Notebook_Semana5_RegresionLineal

June 29, 2023

1 Modelos lineales de regresión

[5]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1.1 Introducción

Uno de los pasos más avanzados del proceso CRISP-DM es el **modelado**. ¿Qué significa exactamente esto?

Los modelos son expresiones matemáticas más o menos simples (depende de la **familia de modelos** que se elija utilizar), que describen algún aspecto de los datos. Hoy vamos a charlar de la familia más conocida y sencilla de modelos, pero que es a la vez muy poderosa y versatil: los **modelos lineales de regresión**.

Los modelos lineales se encuentran entre los modelos más simples que podemos imaginar, pero siguen siendo extremadamente comunes y útiles. Tienen algunas propiedades analíticas simples y son extremadamente fáciles de entrenar e interpretar. Además, en la versión múltiple, son poderosos y versátiles.

Un **modelo de regresión** es aquel en el que se busca describir el comportamiento de una variable (o un conjunto de variables) contínua a partir de otras variables del dataset. La variable que se busca describir (o *predecir*) se suele llamar variable objetivo (*target*). Las variables a partir de las cuales se busca hacer esto se llaman *covariables*, o *variables predictora*, o *variables independientes*, *regresores*, *features* (*características*), etc. Como muchas cosas importantes, tiene muchos nombres.

Ejemplos de problemas de regresión:

1. Predecir la mediana del precio de las casas en un distrito de California a partir de la mediana del ingreso en ese distrito y otras características del dataset.
2. Predecir la altura de los árboles en CABA a partir de su ancho y de la especie a la que pertenecen
3. Predecir el precio de un diamante a partir de sus quilates y el tipo de corte.
4. ...

1.1.1 Celdas preparatorias

Como de costumbre, tenemos que arrancar importando distintos paquetes.

[6]:
```
# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Pandas
import pandas as pd

# Common imports
import numpy as np
import os
from scipy import stats as st

# Para hacer gráficos lindos
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
print("OK")
```

```
OK
```

1.2 Obtención y preparación de los datos.

Vamos a suponer que tenemos interés en entender cómo depende el valor de venta de casas en la Ciudad de Buenos Aires de su superficie.

Vamos a trabajar con un conjunto de datos de un portal inmobiliario Properati, que hasta hace un tiempo era público. Esta versión del dataset viene de Kaggle, y lo modificamos un poco para hacerlo más simple.

El dataset es muy grande, así que vamos a hacer algo de limpieza y filtrado antes de pasar al modelado.

[8]:
```
DATA_DIR = '/content/drive/MyDrive/Colab Notebooks/Modulo 2/Notebooks/datasets'
df = pd.read_csv(os.path.join(DATA_DIR, 'properati_arg_SPA.csv'))
```

[9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 25 columns):
 #   Column          Non-Null Count    Dtype
```

```
---  ------          --------------    -----
 0   id              1000000 non-null  object
 1   ad_type         1000000 non-null  object
 2   start_date      1000000 non-null  object
 3   end_date        1000000 non-null  object
 4   created_on      1000000 non-null  object
 5   lat             848180 non-null   float64
 6   lon             847236 non-null   float64
 7   l1              1000000 non-null  object
 8   l2              1000000 non-null  object
 9   l3              946143 non-null   object
 10  l4              233218 non-null   object
 11  l5              4118 non-null     object
 12  l6              0 non-null        float64
 13  habitaciones    529819 non-null   float64
 14  dormitorios     431304 non-null   float64
 15  baños           767293 non-null   float64
 16  sup_total       430004 non-null   float64
 17  sup_cubierta    416363 non-null   float64
 18  precio          957089 non-null   float64
 19  moneda          953765 non-null   object
 20  price_period    341682 non-null   object
 21  título          1000000 non-null  object
 22  descr           999987 non-null   object
 23  tipo_propiedad  1000000 non-null  object
 24  tipo_operac     1000000 non-null  object
dtypes: float64(9), object(16)
memory usage: 190.7+ MB
```

En primer lugar vamos a quedarnos solo con los registros en los que las variables que nos interesan tienen valores. Para eso, usamos el método `dropna`.

[10]:
```
df = df.dropna(subset=['precio', 'sup_total'])
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 405598 entries, 81 to 999997
Data columns (total 25 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   id              405598 non-null  object
 1   ad_type         405598 non-null  object
 2   start_date      405598 non-null  object
 3   end_date        405598 non-null  object
 4   created_on      405598 non-null  object
 5   lat             355869 non-null  float64
 6   lon             355869 non-null  float64
 7   l1              405598 non-null  object
```

```
 8   l2              405598 non-null  object
 9   l3              393456 non-null  object
 10  l4              125140 non-null  object
 11  l5              3197 non-null    object
 12  l6              0 non-null       float64
 13  habitaciones    278464 non-null  float64
 14  dormitorios     218024 non-null  float64
 15  baños           336937 non-null  float64
 16  sup_total       405598 non-null  float64
 17  sup_cubierta    348392 non-null  float64
 18  precio          405598 non-null  float64
 19  moneda          402927 non-null  object
 20  price_period    169263 non-null  object
 21  título          405598 non-null  object
 22  descr           405589 non-null  object
 23  tipo_propiedad  405598 non-null  object
 24  tipo_operac     405598 non-null  object
dtypes: float64(9), object(16)
memory usage: 80.5+ MB
```

Quitemos algunas columnas con muy pocos datos, como `l5` y `l6`

[11]:
```
# Con inplace=True, la variable df se sobreescribe
df.drop(['l5', 'l6'], axis=1, inplace=True)
```

Ahora algunos filtros:

1. nos quedamos solo con las propiedades en la Ciudad de Buenos Aires.
2. como la relación entre el precio y el tamaño puede cambiar significativamente de barrio en barrio, solo nos quedaremos con un barrio a elección (¡completen cada uno el valor del barrio!)
3. para tener un objeto más uniforme de estudio, solo utilizaremos las **ventas** de **casas**.

Como queremos que se cumplan todas las condiciones, las concatenamos con `&`.

[12]:
```
df.columns
```

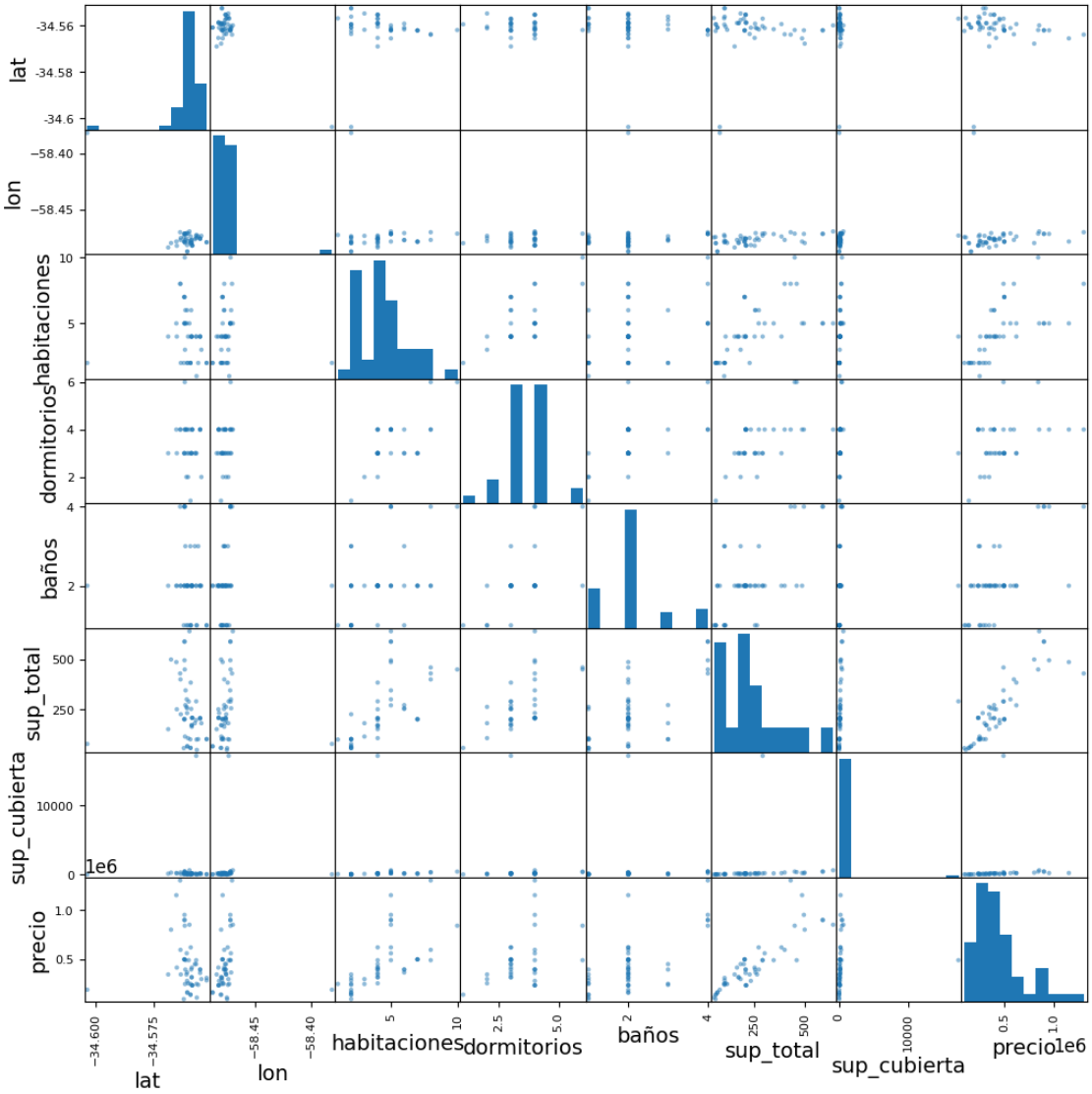
[12]:
```
Index(['id', 'ad_type', 'start_date', 'end_date', 'created_on', 'lat', 'lon',
       'l1', 'l2', 'l3', 'l4', 'habitaciones', 'dormitorios', 'baños',
       'sup_total', 'sup_cubierta', 'precio', 'moneda', 'price_period',
       'título', 'descr', 'tipo_propiedad', 'tipo_operac'],
      dtype='object')
```

[13]:
```
barrio = 'Coghlan'

df_filtro = df[(df.l2=='Capital Federal') &
               (df.l3 == barrio) &
               (df.tipo_propiedad == 'Casa') &
               (df.tipo_operac == 'Venta')]
```

```
df_filtro.info()
```

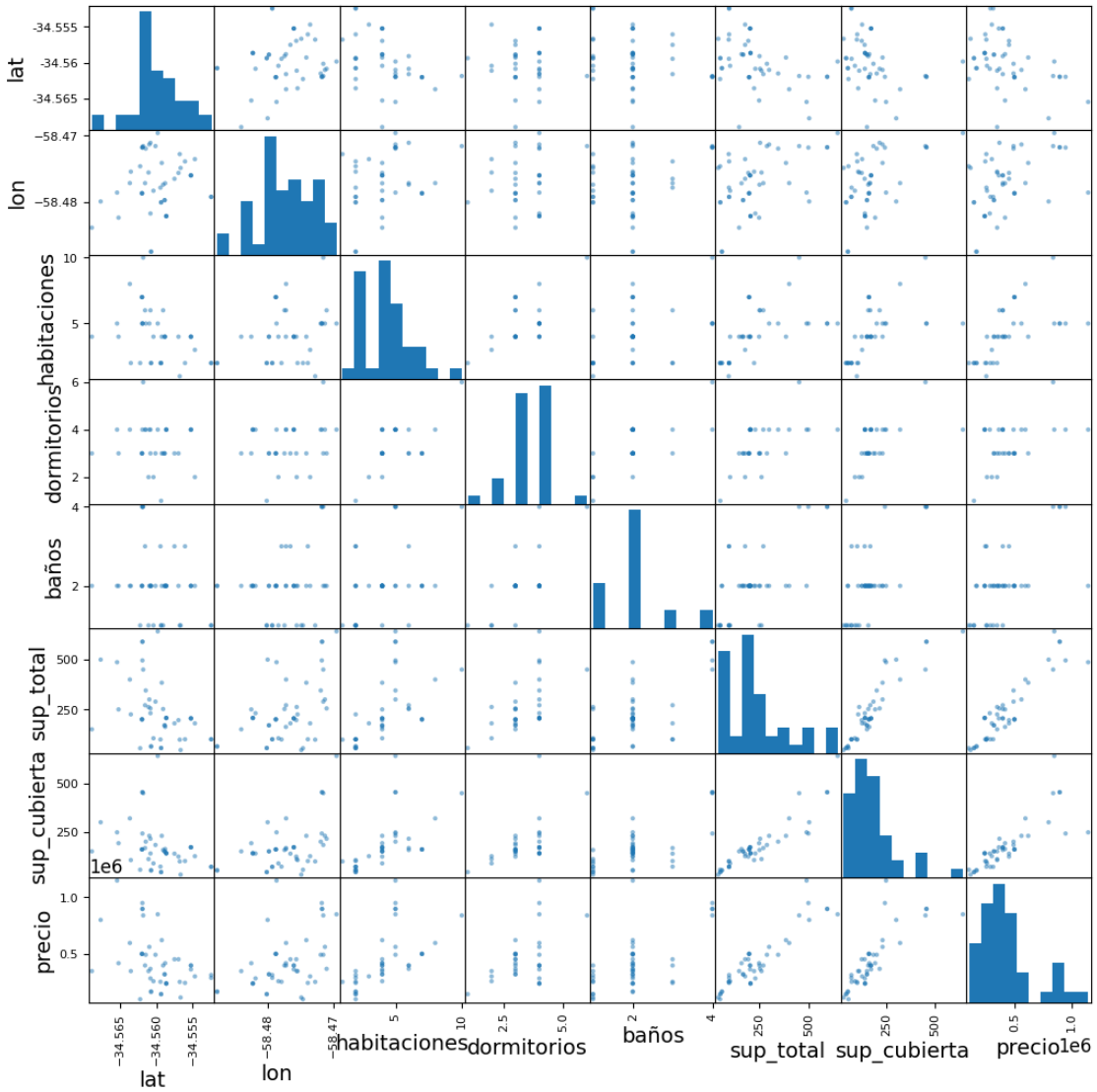
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 53 entries, 13470 to 984253
Data columns (total 23 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   id              53 non-null     object 
 1   ad_type         53 non-null     object 
 2   start_date      53 non-null     object 
 3   end_date        53 non-null     object 
 4   created_on      53 non-null     object 
 5   lat             51 non-null     float64
 6   lon             51 non-null     float64
 7   l1              53 non-null     object 
 8   l2              53 non-null     object 
 9   l3              53 non-null     object 
 10  l4              0 non-null      object 
 11  habitaciones    44 non-null     float64
 12  dormitorios     36 non-null     float64
 13  baños           49 non-null     float64
 14  sup_total       53 non-null     float64
 15  sup_cubierta    51 non-null     float64
 16  precio          53 non-null     float64
 17  moneda          53 non-null     object 
 18  price_period    9 non-null      object 
 19  título          53 non-null     object 
 20  descr           53 non-null     object 
 21  tipo_propiedad  53 non-null     object 
 22  tipo_operac     53 non-null     object 
dtypes: float64(8), object(15)
memory usage: 9.9+ KB
```

