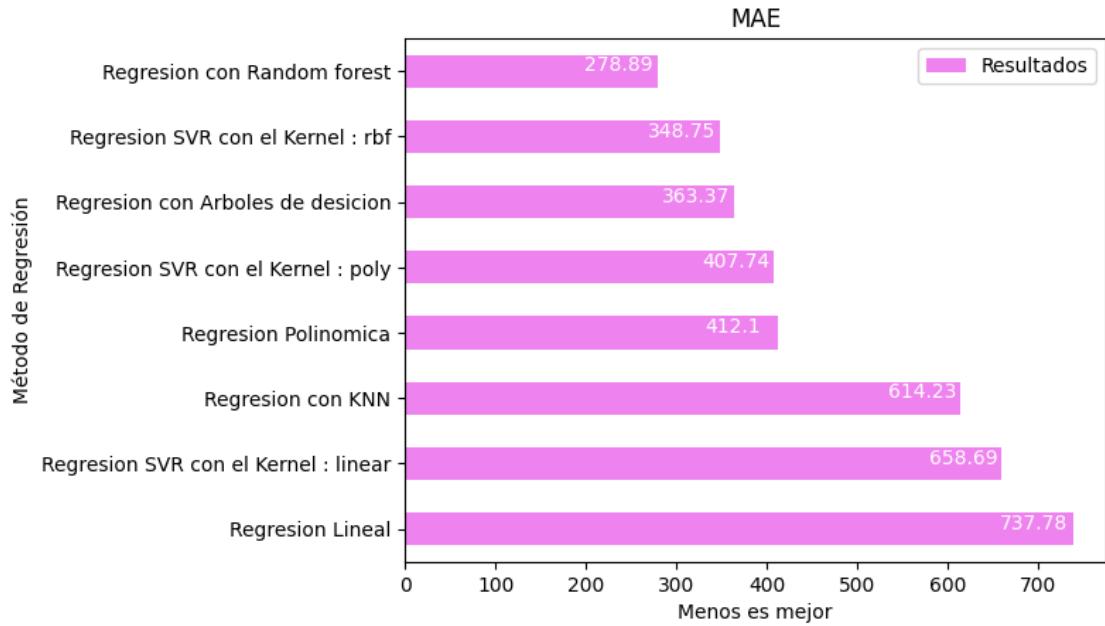
dfMAE = pd.DataFrame.from_dict(algoPrediccion.mean_absolute_errorDic, orient='index', columns=['Resultados'])
dfRMSE = pd.DataFrame.from_dict(algoPrediccion.root_mean_squared_errorDic, orient='index', columns=['Resultados'])

dfMAEPorcentaje = pd.DataFrame.from_dict(algoPrediccion.meanAbsoluteErrorPorcentajeDic, orient='index', columns=['Resultados'])
dfRMSEPorcentaje = pd.DataFrame.from_dict(algoPrediccion.meanSquaredErrorPorcentajeDic, orient='index', columns=['Resultados'])

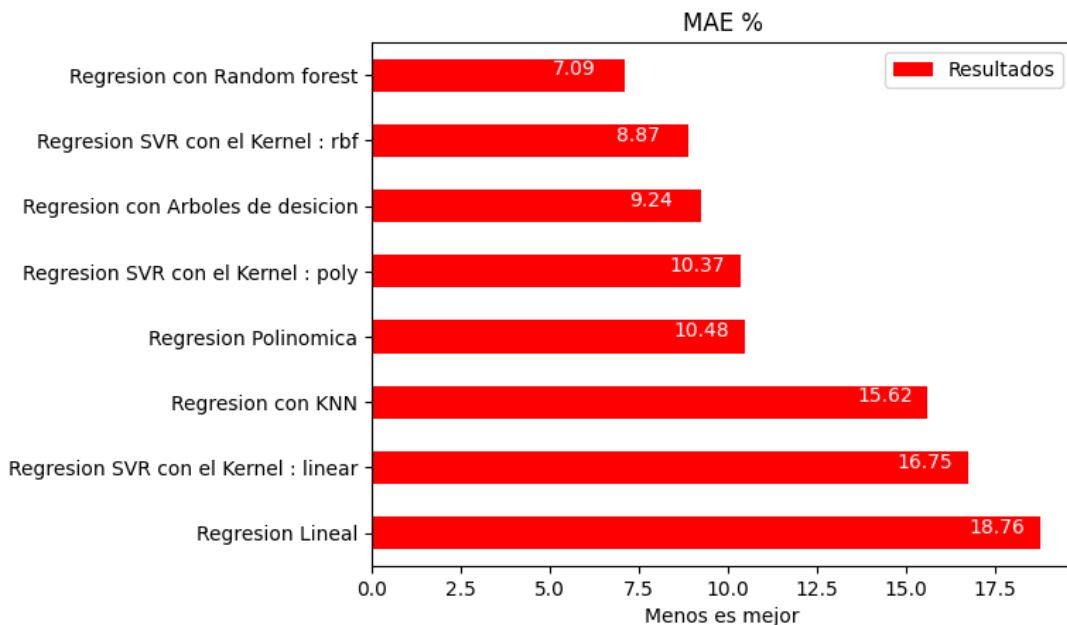
promedioAbsolutoEfectividadPorcentajeDic = pd.DataFrame.from_dict(algoPrediccion.promedioAbsolutoEfectividadPorcentajeDic, orient='index', columns=['Resultados'])
promedioCuadraticoEfectividadPorcentajeDic = pd.DataFrame.from_dict(algoPrediccion.promedioCuadraticoEfectividadPorcentajeDic, orient='index', columns=['Resultados'])

def comparacion(df,ascendente,descripcionAdicional,x,elColor,titulo):
