Regresión

Alcance de la unidad:

- Reconocer la terminología asociada a la modelación estadística.
- Conocer la regresión lineal y sus fundamentos.
- Interpretar los parámetros estimados en la regresión.
- Conocer y ser capaz de interpretar estadísticos de bondad de ajuste y coeficientes.
- Reconocer los supuestos en los que la regresión tiene sustento teórico.
- Implementar un modelo de regresión con `statsmodels`
- Utilizar transformaciones simples en las variables independientes.
- Implementar un modelo predictivo con `scikit-learn`

En esta unidad aprenderemos sobre la regresión en su variante más general: la regresión lineal. Emplearemos una base de datos sobre ingresos de una empresa y buscaremos los determinantes asociados a mayores ingresos.

La regresión responde a la pregunta *¿Cómo el cambio de una variable afecta el valor de otra variable?*

La regresión es un método muy flexible asociado con las preguntas de asociación y causalidad de nuestros fenómenos por estudiar.

Para esta semana trabajaremos con el caso más simple: **la regresión lineal**. Por lineal hablamos del caso donde nuestra variable objetivo (o dependiente) es *contínua* y la relación entre esta y los atributos (o variables independientes) es mediante una combinación lineal de los últimos.

La regresión permite generalizarse a múltiples casos, por lo que entender su caso canónico nos deja bien preparados para modelos más flexibles como los *Generalized Lineal Models, modelos no paramétricos, multinivel, efectos fijos y Generalized Additive Models*, por nombrar algunos.

Se dice **regresión lineal simple** a aquella que tiene *un regresor como atributo (o variable independiente)*. Más adelante hablaremos de la **regresión lineal múltiple**, aquella que tiene *más de un regresor como atributo (o variable independiente)*.

Nuestro primer modelo de regresión

Para esta semana trabajaremos con una base de datos sobre los ingresos de una muestra, en base a una serie de atributos.

Comencemos por incluir los módulos que utilizaremos en el análisis.

```
%matplotlib inline

import pandas as pd
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import statsmodels.api as sm
import statsmodels.formula.api as smf
import seaborn as sns
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

import lec5_graphs as gfx

plt.style.use('seaborn-whitegrid') # Gráficos estilo seaborn
plt.rcParams["figure.figsize"] = (6, 4) # Tamaño gráficos
plt.rcParams["figure.dpi"] = 300 # resolución gráficos
```

```
/Users/ignaciosotoz/anaconda3/lib/python3.6/site-packages/statsmodels/compat/pandas.py:56:
FutureWarning: The pandas.core.datetools module is deprecated and will be removed in a future
version. Please use the pandas.tseries module instead.
  from pandas.core import datetools
```

De forma adicional a `pandas`, `numpy` y `matplotlib`, agregaremos la librería `statsmodels`. Ésta es una librería orientada a la modelación econométrica siguiendo una sintáxis similar a `R`.

Si no encuentran `statsmodels` en su ambiente de trabajo, pueden agregarlo con:

```
conda install statsmodels
```

La documentación de `statsmodels` es muy completa y afable a los usuarios. Para más detalles, pueden dirigirse a https://www.statsmodels.org/stable/index.html

Dentro de nuestro directorio de trabajo tenemos la base de datos `earnings.csv`, la cual importaremos con pandas

```
df = pd.read_csv('earnings.csv')
df.head()

# output omitido
```

Solicitemos las principales estadísticas descriptivas para cada variable con `describe()`. El método nos entrega información sobre:

- **count**: La cantidad de observaciones en la muestra.
- **mean**: La media aritmética de cada variable.
- **std**: La desviación estandar de cada variable.
- **min**: El valor mínimo observado de la variable en la muestra.
- **25%**: El rango intercuartil inferior de la variable en la muestra.
- **50%**: La mediana de la variable en la muestra.
- **75%**: El rango intercuartil superior de la variable en la muestra.
- **max**: El valor máximo observado de la variable en la muestra.

```
df.describe()

# output omitido
```

`describe()` es útil, pero para las variables categóricas es mejor ver sus frecuencias. Para ello, utilizaremos `.value_counts()` en cada variable.

```
# separemos todas las variables que son categóricas
categorical_columns = ['sex', 'race', 'hisp', 'age_category', 'eth', 'male']

# iniciamos un loop para cada variable categórica donde:
for cat in categorical_columns:
    # imprimimos el nombre
    print("\n",cat)
    # solicitamos la frecuencia relativa de cada categoría dentro de la variable
    print(df[cat].value_counts())
    # solicitamos el porcentaje de la frecuencia relativa
    print((df[cat].value_counts()/len(df[cat])))
```

```
 sex
2    856
1    518
Name: sex, dtype: int64
2    0.622999
1    0.377001
Name: sex, dtype: float64

 race
1    1216
2     126
3      17
```