[14]:
```
_ = pd.plotting.scatter_matrix(df_filtro, figsize=(12, 12))
```



Vemos un par de outliers adicionales, en (lat, lon) y en sup_cubierta

[15]:
```
df_filtro = df_filtro[(df_filtro.lon < -58.45) &
                      (df_filtro.sup_cubierta < 10000)]

_ = pd.plotting.scatter_matrix(df_filtro, figsize=(12, 12))
```



Perfecto! Ahora tenemos un dataset mucho más manejable, ideal para mostrar el funcionamiento de los modelos lineales.

PEro antes de arrancar, veamos que los precios de la casas esten expresados de manera consistente.

[16]:
```
# Pedimos los valores únicos de la variable "moneda"
print(pd.unique(df_filtro.moneda))
```

```
['USD' 'ARS']
```

Ah, vemos que hay algunas casas con precios en Pesos y otras en Dólares. Veamos cuántas de cada.

[17]:
```
pd.value_counts(df_filtro.moneda)
```

[17]:
```
USD    46
ARS     1
Name: moneda, dtype: int64
```

[18]:
```
# Veamos el registro de esta casa
df_filtro[df_filtro.moneda=='ARS']
```

[18]:
```
                              id    ad_type  start_date    end_date  \
639311  VnSkW6eEhTyZJprefI+nNg==  Propiedad  2020-07-28  2020-12-03

        created_on       lat       lon         l1               l2       l3  \
639311  2020-07-28 -34.56101 -58.47132  Argentina  Capital Federal  Coghlan

        … baños  sup_total  sup_cubierta    precio  moneda  price_period  \
639311  …   2.0      300.0         230.0  560000.0     ARS           NaN

                                          título  \
639311  Casa Moderna 4 Dorm Patio Garage Taller Dep

                                                 descr tipo_propiedad  \
639311  Corredor Responsable: Juan Carlos Treco - CUCI…           Casa

       tipo_operac
639311       Venta

[1 rows x 23 columns]
```

Pareciera que se trata de un error de ingreso de los datos, y que la moneda debería ser `USD`. Podemos hacer el cambio a mano, solo para ser prolijos.

[19]:
```
# Cambiamos a mano el valor de la variable moneda para este registro
# Atención, hay que usar .loc
df_filtro.loc[df_filtro.moneda=='ARS', 'moneda'] = 'USD'

# Verificamos que ahora solo hay valores en dólares
df_filtro.value_counts('moneda')
```

[19]:
```
moneda
USD    47
dtype: int64
```

Ahora calculemos el coeficiente de Correlación de Pearson entre la variable target y el resto de las variables numéricas

[20]:
```
df_filtro.corr(numeric_only=True).precio.sort_values()
```

[20]:
```
lat          -0.415589
lon           0.457666
```

```
dormitorios     0.474901
habitaciones    0.604886
baños           0.636713
sup_cubierta    0.817229
sup_total       0.917875
precio          1.000000
Name: precio, dtype: float64
```

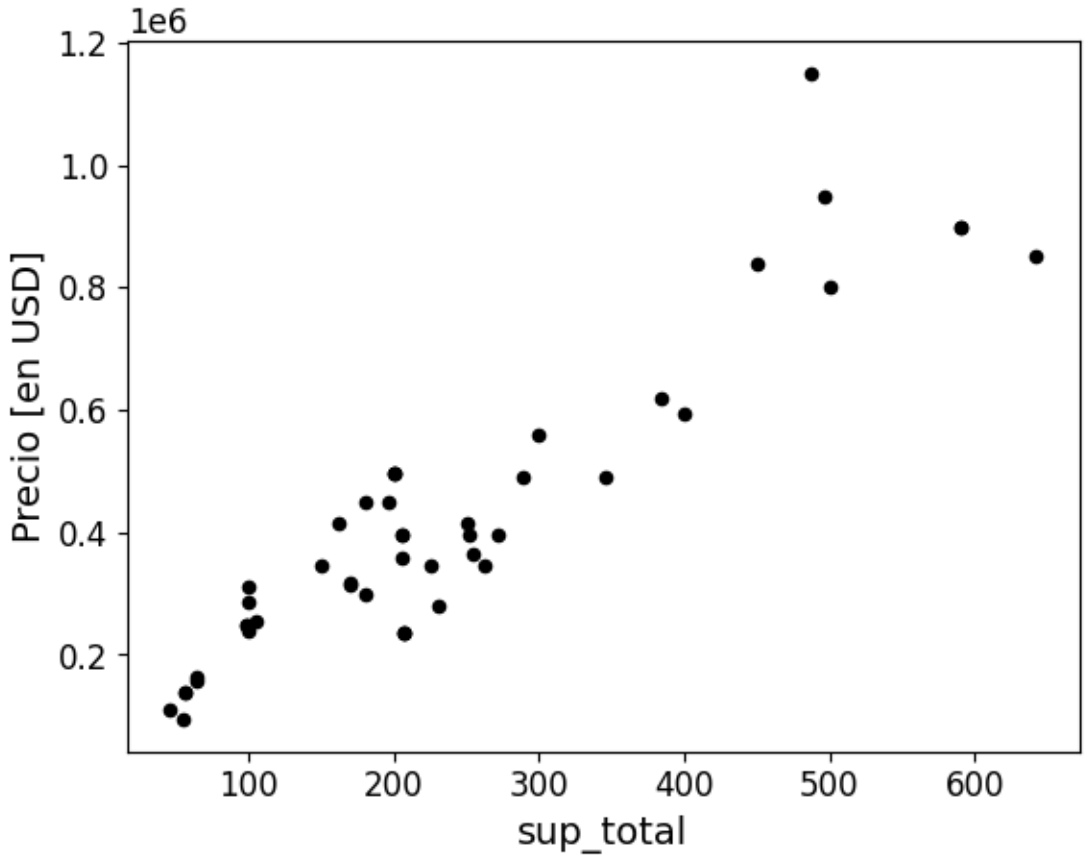
Vemos que existe una importante correlación entre el precio de venta y la superficie total (como era esperable!).

1.2.1 Visualización

Hagamos una mínima visualización de lo que tenemos.

[21]:
```
df_filtro.plot(kind='scatter', x='sup_total', y='precio', c='black',
               ylabel='Precio [en USD]')
```

[21]:
```
<Axes: xlabel='sup_total', ylabel='Precio [en USD]'>
```



Ahora sí estamos listos para arrancar.

1.3 El rol de los modelos y las familias de modelos

El modelado de datos tiene varios objetivos:

1. **Cuantificar** una relación (como la que vimos arriba). Es decir, ponerle números a eso: "el precio de las casas es de X USD por metro cuadrado")
2. **Explorar** los datos. Muchas veces, necesitamos quitar los patrones más obvios para poder detectar cosas más sutiles. En este caso, el patrón obvio es la dependencia con la superficie. ¿Habrá algún efecto secundario con, por ejemplo, el estado de la casa, o la cantidad de baños?
3. **Resumir** la información para transmitirla mejor. Un excelente complemento al gráfico de arriba es dar los números de los parámetros (ver abajo), que resultan una descripción compacta de los datos.
4. **Predecir** el valor de la variable *target* para una propiedad que no hemos observado.

1.4 Regresión lineal simple

Como se dijo arriba, para modelar los datos de arriba, podemos elegir una familia de modelos. Acá vmaos a elegir a los modelos lineales (y por ahora simples; es decir con una única variable predictora).

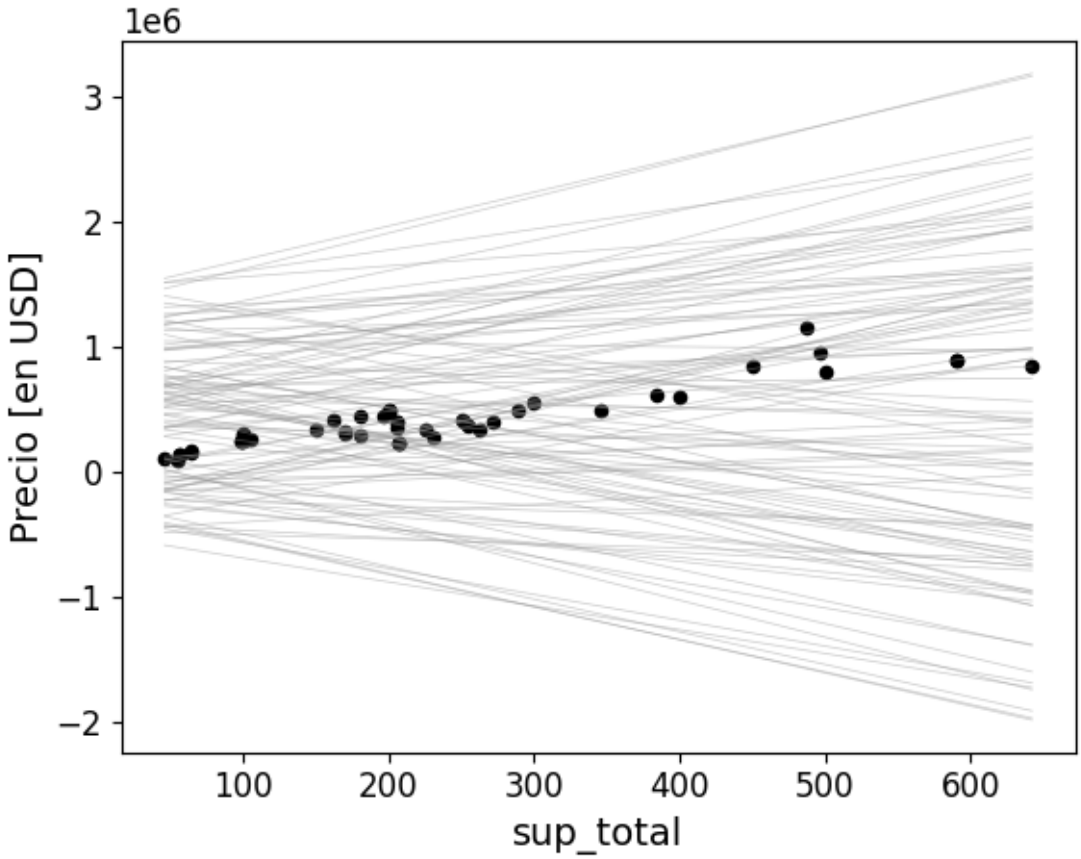
El modelo de regresión lineal más sencillo relaciona una variable *target* (en este caso el precio de las casas) con la covariable (en este caso, la superficie total), *x_1*, utilizando esta fórmula:

$$\text{precio [USD]} = \omega_0 + \omega_1 \cdot \text{sup. total} \ ,$$

donde $\omega = (\omega_0, \omega_1)$ es el **vector de parámetros del modelo**.

Esta fórmula define una **familia de modelos**. Todos los modelos de esta familia se ven como rectas en el gráfico de arriba, pero dependiendo del valor del vector de parámetros, pueden verse muy diferentes.

[22]:
```
df_filtro.plot(kind='scatter', x='sup_total', y='precio', c='black',
               ylabel='Precio [en USD]')

# Defino una serie de valores de los parámetros al azar
np.random.seed(20230605)
w0 = st.uniform(loc=-5e5, scale=2e6).rvs(100)
w1 = st.uniform(loc=-3000, scale=6000).rvs(100)

# Creo un arreglo con dos valores (suficiente para una recta, entre el máximo y␣
 ↪el mínimo de los valors de sup_total)
x = np.linspace(df_filtro.sup_total.min(), df_filtro.sup_total.max(), 2)

# Para cada par de parámetros w0 y w1, grafico una recta
for ww0, ww1 in zip(w0, w1):
    plt.plot(x, ww0 + ww1 * x, '-', color='0.6', lw=0.5, alpha=0.5)
```



De todas estas rectas, muy pocas tienen una similitud con los datos. Sin embargo, todas provienen de la misma familia de funciones.