    df = df.sort_values('Resultados', ascending=ascendente)
    df['Resultados']= df['Resultados'].round(2)
    pd.set_option('display.float_format', lambda x: '{:.2f}'.format(x)) #PARA EVITAR LA NOTACION CIENTIFICA
    # Crear el gráfico de barras horizontales
    ax = df.plot(kind='barh',color=elColor)
    # Añadir los valores de las barras al lado de cada barra
    for i, v in enumerate(df['Resultados']):
        ax.text(v - x , i, str(v), color='white')
    # Añadir título y etiquetas de los ejes
    plt.title(titulo)
    plt.xlabel(descripcionAdicional)
    plt.ylabel('Método de Regresión')
    # Mostrar el gráfico
    plt.show()

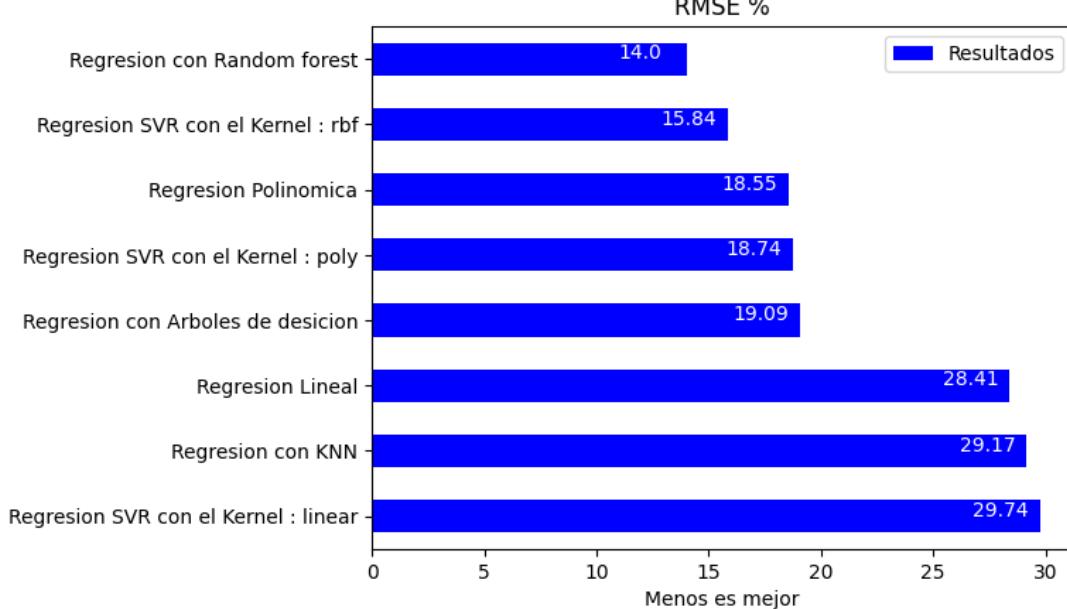
comparacion(dfMAE,False,"Menos es mejor",80,"violet","MAE")
comparacion(dfRMSE,False,"Menos es mejor",130,"black","RMSE")
comparacion(dfMAEPorcentaje,False,"Menos es mejor",2,"red","MAE %")
comparacion(dfRMSEPorcentaje,False,"Menos es mejor",3,"blue","RMSE %")