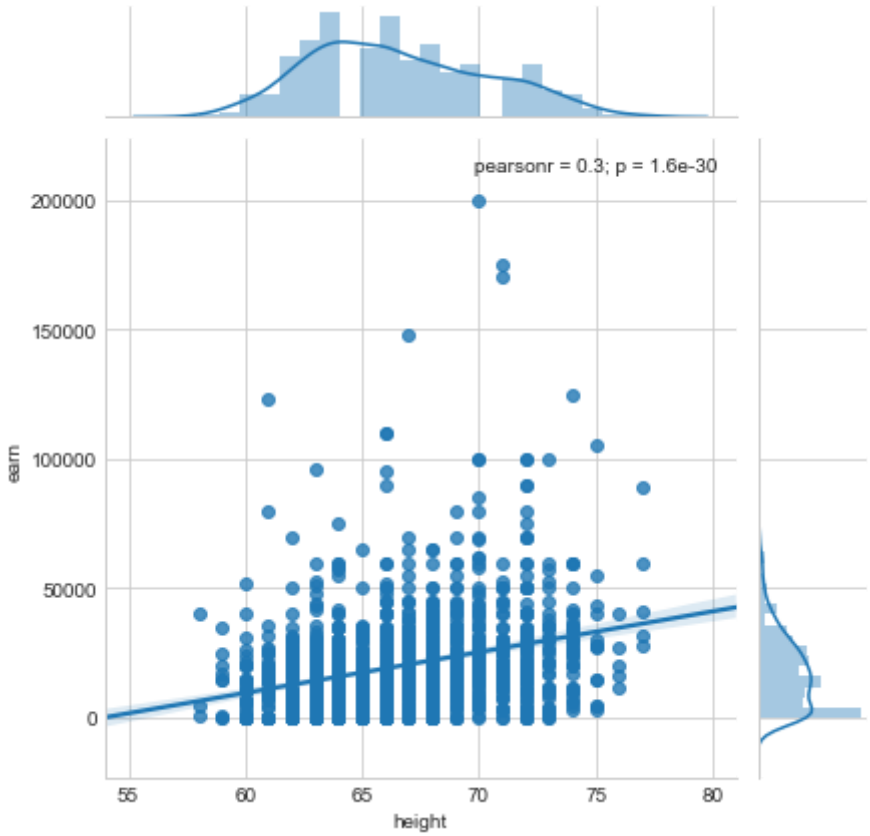
```
4     10
9      5
Name: race, dtype: int64
1    0.885007
2    0.091703
3    0.012373
4    0.007278
9    0.003639
Name: race, dtype: float64

 hisp
2    1294
1      80
Name: hisp, dtype: int64
2    0.941776
1    0.058224
Name: hisp, dtype: float64

 age_category
1    564
2    436
3    374
Name: age_category, dtype: int64
1    0.410480
2    0.317322
3    0.272198
Name: age_category, dtype: float64

 eth
3    1144
1     126
2      77
4      27
Name: eth, dtype: int64
3    0.832606
1    0.091703
2    0.056041
4    0.019651
Name: eth, dtype: float64

 male
0    856
1    518
Name: male, dtype: int64
0    0.622999
1    0.377001
Name: male, dtype: float64
```

Nuestro objetivo es ver la asociación entre los ingresos generados y la altura de cada individuo.

Primero generemos un diagrama de dispersión con `sns.jointplot()`. El gráfico entregará información sobre la distribución de cada unoa de las variables, así como un diagrama de dispersión entre ambas.

La sintáxis de la función es: `sns.jointplot(<var. independiente>, <var. dependiente>)`

```
sns.jointplot(df['height'], df['earn'], kind='reg');
```

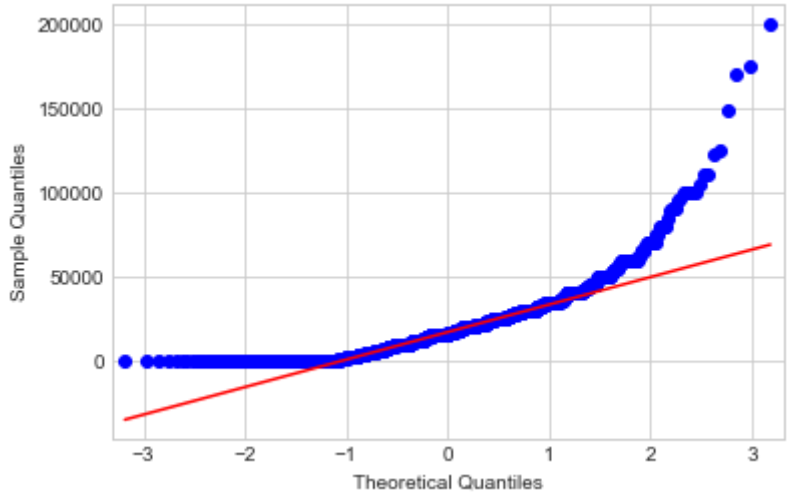


El diagrama ejecutado entrega la siguiente información:

1. El diagrama nos informa de una **asociación positiva entre ambas variables**. Una salvedad a rescatar es que la asociación entre ambas es relativamente débil (su *r de Pearson* es de .30).
2. La baja correlación puede ser explicada por la distribución de las variables. Mientras que `height` persenta una distribución relativamente normal, `earn` está fuertemente sesgada hacia valores bajos.
3. Otro motivo que pueda estar influyendo levemente es que, si nos fijamos en los valores que toma la variable `height` de la tabla entregada por el método describe, los valores al parecer fueron redondeados (min, max y los cuartiles son numeros enteros), esta es la razón también de que se vean 'franjas' en el gráfico de dispersión.

Para examinar la normalidad de nuestras variables, `statsmodels` presenta un gráfico de normalidad Q-Q (*Quantile - Quantile*).

```
sm.qqplot(df['earn'],line='q');
```



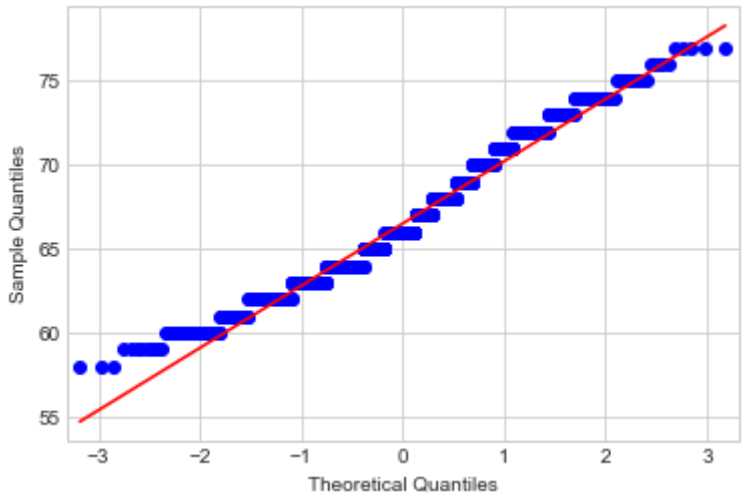
En el gráfico podemos observar que la variable escapa de los valores esperados en los extremos de la distribución. Ésto es un punto a considerar en la modelación que veremos más adelante.

El gráfico Q-Q compara la distribución empírica de nuestra variable en el eje Y (con la etiqueta 'Sample Quantities') contra la distribución *esperada* de la variable (bajo la etiqueta 'Theoretical Quantities').

La opción `line='q'` permite trazar una línea entre cantidades observadas y esperadas en los cuantiles de nuestra variable. Una distribución normal tendrá la mayoría de sus datos cercanos a la línea.

Si ejecutamos el mismo diagnóstico para `height`, el resultado es distinto.

```
sm.qqplot(df['height'], line='q');
```



Inspeccionando contribuciones en nuestra variable dependiente: Regresión Lineal

Una vez realizado las principales exploraciones, nuestro objetivo es generar una explicación plausible de cómo la altura de los individuos afecta los niveles de ingreso en promedio. Para ello utilizaremos la regresión lineal, un método que permite extraer información sobre cómo una variable *independiente* contribuye en una variable *dependiente*.

La regresión lineal es un método que resume cómo los valores de una variable objetivo de característica numérica varían en *subpoblaciones* definidas por una función lineal de atributos. Esta definición proviene de Gelman & Hill (2007) y pone énfasis en la característica descriptiva de la regresión por sobre la idea causal.

Si bien la regresión puede utilizarse para representar asociaciones causales, preferimos enfatizar la capacidad de presentar comparaciones de promedios entre dos subpoblaciones. Así somos explícitos sobre las limitantes de la regresión al no considerar los supuestos de inferencia causal.

Nuestra primera regresión

Nuestro primer modelo a ejecutar tendrá la siguiente especificación:

$$\texttt{earn}_i = \beta_0 + \beta_1 \texttt{height}_i + \varepsilon_i,$$

donde buscamos generar un modelo que explique la varianza de `earn` en función a `height`. En la ecuación agregamos un ε_i para tomar en cuenta la incertidumbre de nuestro modelo. Volveremos a este punto cuando discutamos los supuestos que le dan validez al modelo (Teorema de Gauss-Markov).

El objetivo del algoritmo es generar estimaciones respecto al intercepto (expresado como β_0, que representa el punto de partida de la función lineal) y de la pendiente (expresada como β_1, que representa la contribución de X en Y cuando X cambia en 1 unidad).