Para encontrar una recta que describa correctamente los datos, entonces tenemos que encontrar el valor adecuado de los parámetros. La buena noticia es que existe una manera analítica (es decir, exacta) de encontrar el valor de los parámetros que hacen que la curva se ajuste lo mejor posible a los datos. Pero antes de llegar a eso….

Pensemos

- ¿Qué características nos gustaría que tuviera la curva que describa a los datos?

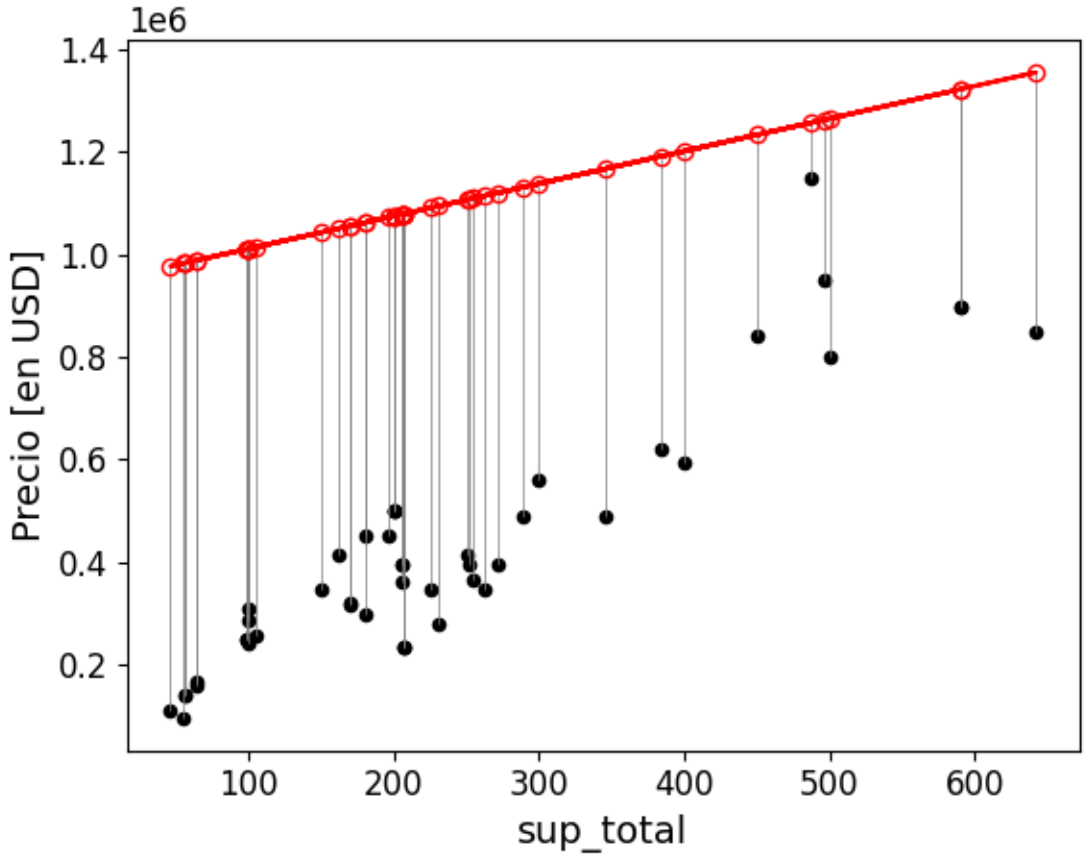
1.5 Los residuos

Para una recta cualquiera, podemos graficar la distancia entre los puntos y la recta como segmentos. Como vamos a usar mucho el gráfico de los datos, definimos la función `plot_data`.

[23]:
```
def plot_data(data, xvar='sup_total', yvar='precio', **kwargs):
    color = kwargs.pop('c', 'black')
    mietiqueta = kwargs.pop('ylabel', 'Precio [en USD]')
    data.plot(kind='scatter', x=xvar, y=yvar, c=color,
                   ylabel=mietiqueta)
```

```
plot_data(df_filtro)

# Genero un número al azar entre 0 y 100
irandom = np.random.randint(100)

# Grafico la curva correspondiente a ese número al azar
plt.plot(x, w0[irandom] + w1[irandom] * x, '-', color='0.6', lw=0.5)

# Agrego los residuos
pred = w0[irandom] + w1[irandom] * df_filtro.sup_total

# Grafico líneas verticales
plt.vlines(x=df_filtro.sup_total.values,
           ymin=df_filtro.precio.values,
           ymax=pred.values, color='0.5', lw=0.5)

# Agrego los puntos sobre la recta
plt.plot(df_filtro.sup_total, pred, marker='o', color='red', mfc='None', ms=6)
```

[23]: [<matplotlib.lines.Line2D at 0x7f62d19c9d50>]



A estas distancias, se las llama **residuos** de la recta. Pueden ser positivos, si los puntos negros están por encima de la recta, o negativos, si están por debajo.

Matemáticamente, para el punto i, podemos escribir que el residuo r_i es:

$r_i = \text{precio}_i - (\omega_0 + \omega_1 \cdot \text{sup. total}_i)$.

Con esta definición, podemos aventurar:

> El mejor conjunto de parámetros será el que haga que los residuos sean lo más chicos posibles.

Pero está claro que como los residuos pueden ser negativos, podemos hacer que los residuos se achiquen todo lo que querramos mandando la recta para arriba, pero eso no tiene sentido. Entonces, reformulamos:

> El mejor conjunto de parámetros será el que haga que **los valores absolutos** de los residuos sean lo más chicos posibles.

Bien. Pero ahora se plantea la pregunta. ¿Los residuos de qué puntos? Fíjense que si agarro dos puntos cualquiera, con una recta puedo pasar exactamente por ellos y lograr que los residuos de esos puntos sean exactamente cero. Pero seguro que eso no es lo que queríamos decir, no?

La idea es que sea lo más chico posible "para todos" los puntos… Lo más parecido que podemos hacer a esto es:

> El mejor conjunto de parámetros será el que haga que **el promedio de los valores absolutos** de los residuos sean lo más chicos posibles.

Ahora sí, llegamos a la definición del **error absoluto promedio (o medio)**, MAE, por sus siglas en inglés, que matemáticamente se escribe:

$$\text{MAE} = \frac{1}{N}\left(|r_1| + |r_2| + \cdots + |r_N|\right) = \frac{1}{N}\sum_{i=1}^{N}|r_i| \quad .$$

Por razones que escapan un poco el enfoque del curso, muchas veces se buscan los parámetros que minimizan el promedio de los residuos al cuadrado. Como el cuadrado es una función creciente, cuando se minimice uno, se minimiza el otro. Esto define el **error cuadrático promedio**, MSE:

$$\text{MSE} = \frac{1}{N}\left(r_1^2 + r_2^2 + \cdots + r_N^2\right) = \frac{1}{N}\sum_{i=1}^{N}(r_i)^2 \quad .$$

Como se dijo arriba, bajo una serie de suposiciones, existe una manera matemática exacta de encontrar los valores de los parámetros que minimizan estas funciones

1.6 Uso de las funciones de `scikit-learn`

Tomemos ahora un modelo de regresión lineal de `sklearn` y utilicémoslo para ajustar estos datos. Para simplificar, cambiaremos primero los nombres de las variables relevantes.

Además, las variables predictoras tienen que ir en formato matricial: los datos en cada fila y las covariables en cada columna. Esa matriz es la **matriz de diseño**.

[24]:
```
# Variables predictoras (en una matriz, por eso la linea al final dle nombre)
X_ = df_filtro.sup_total.values.reshape(-1, 1)
t = df_filtro.precio.values

print(X_.shape, t.shape)
```

```
(47, 1) (47,)
```

Instanciemos el regresor lineal y ajustemos los datos.

[25]:
```
from sklearn.linear_model import LinearRegression

# Instanciemos el modelo (le damos fit_intercept=True, para que ajuste también␣
 ↪omega0)
lr = LinearRegression(fit_intercept=True)

# Ajustamos (como siempre, con el método fit)
lr.fit(X_, t)
```

[25]:
```
LinearRegression()
```

Nota fuera de contexto.

Estamos resolviendo las ecuaciones normales, ni más ni menos.

$$\begin{array}{rcl} \hat{\omega_1} & = & \sum_{i=0}^{N}(x_i - \bar{X})(t_i - \bar{T}) \left[\sum_{i=0}^{N}(x_i - \bar{X})^2\right]^{-1} \\ \hat{\omega_0} & = & \bar{T} - \hat{\omega_1}\bar{X} \end{array}$$

1.6.1 Los parámetros

Podemos ver el valor de los parámetros encontrados. El parámetro que no acompaña a ninguna variable (ω_0) está en el atributo `intercept_`. El resto (en este caso solo ω_1) están en el atributo `coef_`.

[26]:
```
print('omega_0 = {:.3f}\n omega_1 = {:.3f}'.format(lr.intercept_, lr.coef_[0]))
```

```
omega_0 = 77672.600
 omega_1 = 1460.681
```

Interpretemos estos números...

1.7 Evaluación del modelo.

Ahora que el modelo está ajustado (¡decir "entrenado" es demasiado en este contexto!), vamos a analizar si las cosas han funcionado como se esperaba.

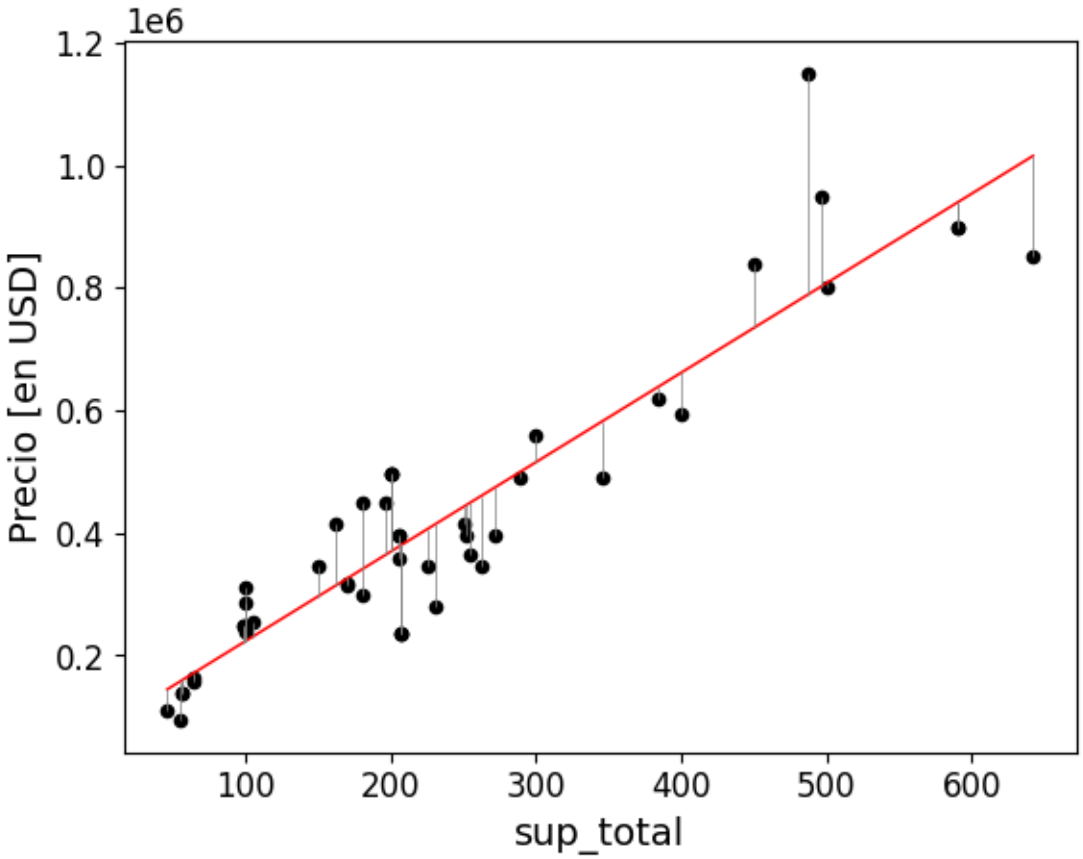
En esta sección evaluaremos si el modelo funciona como se esperaba, y obtendremos algunas ideas sobre cómo mejorarlo.

Antes de empezar, vamos a representar el modelo obtenido junto con los datos.

Para ello, necesitamos obtener las predicciones que hace el modelo en una serie de puntos. Reucerden que para calcular las predicciones, todos los estimadores en `sklearn` usan el método `predict`.

[27]:
```
# Plot de los datos
plot_data(df_filtro)

# Calculamos la recta como antes, pero usando los parámetros del ajuste
y = lr.intercept_ + lr.coef_[0] * x

# Otra forma es con el método predict
y = lr.predict(x.reshape(-1, 1))

# Graficamos
plt.gcf().axes[0].plot(x, y, 'r-', lw=1)


## Agregamos residuos

# predicciones del modelo en los puntos conocidos
pred = lr.predict(X_)

# Grafico líneas verticales
plt.vlines(x=df_filtro.sup_total.values,
           ymin=df_filtro.precio.values,
           ymax=pred, color='0.5', lw=0.5)
```

[27]:
```
<matplotlib.collections.LineCollection at 0x7f62e57a2710>
```



Métricas

Error cuadrático medio La primera forma de evaluar un modelo es calcular la métrica que se utilizaron para ajustar los parámetros del modelo, la MAE y MSE.