comparacion(promedioAbsolutoEfectividadPorcentajeDic,True,"MAS es mejor",30,"green","Porcentaje de efectividad ABSOLUTO(100-MAE)")
comparacion(promedioCuadraticoEfectividadPorcentajeDic,True,"MAS es mejor",30,"orange","Porcentaje de efectividad CUADRATICO(100-RMSE)")
```



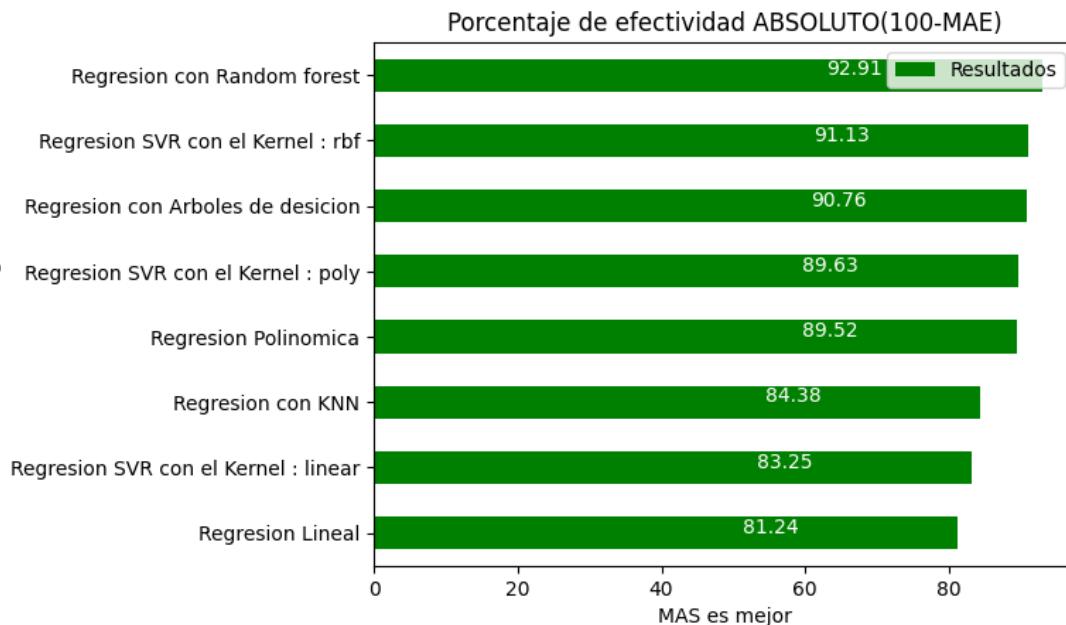
Método de Regresión



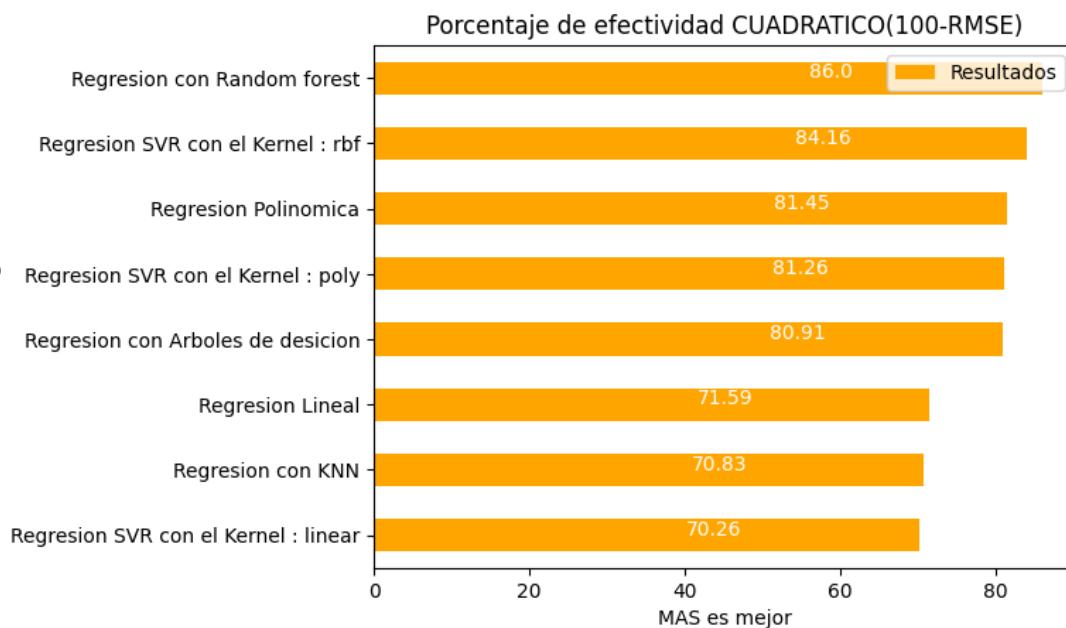
Método de Regresión



Método de Regresión



Método de Regresión



Definitivamente, el algoritmo que mejor rendimiento mostró en nuestro estudio fue el Random Forest. Si tuviéramos que predecir el precio de un nuevo diamante, sin duda utilizaríamos el Random Forest como nuestra primera opción. Este algoritmo demostró ser no solo rápido, sino también capaz de darnos las mejores predicciones en comparación con otros algoritmos evaluados. Definitivamente no utilizaríamos los modelos de Regresión Lineal Y KNN que si bien no dieron resultados tan malos aun así serían las últimas opciones a considerar.