Para implementar nuestra primera regresión generamos un objeto a partir de `smf.ols`, el cual genera un modelo de regresión mediante el **Método de Mínimos Cuadrados (Ordinary Least Squares)**. La función de `statsmodels` toma como mínimo dos parámetros:

1. La declaración de la ecuación, que se realiza mediante sintáxis Patsy* con la siguiente forma canónica:

```
'<objetivo> ~ <atributos>'
```

2. un objeto `data`, que en este caso responde a nuestro `pd.DataFrame` generado con pandas.

- La forma canónica permite separar entre nuestro fenómeno a estudiar y los atributos a agregar, separándolos por el operador `~`, el cual define que el objetivo se puede explicar a partir de una función que contiene los atributos.
- Mediante `smf.ols()` pasamos la función y una tabla de datos para generar un objeto con clase `statsmodels.regression.linear_model.OLS`.

```
# generamos un objeto que contenga nuestra ecuación descrita
model1 = smf.ols('earn ~ height', data = df)
```

```
# pidamos la extensión de la clase en nuestro objeto.
model1.__class__
```

```
statsmodels.regression.linear_model.OLS
```

- Hasta el momento sólo hemos generado un objeto con la clase detallada. Para generar las estimaciones de nuestro modelo, debemos ejecutar el método `.fit()`, que instruye al objeto para estimar los parámetros.

```
model1 = model1.fit()
```

- Una vez que el modelo ya está estimado, podemos solicitar sus resultados mediante `.summary()`

```
model1.summary()
```

OLS Regression Results

Dep. Variable:	earn	**R-squared:**	0.092
Model:	OLS	**Adj. R-squared:**	0.091
Method:	Least Squares	**F-statistic:**	138.4
Date:	Sun, 08 Jul 2018	**Prob (F-statistic):**	1.65e-30
Time:	22:17:30	**Log-Likelihood:**	-15475.
No. Observations:	1374	**AIC:**	3.095e+04
Df Residuals:	1372	**BIC:**	3.097e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	-8.487e+04	8926.986	-9.507	0.000	-1.02e+05	-6.74e+04
height	1574.4103	133.829	11.764	0.000	1311.879	1836.942

Omnibus:	868.584	**Durbin-Watson:**	1.903
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	13733.247
Skew:	2.677	**Prob(JB):**	0.00
Kurtosis:	17.533	**Cond. No.**	1.17e+03

Interpretación de un modelo de regresión

`statsmodel` por defecto entrega variada información respecto a la ejecución, estadísticos de bondad de ajuste y los parámetros estimados.

Estadísticos de Bondad de Ajuste: (*R-squared, Adj. R-squared, F-statistic, Prob(F-statistic), No.Observations, etc...*): Informan aspectos generales sobre nuestro modelo tales como su capacidad explicativa.

1. **R-squared y Adj R-squared:** R^2 busca resumir en una cifra *cuál es la capacidad explicativa de nuestros regresores (variables independientes) en la variabilidad de nuestro objetivo (variable dependiente)*. Para nuestro modelo ejecutado, la variabilidad de los ingresos en la muestra se explica en un 9.2% por la altura de los individuos en la muestra. También se entrega un **Adj. R-squared** que entrega un estadístico similar al **R-squared**, pero penalizado por la cantidad de parámetros a estimar. Dado que no hay más parámetros a estimar, las diferencias entre ambos no son substanciales.
2. **F-statistic y Prob(F-statistic):** La prueba F surge de la distribución F, que describe el rango de variabilidad entre partes explicadas y no explicadas. El valor de F se contrasta a un puntaje de corte. En esta prueba se genera una hipótesis nula donde **el modelo no es estadísticamente válido**, y una hipótesis alternativa donde **el modelo es estadísticamente válido**. Estas reglas se simplifican en `Prob(F-statistic)`, donde evaluarmos que si $\texttt{Prob(F-statistic)} \leq 0.05$, tenemos evidencia en contra de la hipótesis nula y hay más evidencia a favor de la significancia estadística general del modelo.

Parámetros Estimados: La parte substancial del modelo se encuentra en su estimación paramétrica. Se estiman $p+1$ parámetros, donde p es la cantidad de atributos a incluir en la fórmula. El $+1$ surge de la estimación del intercepto (β_0), el punto de partida de nuestra recta. Por cada atributo agregado, se genera un β.

Los parámetros se reportan con una serie de estadísticos asociados:

- `coef`: El parámetro estimado en sí.
- `std err`: El error estandar asociado al parámetro.
- `t`: El valor crítico del coeficiente.
- `P>|t|`: El valor p que pone a prueba la significancia estadística del coeficiente.
- `[0.025 0.975]`: El intervalo de confianza al 95% del parámetro.

El **intercepto** informa sobre el punto de partida de nuestra recta. En nuestro primer modelo, nos sugiere que *un individuo en la muestra con 0 pulgadas de altura* tendrá en promedio un salario de 8847 dólares. La estimación de este coeficiente es significativa al 95% (nuestro valor p es menor a 0.05).