Ambas están implementadas en `sklearn`, en el paquete `metrics`, y se usan de manera idéntica: se les pase el valor de la variable target y lo que el modelo predice para esos datos.

[28]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error

print('MAE [K$] = {:.2f}'.format(mean_absolute_error(t, pred)/1e3))
print('MSE [$^2]= {:.2f}'.format(mean_squared_error(t, pred)))
```

```
MAE [K$] = 70.91
MSE [$^2]= 9112824894.87
```

Como las unidades del MSE son dólares al cuadrado, muchas veces se reporta la raíz del error cuadrático medio, RMSE

[29]:
```
print('RMSE [K$]= {:.2f}'.format(np.sqrt(mean_squared_error(t, pred))/1e3))
```

```
RMSE [K$]= 95.46
```

Vemos que las métricas no dan el mismo número, pero sí estamos seguros de que ambas son mínimas.

Coeficiente de determinación Otra métrica muy frecuentemente usada para evaluar un modelo es el coeficiente de determinación:

$$R^2 = \frac{SC_{\text{tot}} - SC_{\text{res}}}{SC_{\text{tot}}} \ ,$$

que refleja la parte de la varianza de los datos que el modelo alcanza a explicar. Esto se puede ver de la expresión (no obvia):

$$\underbrace{\sum_{i=1}^{N} (t - \bar{t})^2}_{SC_{\text{tot}}} = \underbrace{\sum_{i=1}^{N} (t - y)^2}_{SC_{\text{res}}} + \underbrace{\sum_{i=1}^{N} (y - \bar{t})^2}_{SC_{\text{reg}}} \ .$$

Se puede ver que R^2 toma valores entre cero y uno. El paquete `sklearn.metrics` tiene una implementación de esta métrica: `r2_score`. Además, es la métrica por defecto de muchos algoritmos de regresión, bajo el método `score`.

[30]:
```
from sklearn.metrics import r2_score
print('R^2 (conjunto de entrenamiento) = {:.3f}'.format(r2_score(t, pred)))
```

```
R^2 (conjunto de entrenamiento) = 0.842
```

Esto significa que alrededor del 84% de la varianza de los datos desaparece con el modelo lineal simple.

Una linda propiedad es que R^2 es el cuadrado del coeficiente de Pearson entre X y t.

Por ahora, no estamos separando el conjunto de entrenamiento de un conjunto de testeo. Esto es tema de la segunda parte del módulo. Pero sepan que todos los valores de las métricas que conseguimos son en relidad optimistas. ***

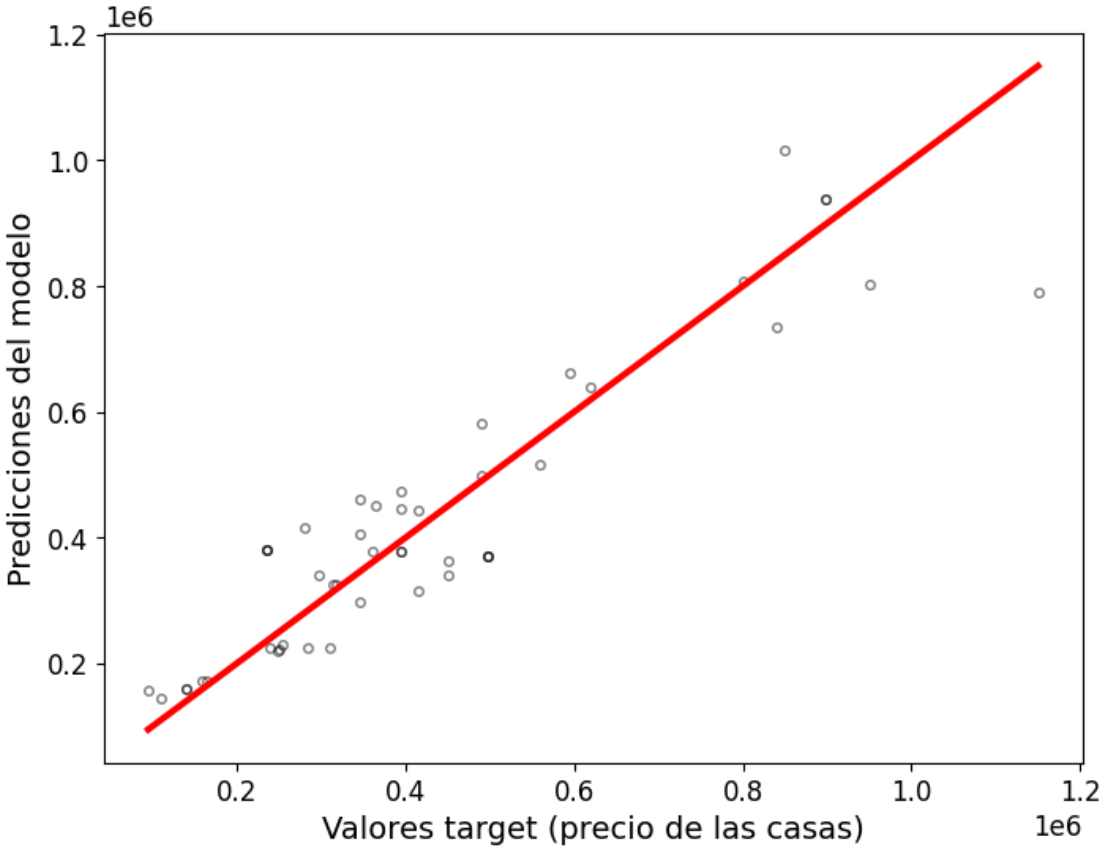
Predicciones vs. Targets Otra buena forma de evaluar el modelo es visualizando los datos y las predicciones.

En el gráfico de abajo está la predicción del modelo (eje y) vs. el valor de la variable target (eje x).

[31]:
```
# Graficar las predicciones contra los valores target.
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
ax.plot(t, pred, 'ok', mfc='None', ms=4, alpha=0.5)

ax.set_xlabel('Valores target (precio de las casas)')
ax.set_ylabel('Predicciones del modelo')

# Agregar la línea identidad
```

```
ax.plot([t.min(), t.max()], [t.min(), t.max()], color='r', lw=3)
```

[31]: [<matplotlib.lines.Line2D at 0x7f62e1488a90>]



Pregunta. ¿Creen que el modelo está funcionando correctamente? Si no, ¿qué podrían señalar como deficiencia?

Residuos También es fundamental explorar el comportamiento de los residuos del modelo.

[32]:
```
# Calculamos los residuos a partir de la variable target y las predicciones del␣
↪modelo
res = t - pred
```

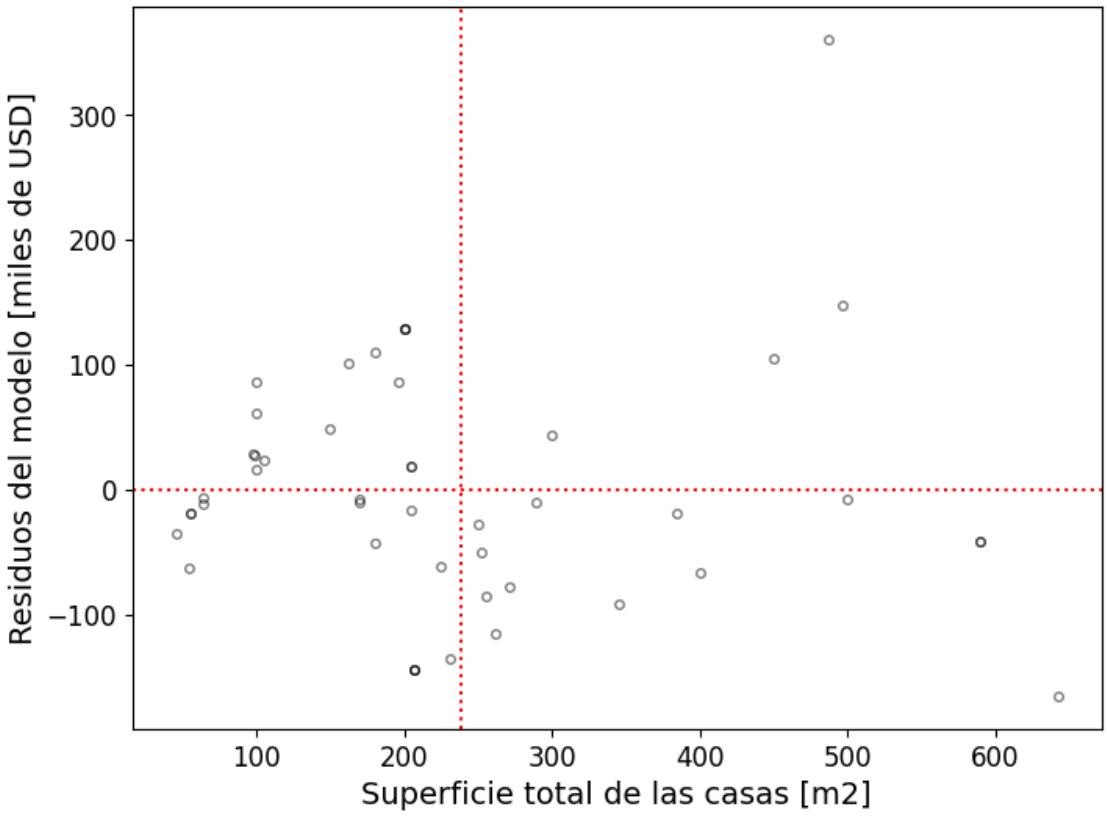
Residuos vs. Predictores Hay varios gráficos que son importantes.

En primer lugar, ver los residuos en función de la variable predictora:

[33]:
```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
ax.plot(X_, res/1e3, 'ok', mfc='None', ms=4, alpha=0.5)
```

```
# Agragamos líneas vertical y horizontal
ax.axhline(0, color='r', ls=':')
ax.axvline(X_.mean(), color='r', ls=':')

# Add axes labels
ax.set_xlabel('Superficie total de las casas [m2]')
ax.set_ylabel('Residuos del modelo [miles de USD]')
```

[33]: `Text(0, 0.5, 'Residuos del modelo [miles de USD]')`



Este gráfico es útil para identificar las curvaturas y otras características que podrían indicar que el modelo elegido o las variables predictoras son inadecuados.

Nota fuera de contexto

Las ecuaciones normales implican dos cosas: 1. los residuos tienen media cero. 2. el coeficiente de Pearson entre la variable X y los residuos es cero.

Vamos a verificar estas propiedades con el código de acá abajo.

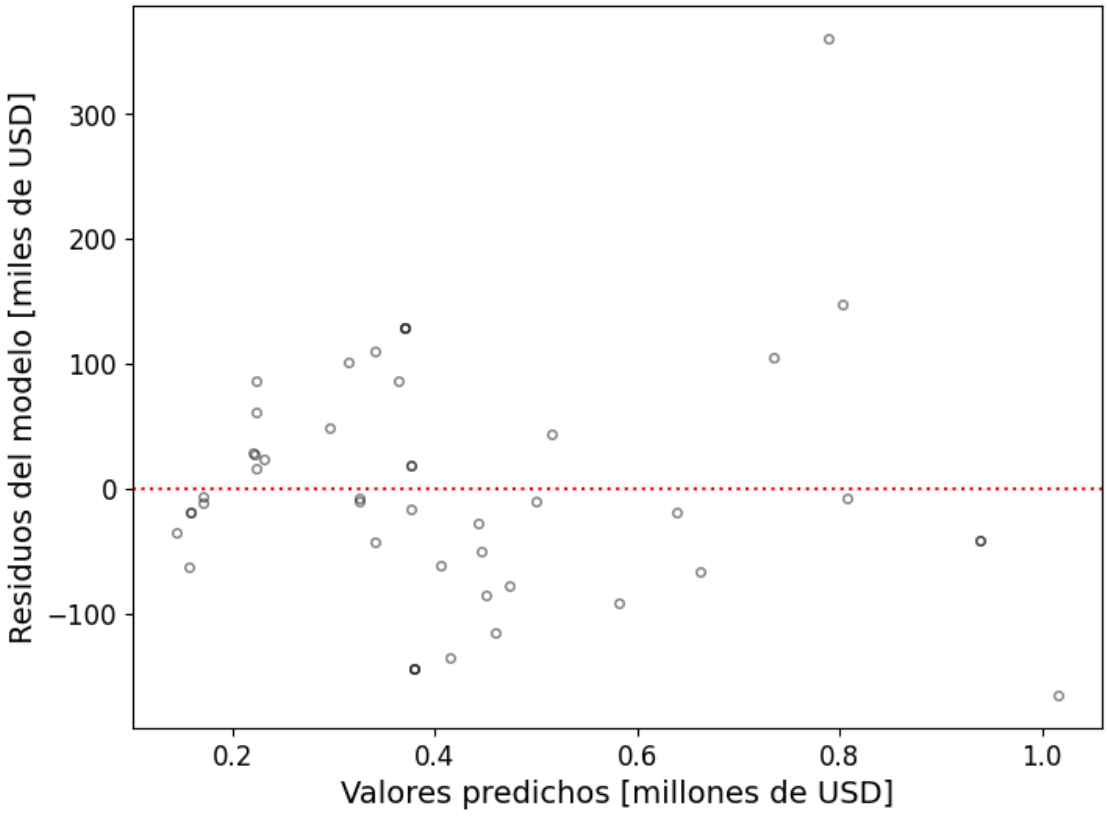
[34]:
```
xmean = X_.mean()
resmean = res.mean()
print('$\\bar{{y - t}}$ = {:.12f}'.format(resmean))
print('$rho(X, res)$ = {:.12f}'.format(np.corrcoef(X_.flatten(), res)[0, 1]))
```

```
$\bar{y - t}$ = -0.000000000050
$rho(X, res)$ = -0.000000000000
```

Residuos vs. Valores predichos Un gráfico similar es el de los residuos en función de los valores predichos.

[35]:
```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
ax.plot(pred/1e6, res/1e3, 'ok', mfc='None', ms=4, alpha=0.5)

ax.axhline(0, color='r', ls=':')
ax.set_xlabel('Valores predichos [millones de USD]')
ax.set_ylabel('Residuos del modelo [miles de USD]')
```

[35]:
```
Text(0, 0.5, 'Residuos del modelo [miles de USD]')
```



Este gráfico es su amigo, porque puede construirse incluso en el caso de la regresión lineal múltiple.

Al igual que arriba, este gráfico es útil para identificar las curvaturas.

1.8 TEMA AVANZADO: Palanca

La *palanca* (*leverage* en inglés) es un concepto interesante para evaluar ajustes a modelos lineales.

Se puede mostrar que el valor predicho por el modelo para un dado valor x_i puede expresarse como

$$y_i = \hat{\omega_0} + \hat{\omega_1} x_i = \sum_{k=1}^{N} h_{ik} t_k \quad ,$$

es decir como una combinación lineal de los valores de la variable *target*. La suma que aparece en el término de la derecha corre sobre todos los puntos del dataset. Es decir que para saber la predicción de un punto tenemos que sumar los valores de la variable *target* de todos los puntos t_k, pero ponderados por h_{ik}:

$$h_{ik} = \frac{1}{N} + \frac{(x_i - \bar{x})\,(x_k - \bar{x})}{S_{xx}} \quad ,$$

con

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

y

$$S_{xx} = \sum_{i=1}^{N} (x_i - \bar{x})^2 \quad .$$

El elemento h_{ik} nos dice cuánto contribuye un dado punto a la predicción y_i.

Vamos a calcular la matriz cuyos elementos ij son h_{ij}. Esta matrix se llama matriz sombrero o sombrerera.

[36]:
```
from scipy.linalg import cho_factor, cho_solve

def hat_matrix(X, include_bias=True):
    """
    Compute hat matrix for design matrix X.

    :param np.array X: design matrix of dimensions (n x d),
    where n is the number of observations and d is the number of
    features.
    :param bool include_bias: if True (default), then include a bias column,
    in design matrix X (i.e. a column of ones - acts as an
    intercept term in a linear model).
    """
    if include_bias:
        X = np.hstack([np.ones([len(X), 1]), X])

    A = np.matmul(X.T, X)
```

```
    LL = cho_factor(A)
    return np.matmul(X, cho_solve(LL, X.T))

# Define HAT matrix, whose diagonal are the leverage values
h = hat_matrix(X_)
```

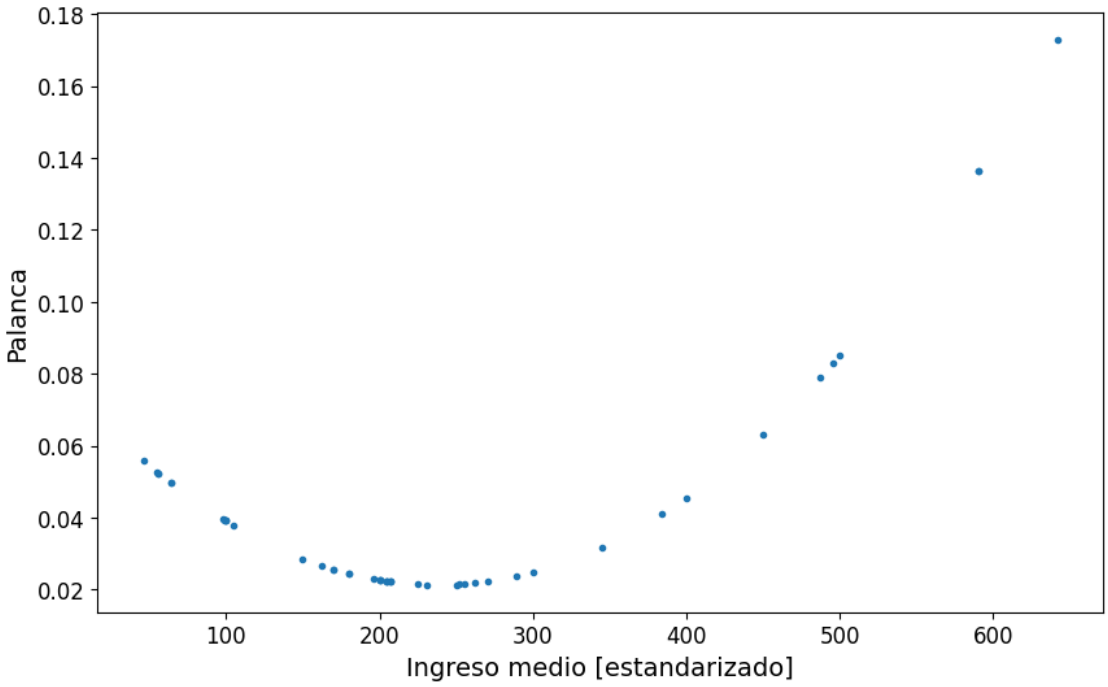
Pregunta. ¿Cuál es la forma de la matriz sombrerera? Verifiquen con el atributo `shape`.

Esta matriz nos dice cómo influye cada punto en la predicción de todos los demás.

En particular, podemos trazar la influencia de cada observación sobre sí misma. Esto esto es la palanca de la observación, y se obitene como la diagonal de la matrix H. Noten que esto es independiente de t; sólo se refiere a los valores de la variable predictora).

[37]:
```
leverage = np.diag(h)
plt.figure(figsize=(10, 6))
plt.plot(X_.flatten(), leverage, '.')
plt.xlabel('Ingreso medio [estandarizado]')
plt.ylabel('Palanca')
```

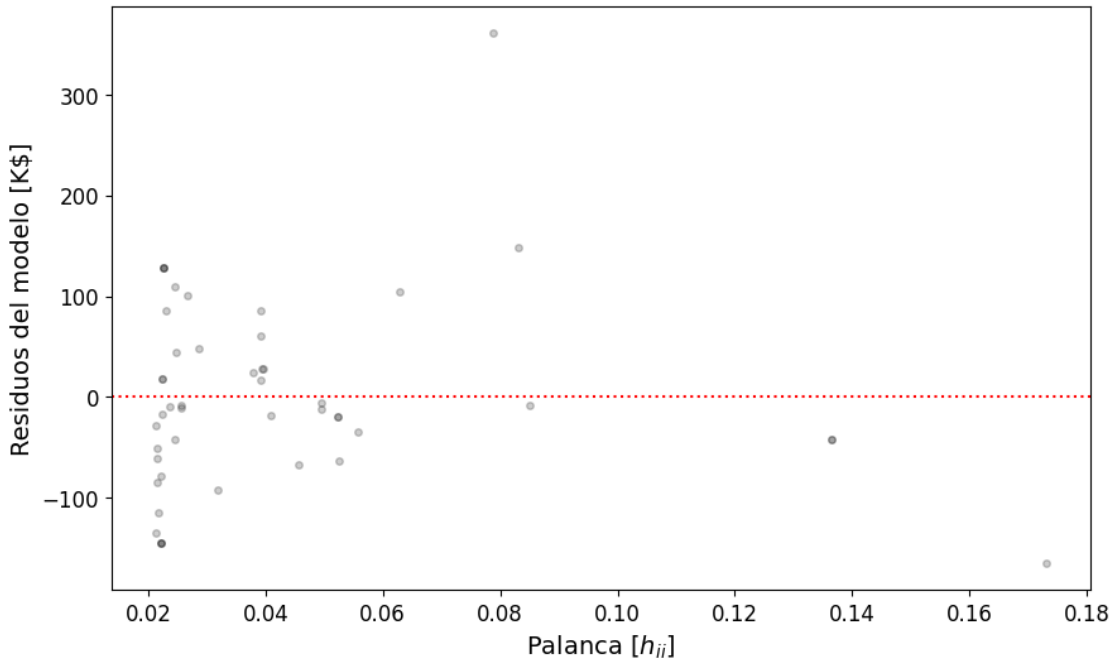
[37]:
```
Text(0, 0.5, 'Palanca')
```



Residuos vs. Palanca Un gráfico muy informativo es el de los residuos como función de la palanca.

[38]:
```
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111)

ax.plot(leverage, res/1e3, 'ok', ms=4, mfc=None, alpha=0.2)
ax.axhline(0, color='r', ls=':')
ax.set_xlabel('Palanca [$h_{ii}$]')
ax.set_ylabel('Residuos del modelo [K$]')
```

[38]:
```
Text(0, 0.5, 'Residuos del modelo [K$]')
```

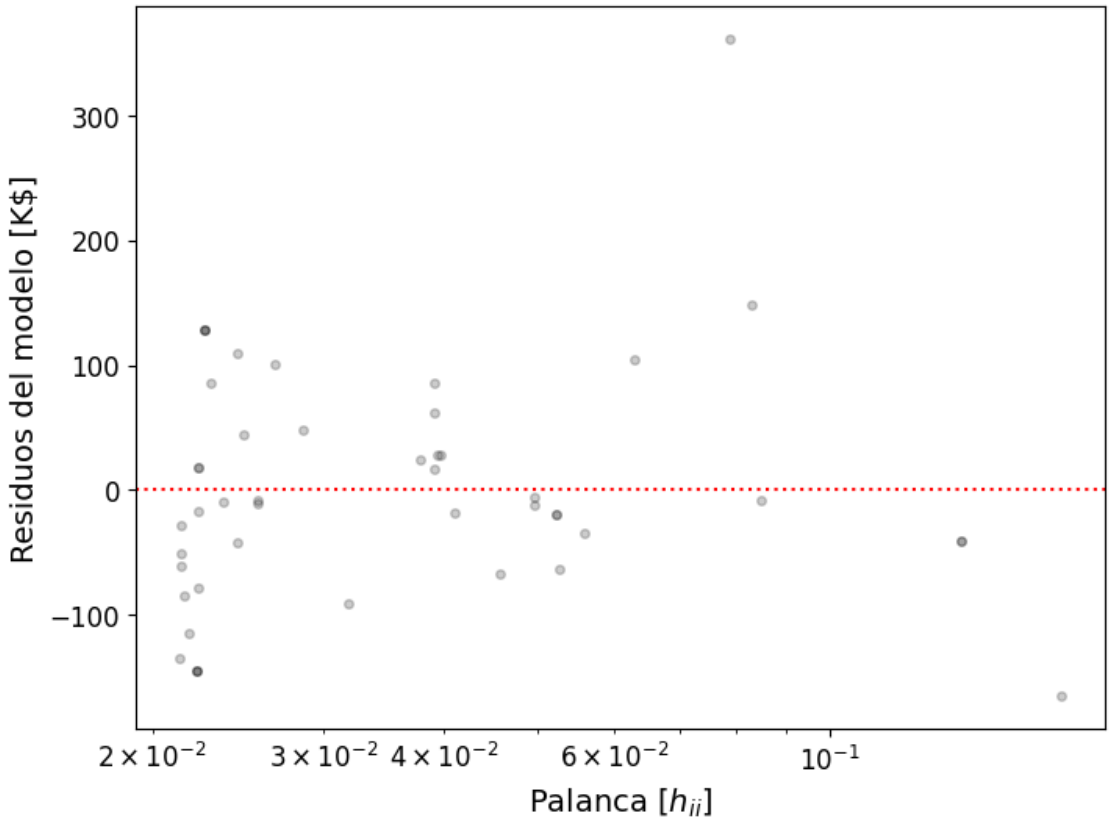


En este gráfico, podemos ver que hay unos pocos puntos con una palanca elevada y residuos distintos de cero. Estos puntos suelen ser muy relevantes para el ajuste, ya que tiran del modelo hacia ellos con más fuerza que el resto. Como hay tantos puntos en total en este caso, no esperamos que estos valores influyan mucho en el ajuste final, pero en una situación con menos datos, estos datos pueden ser muy influyentes.

Cuando los valores de palanca están muy pegados, podemos ver el gráfico en escala logarítimica.

[39]:
```
# Mismo gráfico en escala log
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)

ax.semilogx(np.diag(h), res/1e3, 'ok', ms=4, mfc=None, alpha=0.2)
ax.axhline(0, color='r', ls=':')
ax.set_xlabel('Palanca [$h_{ii}$]')
ax.set_ylabel('Residuos del modelo [K$]')
```

[39]: `Text(0, 0.5, 'Residuos del modelo [K$]')`



Otra propiedad de las palancas Se puede probar que la suma de las *palancas* son un valor constante:

$$\sum_{i=1}^{N} h_{ii} = 2 \ .$$

Por lo tanto, cada vez que un punto gana en palanca, el resto pierde. En general, la suma de todos los valores de *palanca* debe ser igual a las dimensiones del parámetro, ω.

Verifiquemos la propiedad con el código

[40]:
```
print('La suma de las palancas es {:.2f}.'.format(np.sum(np.diag(h))))
```