El problema de ésta interpretación del intercepto es que no tiene mucho sentido. No existen personas con 0 pulgadas. Para esto, hay que considerar el hecho que nuestro intercepto tiene sentido **sólo en el contexto de nuestro modelo**. Para tener una visión general, debemos considerar la pendiente de nuestra variable independiente.

El coeficiente de **height** informa en cuánto cambia nuestro Y cuando x aumenta en 1 unidad. Con un valor estimado de 1574 estadísticamente significativo al 95%, esperamos que la diferencia entre dos individuos de similares características que difieren sólo en una pulgada, sea de 1574 dólares.

Podemos tentarnos de interpretar este coeficiente desde la causaldad: *en la medida que un individuo crezca una pulgada, esperaremos un incremento en su salario de 1574 dólares*. Esta interpretación presenta una serie de problemas asociados a la causalidad:

- La interpretación es menos flexible, puesto que forzamos un determinado flujo causal. `height` $\rightarrow$ `earn` implica que de forma irrestricta el salario subirá en la medida que la altura aumenta. Esto obliga al investigador a generar contrafactuales sin observaciones empíricas (*¿Qué pasa con un individuo que por X motivo disminuye su altura?*).
- La interpretación causal también asume diseños estadísticos sofisticados, como series de panel, donde se realizan seguimientos a una muestra de individuos de forma temporal.
- La interpretación causal impone una serie de supuestos difíciles de comprobar fuera de la estadística, asociados al diseño.

Validez de las estimaciones

Para entender la validez de nuestras estimaciones mediante Mínimos Cuadrados Ordinarios, es necesario estudiar someramente cómo se generan.

La obtención paramétrica de los β descansa en el método de mínimos cuadrados (MCO).

El objetivo del MCO es **encontrar un estimador que reduzca la distancia residual entre los valores predichos y sus correlatos observados**.

Bajo este enfoque, se asume que los parámetros verdaderos (o poblacionales) son desconocidos y buscamos generar aproximaciones.

Si tenemos un modelo con la siguiente forma:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$

buscamos un estimador que minimice la suma de errores cuadráticos:

$$\begin{aligned} \boldsymbol{\beta} &= \arg\min_{\beta \in \mathbb{R}^d} \mathbb{E}\left[(y_i - X_i^T \beta)^2\right] \\ &= \sum_{i=0}^{N} (Y_i - (\beta_0 + \beta_1 X))^2 \end{aligned}$$

Observación: MCO es un método intuitivo y fácil de entender y programar, sin embargo, está lejos de ser perfecto, una de las más grandes debilidades de MCO es que es extremadamente sensible a valores extremos (outliers), ya sea porque los puntos (x_i, y_i) son muy pequeños o demasiado grandes en comparación con el resto de la muestra, en estos casos el cuadrado de la distancia es una cantidad que puede llevar a desequilibrar por completo la recta de la regresión, los casos en que estos outliers son producto del error humano o de instrumentación pueden ser fatales para el modelo.

En la figura generada con `gfx.gauss_markov()` se presenta tres posibles estimadores candidatos para encontrar nuestro parámetro verdadero ($\boldsymbol{\beta}$). Éstos difieren en sus componentes paramétricos, asumiendo que se comportan de forma asintóticamente normal $\hat{\beta}_j \sim \mathcal{N}(\cdot)$.

- Nuestra primera estimación $\hat{\beta}_1$ tiene una media similar a nuestro parámetro verdadero, pero su varianza es grande.
- Nuestra segunda estimación $\hat{\beta}_2$ tiene una media superior al parámetro verdadero y una varianza similar a la primera estimación.
- Nuestra tercera y última estimación $\hat{\beta}_3$ tiene una media idéntica a la primera estimación, pero una menor varianza.
- En este caso, la estimación $\hat{\beta}_2$ falla en capturar el parámetro poblacional, por lo que es descartada.
- Entre la estimación $\hat{\beta}_1$ y $\hat{\beta}_3$, ésta última satisface las condiciones de Gauss-Markov, dado que su varianza es menor y asegura que error tendrá una media menor.
- El cumplimiento de las condiciones de Gauss Markov nos asegurará que el estimador será el estimador lineal insesgado y óptimo, obteniendo el estimador eficiente dentro de la clase de estimadores lineales insesgados.

```
gfx.gauss_markov()
```

¿Cuáles son las condiciones de Gauss-Markov?