```
La suma de las palancas es 2.00.
```

1.8.1 TEMA AVANZADO: Residuos estandarizados

En realidad, los gráficos de los residuos pueden ser engañosos, porque la varianza de los residuos no es constante (¡la varianza de los *errores* sí lo es!). Se puede demostrar que la varianza de los

residuos es

$$\mathrm{var}(r_i) = \sigma^2(1 - h_{ii}) \ ,$$

donde h_{ii} es la palanca del punto i. Si no conocemos la dispersión del término de error, σ^2, podemos usar el estimador:

$$\widehat{\sigma^2} = \frac{1}{N-2}\sum_{i=1}^{N}(y_i - t_i)^2 \ .$$

Podemos entonces calcular los **residuos estandarizados** que tienen la misma varianza para todo el rango de la variable predictora X:

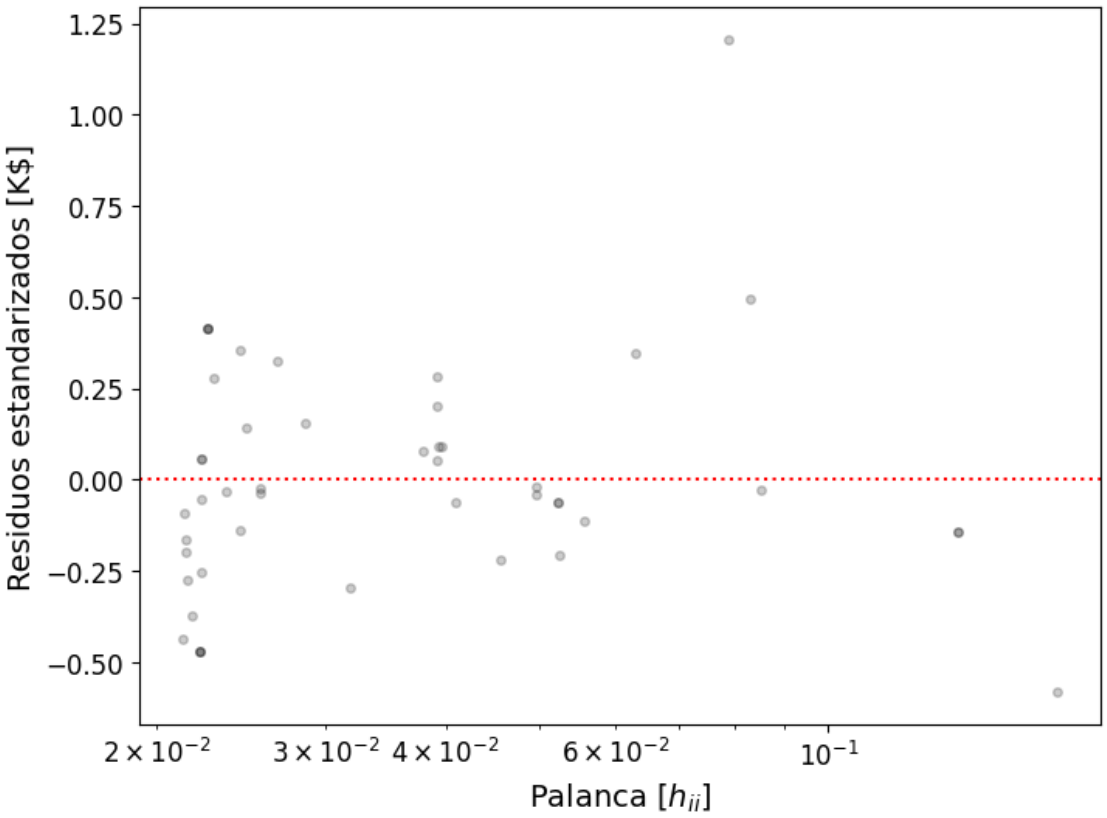
$$\text{st-res}_i = \frac{y_i - t_i}{\sqrt{\widehat{\sigma^2}\,(1 - h_{ii})}} \ .$$

[41]:
```
# Calculamos residuos estandarizados
stres = (t - pred) / np.sqrt(res.std(ddof=2) * (1 - leverage))
```

Y podemos usarlos para hacer gráficos análogos a los de arriba.

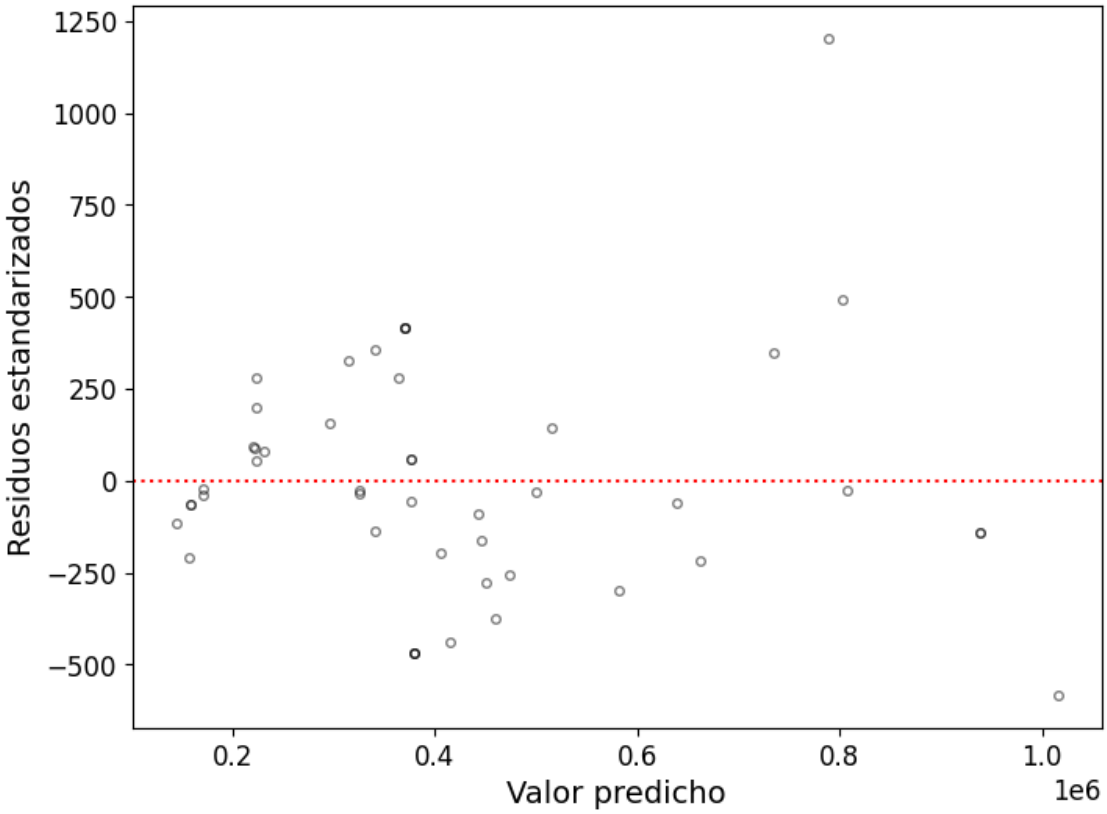
[42]:
```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)

ax.semilogx(np.diag(h), stres/1e3, 'ok', ms=4, mfc=None, alpha=0.2)
ax.axhline(0, color='r', ls=':')
ax.set_xlabel('Palanca [$h_{ii}$]')
ax.set_ylabel('Residuos estandarizados [K$]')
```

[42]:
```
Text(0, 0.5, 'Residuos estandarizados [K$]')
```



[43]:
```
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
ax.plot(pred, stres, 'ok', mfc='None', ms=4, alpha=0.5)

ax.axhline(0, color='r', ls=':')
ax.set_xlabel('Valor predicho')
ax.set_ylabel('Residuos estandarizados')
```

[43]:
```
Text(0, 0.5, 'Residuos estandarizados')
```



1.9 Errores en los parámetros

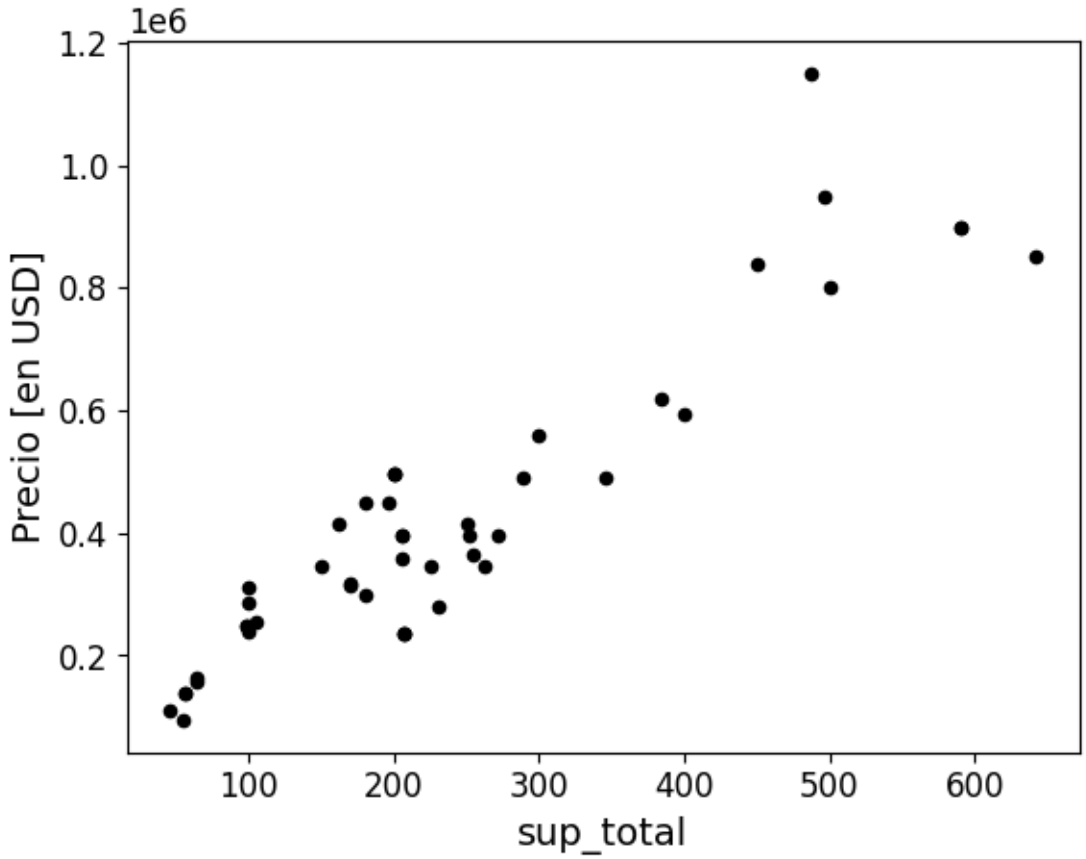
Los valores de los parámetros obtenidos dependen de las características particulares de este dataset.

Piensen qué pasaría si tuviéramos acceso a un dataset similar, de propiedades del mismo tipo, en el mismo barrio, pero de otra inmobiliaria. Seguramente si repetimos el proceso encontraríamos parámetros similares, pero no exactamente igual.

La idea de que estos parámetros entonces tienen un cierto "margen de error" es clave. ¿Pero como no tenemos acceso a otros datasets, cómo podemos estimar el error de nuestros parámetros?

Una opción es repetir el ajuste cambiando ligeramente el conjunto de datos que tenemos.

[44]:
```
plot_data(df_filtro)
```



[45]:
```python
# Repetir el experimento varias veces
Nrepet = 10000

# Creo listas vacias para ir llenando
w0_lista = np.zeros(Nrepet)
w1_lista = np.zeros(Nrepet)

for i in range(Nrepet):
    # Elijo al azar 53 números (puede haber repetidos)
    irandom = np.random.randint(0, len(df_filtro)-1, size=len(df_filtro))