El teorema de Gauss-Markov descansa en cinco supuestos:

1. **La media del error es 0** ($\mathbb{E}[\varepsilon_i] = 0$): No hay sesgo sistemático de forma positiva o negativa respeto a los errores muestrales. Su media será 0 para estos casos. Surge de éste supuesto que cuando X es igual a su media, Y también lo será.
2. **Independencia del error y las variables explicativas** ($\mathbb{E}[\varepsilon_i | X_i] = 0$): Los residuos no están determinados por la variable explicativa.
3. **Ausencia de correlación entre los residuos** ($\mathbb{E}[\varepsilon_i | \varepsilon_j] = 0 (i \neq j)$): Los residuos de las observaciones distitnas deben ser estadísticamente independientes. La información del residuo para uno caso no debe afectar el signo o valor de los residuos para otros casos. Ante los casos de correlación existente entre los residuos, hablamos de *correlación serial* o *correlación espacial*.
4. **Ausencia de correlación entre la varianza y las variables** ($\mathbb{E}[\varepsilon_i^2] = \text{constante}$): La varianza de los errores en cualquier valor de X debe ser independiente a X e Y.
5. **Distribución normal de los errores** ($\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$): Se asume que los errores deben tener una distribución empírica cercana a la normal para todo el rango de X.

Digresión: ¿Qué supuestos son los más relevantes?

Gelman y Hill (2007) sugieren que los supuestos se pueden ordernar de más importantes a menos importantes.

1. *Validez:* Los datos que analizamos deben estar relacionados a la pregunta de investigación que se busca responder. Suena obvio, pero muchas veces se obvía.
2. *Aditividad y Linealidad:* El supuesto más importante desde la matemática es que el componente determinístico (la ecuación) es una función lineal de los predictores.
3. *Independencia de los errores:* Los errores deben estar libres de correlaciones entre sí mismos y no deben afectar a las demás mediciones.
4. *Normalidad de los errores:* Generalmente no es problemático el tener distribuciones no-normales, puesto que se pueden aplicar transformaciones como $log(x)$, $\sqrt{x}$, $1/x$.

> *Gelman, A; Hill, J. 2007. Data Analysis Using Regression and Multilevel/Hierarchcal Models. Cambridge University Press. Ch3*

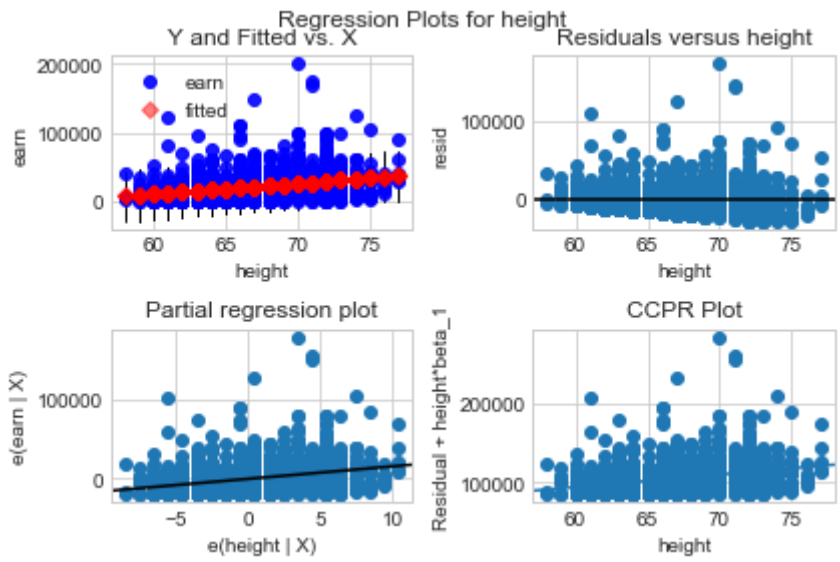
Diagnósticos de nuestro modelo

Para comprobar los supuestos Gauss-Markov mencionados arriba, la inspección más intuitiva es mediante gráficos.

`statsmodels` presenta `plot_regress_exog`, que entrega cuatro gráficos respecto a nuestro modelo de regresión:

1. *Y and Fitted vs. X:* Grafica los valores predichos ($\hat{y}$) y valores empíricos a lo largo de X. Sirve para visualizar en qué valores de X se genera una mejor predicción de los valores empíricos. Observamos que hay valores que se escapan substancialmente de lo predicho cuando la altura es cercana a las 70 pulgadas.
2. *Residuals versus X:* Permite visualizar la heterocedasticidad del modelo al graficar el error residual (ε_i) a lo largo de nuestra variable exógena. Si encontramos una tendencia lineal en la nube de datos. En base al gráfico observamos un patrón de embudo, donde en la medida que aumenta la altura los errores tienden a variar más. Podemos decir que el predictor es heterocedástico en el modelo y necesitamos corregir.
3. *Partial regression plot*: Grafica la evolución de la expectativa de la variable dependiente condicional a los valores de la variable independiente estandarizada.
4. *CCPR Plot (Component and Component Plus Residual)*: Generan un gráfico entre el componente (X) y los residuales de predicción. Permiten diagnosticar espeficicaciones a la variable.

```
sm.graphics.plot_regress_exog(model1, 'height');
```



Variables independientes binarias y sus amigas

Los supuestos de Gauss Markov de la regresión hacen referencia a los errores y estamos limitados por la naturaleza contínua de la variable dependiente a analizar. Podemos flexibilizar nuestro modelo al incluir distintas operacionalizaciones de las variables independientes.