    # Construyo un nuevo set de datos a partir de estos números
    X_new = X_[irandom]
    t_new = t[irandom]

    # Ajusto el modelo nuevamente con estos datos
    lr.fit(X_new, t_new)

    # Meto el resultado del ajuste en la lista
```

```
    w0_lista[i] = lr.intercept_
    w1_lista[i] = lr.coef_[0]
```

Con las listas de parámetros, podemos calcular algunas métricas. Para eso, nada como usar **pandas**

[46]:
```
parametros = pd.DataFrame(data={'w0': w0_lista, 'w1': w1_lista})
parametros.describe()
```

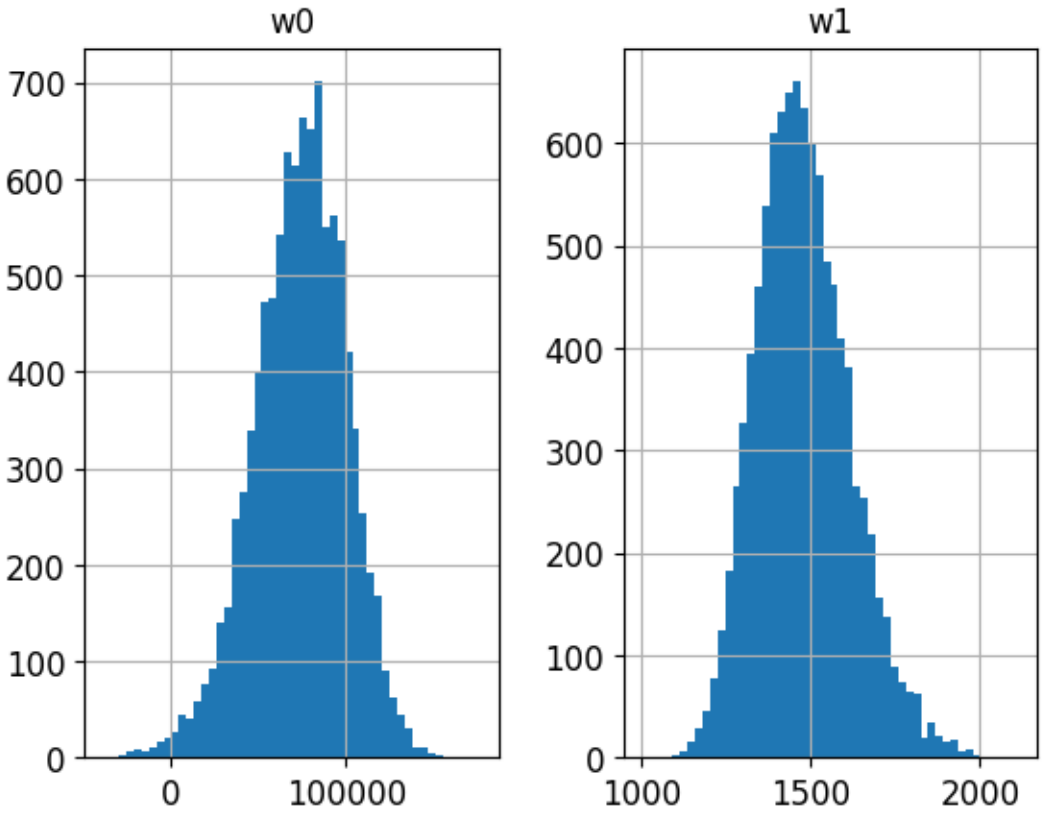
[46]:

	w0	w1
count	10000.000000	10000.000000
mean	73884.580893	1480.273691
std	26392.038894	138.297571
min	-38630.718225	1003.512319
25%	56787.480737	1382.304442
50%	75514.684055	1469.696633
75%	92477.170496	1568.788021
max	176949.553944	2114.403546

Podemos también graficar los resultados como histogramas

[47]:
```
parametros.hist(bins=50)
```

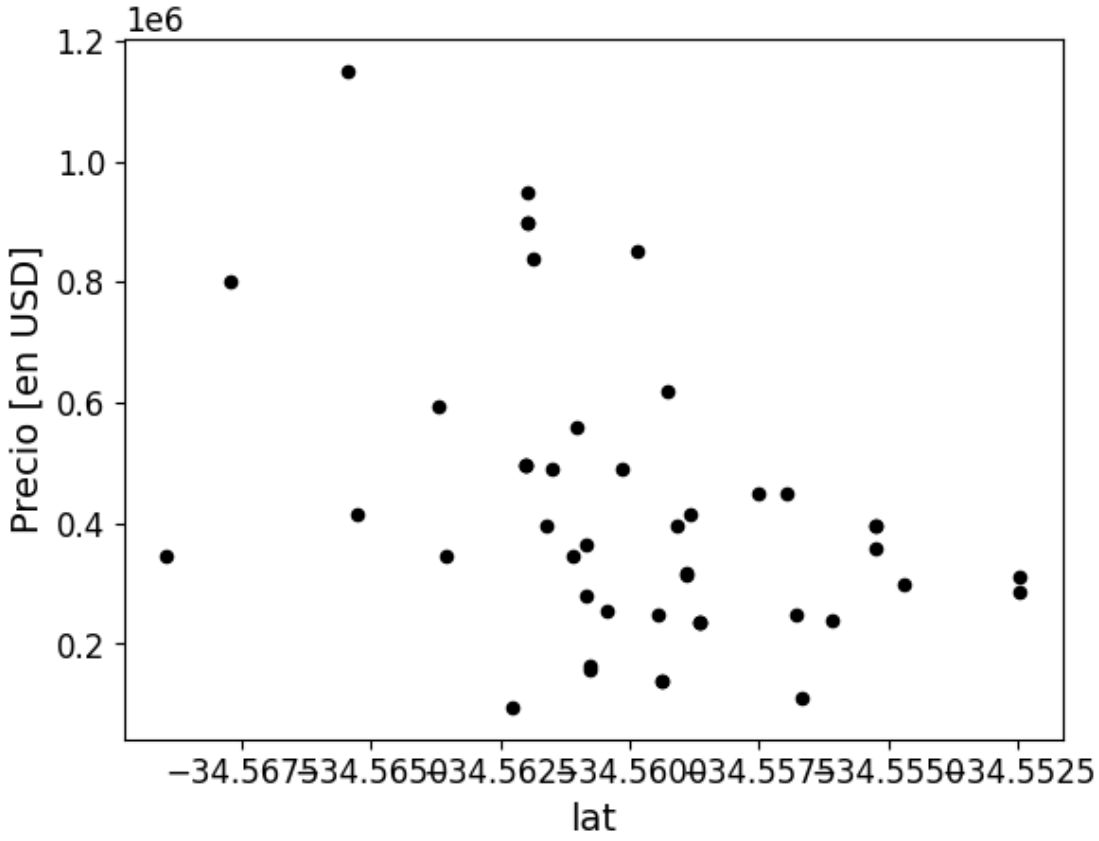
[47]:
```
array([[<Axes: title={'center': 'w0'}>, <Axes: title={'center': 'w1'}>]],
      dtype=object)
```



1.9.1 Significancia

¿Qué creen que pasaría si repetimos todo el procedimiento de arriba pero usando una variable que no es relevante pare el análisis?

[48]:
```
plot_data(df_filtro, xvar='lat')
```



[49]:
```
X_lat = df_filtro.lat.values.reshape(-1, 1)
```

[50]:
```
# Repetir el experimento varias veces
Nrepet = 10000

# Creo listas vacias para ir llenando
w0_lista = np.zeros(Nrepet)
w1_lista = np.zeros(Nrepet)

for i in range(Nrepet):
    # Elijo al azar 53 números (puede haber repetidos)
```

```
    irandom = np.random.randint(0, len(df_filtro)-1, size=len(df_filtro))

    # Construyo un nuevo set de datos a partir de estos números
    X_new = X_lat[irandom]
    t_new = t[irandom]

    # Ajusto el modelo nuevamente con estos datos
    lr.fit(X_new, t_new)

    # Meto el resultado del ajuste en la lista
    w0_lista[i] = lr.intercept_
    w1_lista[i] = lr.coef_[0]
```

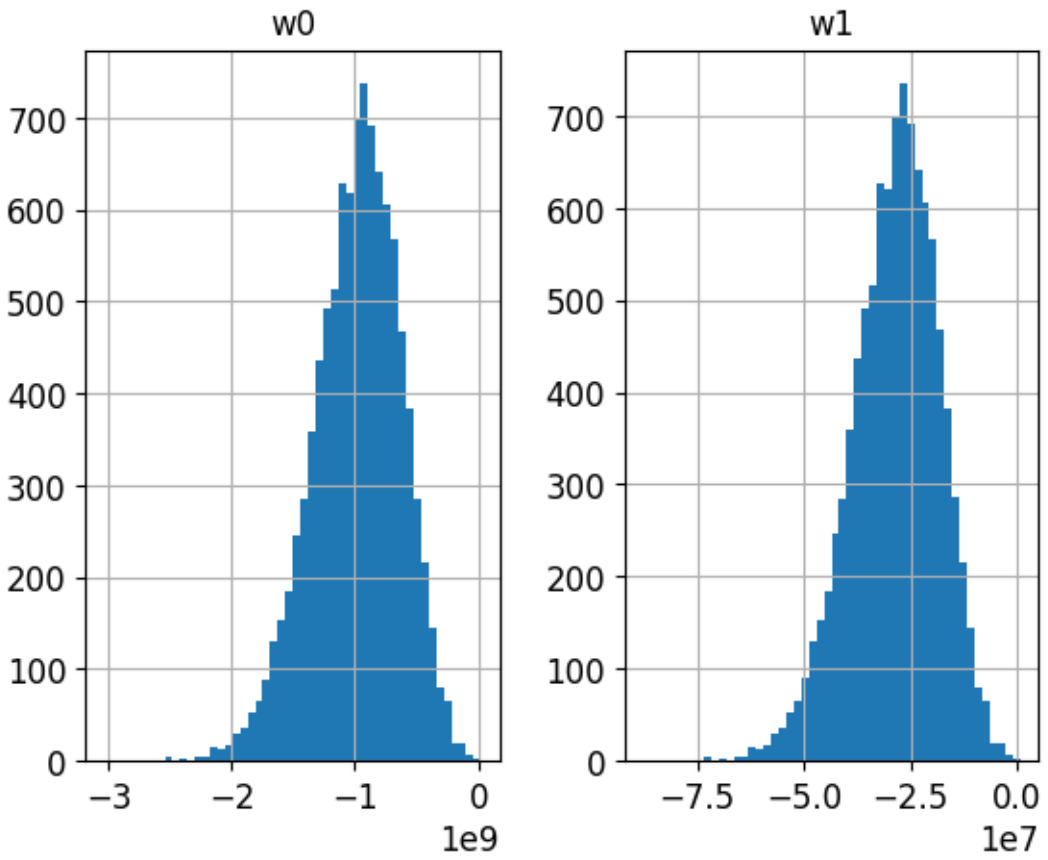
[51]:
```
parametros_lat = pd.DataFrame(data={'w0': w0_lista, 'w1': w1_lista})
parametros_lat.describe()
```

[51]:

	w0	w1
count	1.000000e+04	1.000000e+04
mean	-9.864601e+08	-2.855548e+07
std	3.529441e+08	1.021300e+07
min	-3.028868e+09	-8.765078e+07
25%	-1.210584e+09	-3.504034e+07
50%	-9.600640e+08	-2.779190e+07
75%	-7.373915e+08	-2.134854e+07
max	2.304254e+07	6.567295e+05

[52]:
```
parametros_lat.hist(bins=50)
```

[52]:
```
array([[<Axes: title={'center': 'w0'}>, <Axes: title={'center': 'w1'}>]],
      dtype=object)
```



2 Su turno. Modelo lineal múltiple

- Hagan un análisis similar eligiendo una variable predictiva diferente.
- Compare el rendimiento del modelo lineal simple que utiliza "sup_total" con el que han elegido.
- Ahora viene la parte bonita. Incluya en la matriz X una nueva columna, de modo que combine la "sup_total" con su nueva variable, y realice un análisis similar para este "modelo lineal múltiple". ¿Qué gráficos puedes hacer todavía? **Tip**: tal vez se pueda crear una variable `sup_fondo`…

Tip: Para incluir una nueva variable predictora en X (la matriz de diseño); cambien YOUR_FEATURE por el nombre de la columna que desean incluir.

[54]:
```
#X = df_filtro.loc[:, ['sup_total', 'YOUR_FEATURE1', 'YOUR_FEATURE2',␣
↪'YOUR_FEATURE3', ...]].values
```

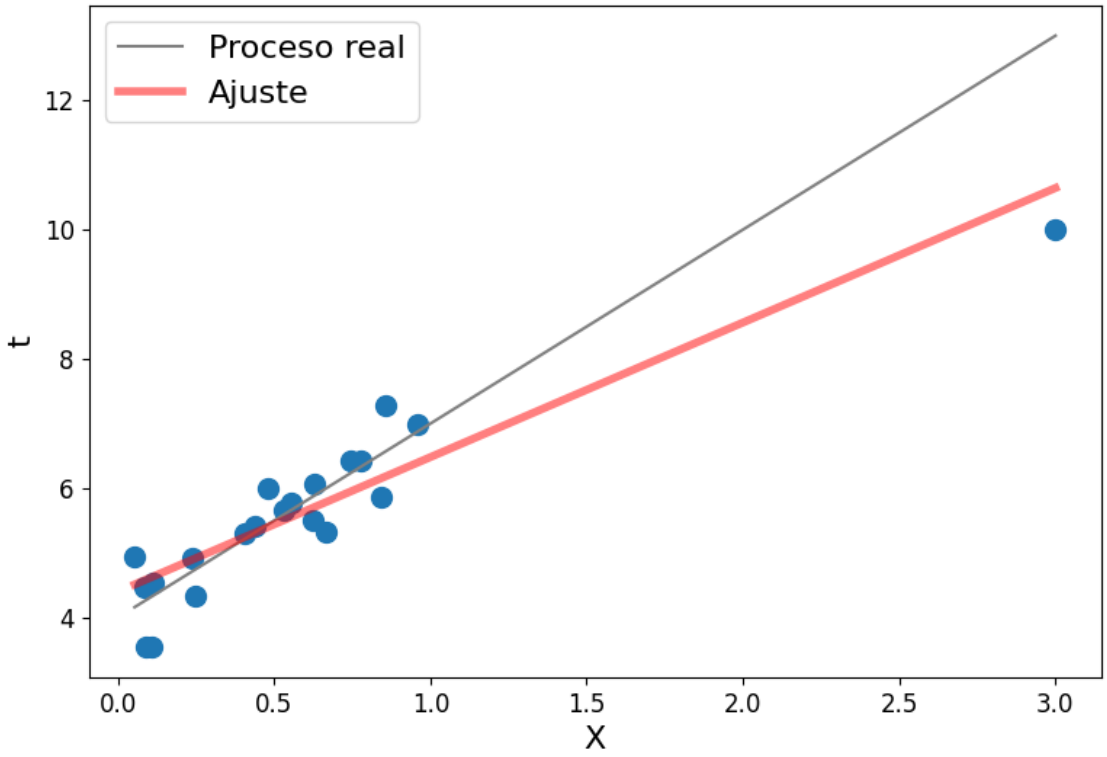
3 Extra! Un procedimiento para detectar outliers

Es importante comprobar la presencia de valores atípicos (*outliers*). Si por alguna razón los datos tienen algún valor que no proviene del proceso real que estamos modelando, esto puede modificar los resultados del ajuste y, en particular, sesgarlo fuertemente.

Por ejemplo, tomemos los datos sintéticos y hagamos que uno de los puntos se convierta en un valor atípico, fijando un valor a mano. Para que el efecto sea claro, tenemos que hacerlo en un punto que tenga una alta palanca. Veamos…

[55]:
```
def ground_truth(x):
    return 3*x + 4 #+ 0.1*x**2

# Función que genera un dataset al azar en base a la "ground truth"
def make_default_dataset(real_process, sigma=0.5, high_leverage=None,␣
 ↪random_seed=20210331):
    # Fijo seed
    np.random.seed(random_seed)

    # Defino vector de x
    x = np.random.rand(20)

    # Por si quiero otra nube de puntos
#     x2 = np.random.rand(4) + 2.5
#     x = np.concatenate([x, x2])

    x = np.sort(x)

    # Agrego un punto con mucha palanca
    if high_leverage is not None:
        high_leverage_x = np.array(high_leverage)
        x = np.append(x, high_leverage_x)

    x_plot = np.linspace(x.min(), x.max(), 100).reshape(-1, 1)

    # Error
    t = real_process(x) + np.random.randn(len(x)) * sigma

    return x, t, x_plot

x, t, x_plot = make_default_dataset(ground_truth, high_leverage=3)

# Recalculo la palanca
h = np.diag(hat_matrix(x.reshape(-1, 1)))

# Defino una observación outlier
t[-1] = 10
```

[56]:
```
plt.figure(figsize=(9, 6))
plt.plot(x, t, 'o', ms=10)
plt.plot(x_plot, ground_truth(x_plot), '-', color='0.5', label='Proceso real')
plt.xlabel('X', fontsize=16)
plt.ylabel('t', fontsize=16)

# Ajuste y grafico
lr = LinearRegression()
lr.fit(x.reshape(-1, 1), t)
plt.plot(x_plot, lr.predict(x_plot), '-r', lw=4, alpha=0.5, label='Ajuste')
plt.legend(loc=0, fontsize=16)

# Calculo residuos
res = t - lr.predict(x.reshape(-1, 1))
```



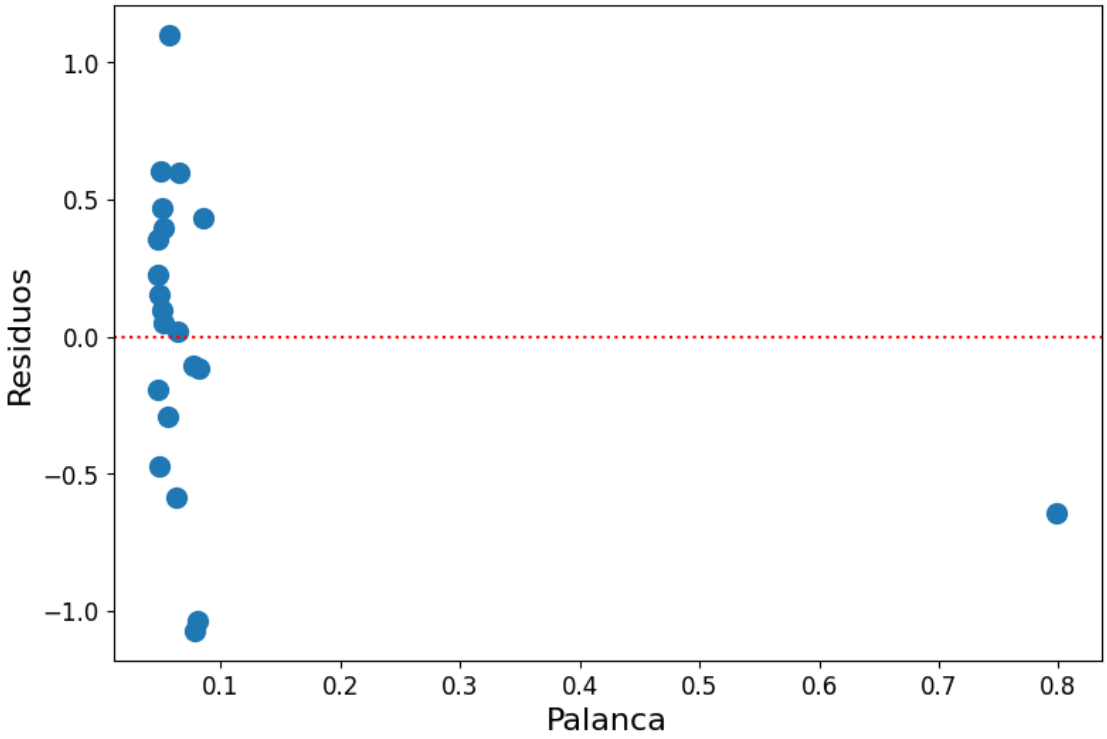
Por supuesto, si no conociéramos el proceso real, identificar el punto en $x = 3,0$ como *outlier* sería más difícil. Y recordemos que en muchas dimensiones todo es más difícil.

Veamos el gráfico de residuos vs. la palanca, a ver si nos da más pistas.

[57]:
```
plt.figure(figsize=(9, 6))
plt.plot(h, res, 'o', ms=10)
plt.axhline(0, color='r', ls=':')
```

```
# plt.gca().set_xscale('log')

plt.ylabel('Residuos', fontsize=16)
plt.xlabel('Palanca', fontsize=16)
```

[57]: Text(0.5, 0, 'Palanca')



Aunque el punto atípico está bastante lejos de cero, no es dramáticamente extraño. Está claro lo que ocurre aquí: el *outlier* ya ha modificado la solución y, por tanto, el residuo es relativamente pequeño en ese punto.

¿Cómo podemos hacer entonces? ¿Ideas? ¿Ideas que funcionen en muchas dimensiones?

3.0.1 Entra Validación Cruzada

Una posibilidad es hacer uso de una herramienta que será fundamental en el curso para detectar y evitar el sobreajuste, que se llama validación cruzada.

En este caso la vamos a utilizar para detectar el punto atípico. En cierto modo, también se trata de un sobreajuste, porque el modelo intenta captar todo el conjunto de entrenamiento, en lugar de preocuparse por reproducir la tendencia general, aquí dada por el proceso real (¡que por suerte conocemos en este caso!).

La idea es construir subconjuntos de datos, partiendo del conjunto original, sacando un punto cada

vez. Es decir, se construyen N conjuntos de $N-1$ datos, y se ajusta un modelo para cada uno de esos nuevos datos. A continuación, se compara la predicción realizada por cada modelo para el punto que hemos eliminado con la realizada por el modelo ajustado a todos los datos.

La intuición es que cuando eliminamos el valor atípico que está influyendo en el ajuste, la predicción cambiará drásticamente.

Esto se llama *validación cruzada de uno a uno* (*leave-one-out cross-validation*), porque los puntos se sacan uno a uno. Hay muchos más sabores de validación cruzada, algunos de los cuales cubriremos más adelante en el curso.

Ejercicio. Corran el código de la próxima celda e intenten entender qué hacen `LeaveOneOut` y `LeavePOut`.

[58]:
```
# Implementación de LOOCV en sklearn
from sklearn.model_selection import LeaveOneOut, LeavePOut, cross_val_predict

loo = LeaveOneOut()
# loo = LeavePOut(3)
for train, test in loo.split(x):
    print(train, test)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [0]
[ 0  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [1]
[ 0  1  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [2]
[ 0  1  2  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [3]
[ 0  1  2  3  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [4]
[ 0  1  2  3  4  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [5]
[ 0  1  2  3  4  5  7  8  9 10 11 12 13 14 15 16 17 18 19 20] [6]
[ 0  1  2  3  4  5  6  8  9 10 11 12 13 14 15 16 17 18 19 20] [7]
[ 0  1  2  3  4  5  6  7  9 10 11 12 13 14 15 16 17 18 19 20] [8]
[ 0  1  2  3  4  5  6  7  8 10 11 12 13 14 15 16 17 18 19 20] [9]
[ 0  1  2  3  4  5  6  7  8  9 11 12 13 14 15 16 17 18 19 20] [10]
[ 0  1  2  3  4  5  6  7  8  9 10 12 13 14 15 16 17 18 19 20] [11]
[ 0  1  2  3  4  5  6  7  8  9 10 11 13 14 15 16 17 18 19 20] [12]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 14 15 16 17 18 19 20] [13]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 15 16 17 18 19 20] [14]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 16 17 18 19 20] [15]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 17 18 19 20] [16]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 18 19 20] [17]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 19 20] [18]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 20] [19]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19] [20]
```

A continuación, utilicen la función `cross_val_predict` para calcular la predicción en todos los puntos, pero sólo cuando no estén incluidos en el conjunto de entrenamiento.

[59]:
```
# Cálculo de las predicciones de la regresión lineal dejando cada punto afuera
y_iout = cross_val_predict(lr, x.reshape(-1, 1), t, cv=loo)
```

Verificamos la forma de la salida

[60]:
```
print(y_iout.shape, x.shape)
```

```
(21,) (21,)
```

Por si no se entendió lo que acabamos de hacer, acá hay un código que lo hace a mano.

[61]:
```
y_iout_mano = np.empty_like(t)

for i, [train, test] in enumerate(loo.split(x)):
    x_i = x[train]
    t_i = t[train]

    lr.fit(x_i.reshape(-1, 1), t_i)
    y_ii = lr.predict(x[test].reshape(-1, 1))

    y_iout_mano[i] = y_ii
```

Y podemos verificar que dan lo mismo.

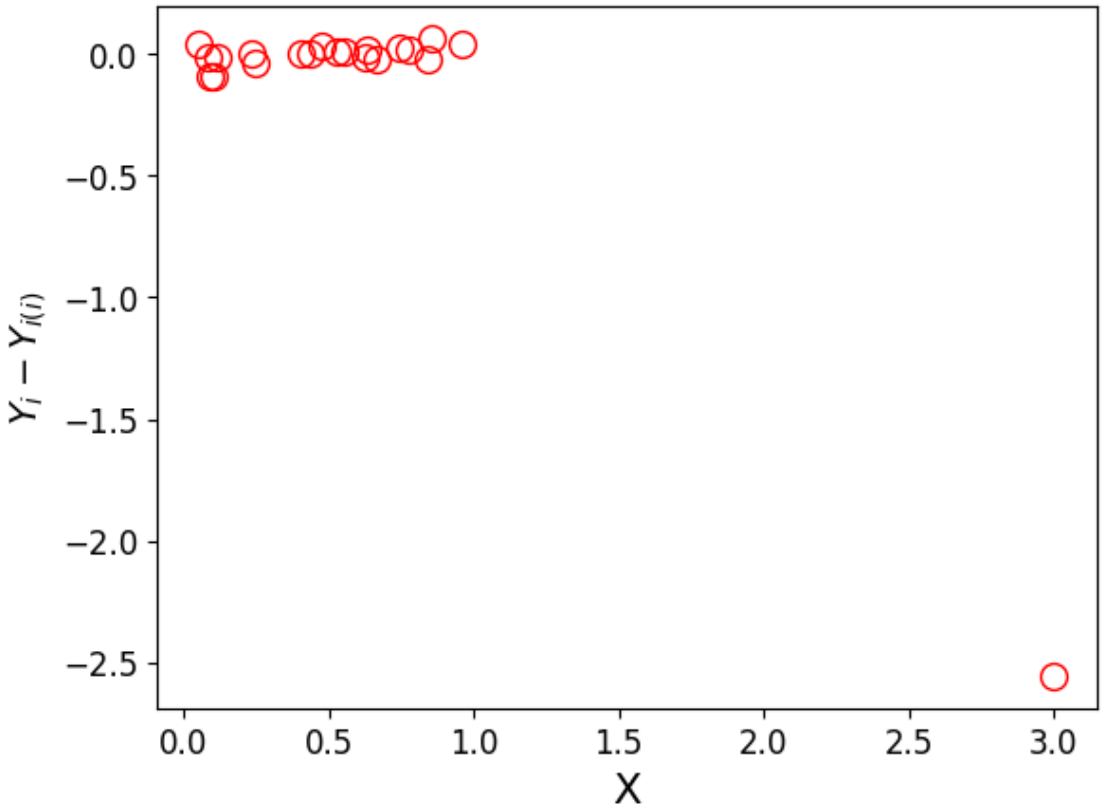
[62]:
```
np.allclose(y_iout, y_iout_mano)
```

[62]:
```
True
```

Por último graficamos la diferencia de las predicciónes para cada punto.

[63]:
```
lr.fit(x.reshape(-1, 1), t)
y = lr.predict(x.reshape(-1, 1))

plt.plot(x, (y - y_iout), 'or', ms=10, mfc='None')
plt.xlabel('X', fontsize=16)
plt.ylabel('$Y_i - Y_{i(i)}$')
```

[63]:
```
Text(0, 0.5, '$Y_i - Y_{i(i)}$')
```



Podemos ver que el valor atípico es claramente visible.

Para casos menos obvios, podemos definir una estadística que se compare con alguna distribución razonable, pero no entraremos en esos detalles aquí.

3.0.2 Relevancia de una observación

Obviamente, un valor atípico como el que acabamos de ver es una observación muy influyente, que determina fuertemente el resultado del ajuste/entrenamiento.

Pero si ese mismo valor atípico hubiera tenido menos influencia, las cosas habrían sido diferentes.

Ejercicio

- Modificar la posición y el valor del *outlier*. Ver qué influencia tiene en cada caso.
- Discutir qué combinación tiene que darse para que una medida sea influyente. ***