Regresión con una variable binaria

Ahora ejecutaremos una regresión donde nuestra variable independiente toma dos valores: 1 para hombres y 0 para mujeres. Ésta variable se conoce como *binaria* y permite identificar atributos simples en una muestra.

Deseamos ver el efecto que tiene el ser hombre en el salario. Nuestro modelo queda de la siguiente forma:

$$\texttt{earn}_i = \beta_0 + \gamma_1 \times \texttt{male} + \varepsilon_i$$

donde β_0 es nuestro parámetro estimado para el intercepto y γ_1 es el parámetro estimado para la diferencia entre hombres y mujeres en ingreso.

Cabe destacar que éste modelo es el equivalente a una prueba de hipótesis entre 2 muestras independientes.

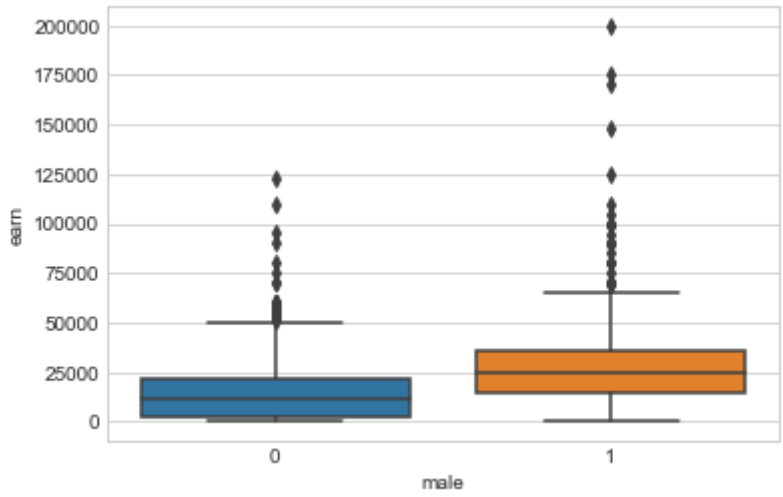
Si solicitamos un gráfico de cajas entre ambas variables, observamos que el rango del salario para los hombres es mucho mayor que el de las mujeres y la mediana se sitúa en salarios más altos.

```
df.loc[df['male'] == 0 ]['earn'].quantile(.75)-df.loc[df['male'] == 0]['earn'].quantile(.25)
```

```
19125.0
```

```
sns.boxplot(x=df['male'], y=df['earn'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c17ed42e8>
```



Digresión: Sobre las variables binarias

En muchas ocasiones nuestro interés es estimar el efecto de un atributo binario (donde 1 indica la presencia de éste y 0 la ausencia) en nuestra variable objetivo. La convención es siempre como 0 aquella característica más común, dado que podemos capturar el comportamiento más común mediante el intercepto.

Si ejecutamos el modelo con una variable binaria de forma `'earn ~ male'`, observamos que el sexo del individuo explica en un 12.4% la variabilidad en el salario de la muestra (esto al ver el R-squared reportado). El intercepto sugiere que para las mujeres el salario promedio es de 14,560 dólares, mientras que los hombres presentan una diferencia de 14,380 dólares más en promedio. Ambos coeficientes son significativos al 99%.

```
model_dummy = smf.ols('earn ~male', data = df).fit()
model_dummy.summary()
```

OLS Regression Results

Dep. Variable:	earn	**R-squared:**	0.124
Model:	OLS	**Adj. R-squared:**	0.124
Method:	Least Squares	**F-statistic:**	194.5
Date:	Sun, 08 Jul 2018	**Prob (F-statistic):**	1.95e-41
Time:	22:17:31	**Log-Likelihood:**	-15450.
No. Observations:	1374	**AIC:**	3.090e+04
Df Residuals:	1372	**BIC:**	3.092e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.456e+04	632.986	23.004	0.000	1.33e+04	1.58e+04
male	1.438e+04	1030.915	13.946	0.000	1.24e+04	1.64e+04

Omnibus:	864.521	**Durbin-Watson:**	1.912
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	13531.216
Skew:	2.664	**Prob(JB):**	0.00
Kurtosis:	17.421	**Cond. No.**	2.43

Regresión con una variable polinomial

Otro aspecto que podemos mejorar cuando incluímos variables es considerar **no linealidades en las variables independientes**. Consideremos el caso donde incluímos la edad del individuo al modelo, que quedaría de la siguiente manera:

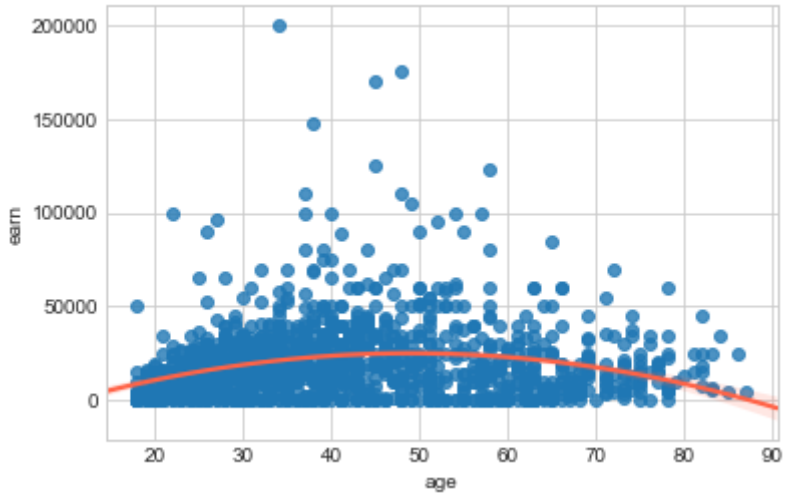
$$\texttt{earn}_i = \beta_0 + \beta_1 \times \texttt{age} + \varepsilon_i$$

Resulta que podemos pensar que los individuos con mayor edad tienden a percibir menores niveles de ingreso, dado que tienen menor poder de negociación y están más cerca de la jubilación. Para ello podemos incluír un término cuadrático para considerar el hecho que el salario puede bajar en la función de la edad. Nuestro modelo quedaría así:

$$\texttt{earn}_i = \beta_0 + \beta_1 \times \texttt{age} + \beta_2 \times \texttt{age}^2 + \varepsilon_i$$

Primero visualicemos la recta con `sns.regplot`:

```
#  generamos un scatterplot entre age y earn
sns.regplot(df['age'], df['earn'],
        # donde definimos que estimaremos una recta con dos grados polinomiales
        order=2,
        # y declaramos el color de la recta para diferenciar.
        line_kws={'color':'tomato'});
```



Se aprecia que la recta indica una parábola negativa: esperamos un peak en el salario percibido cuando los individuos están cerca a los 50 años de edad, declinando después de esa edad.

Ahora generemos el modelo:

```
# generamos una nueva columna que guarde los resultados de elevar al cuadrado la edad
df['age_sq'] = df['age'] ** 2
# iniciamos el modelo incluyendo ambos términos
model3= smf.ols('earn ~ age + age_sq', data=df).fit()
# pedimos los resultados
model3.summary()
```

OLS Regression Results

Dep. Variable:	earn	**R-squared:**	0.057
Model:	OLS	**Adj. R-squared:**	0.055
Method:	Least Squares	**F-statistic:**	41.06
Date:	Sun, 08 Jul 2018	**Prob (F-statistic):**	4.80e-18
Time:	22:17:31	**Log-Likelihood:**	-15501.
No. Observations:	1374	**AIC:**	3.101e+04
Df Residuals:	1371	**BIC:**	3.102e+04
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1.566e+04	3975.144	-3.940	0.000	-2.35e+04	-7863.055
age	1664.1728	184.765	9.007	0.000	1301.719	2026.626
age_sq	-16.9734	1.956	-8.678	0.000	-20.810	-13.137

Omnibus:	843.307	**Durbin-Watson:**	1.956
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	12732.413
Skew:	2.585	**Prob(JB):**	0.00
Kurtosis:	16.988	**Cond. No.**	1.86e+04

El modelo presenta observaciones similares a las del gráfico, mientras que el primer término indica que hay una diferencia de 1.664 dólares entre dos individuos que difieren en 1 año; el segundo término indica una penalización de 16 dólares entre dos individuos que difieren en un año de edad cuando superan la cúspide de ingresos.

Regresión con más de una variable independiente

El modelo de regresión se puede expandir en la cantidad de variables independientes a incluír en la ecuación, dando pie a una *regresión lineal múltiple*. Agregar variables responde a variados objetivos:

- Para mejorar nuestra capacidad descriptiva de un modelo y mejorar nuestro entendimiento de las relaciones presentes entre los datos.
- Para mejorar nuestra capacidad predictiva en la medida que incluímos más información.
- Para considerar de forma explícita problemas causales como las variables intervinientes y controlar por mecanismos alternativas.

Vamos a generar una regresión con la siguiente forma:

$$\texttt{earn}_i = \beta_0 + \beta_1 \times \texttt{ed} + \gamma_2 \times \texttt{male=1} + \varepsilon_i$$

donde modelamos el efecto que tiene la educación y el sexo del individuo en cuánto salario percibe. Para incluír más de un regresor en nuestra sintáxis de `statsmodels`, procedemos de la siguiente manera: `'variabledependiente ~ varindp1 + varindp2'`.

```
model2 = smf.ols('earn ~ ed + male', data=df)
model2 = model2.fit()
model2.summary()
```

OLS Regression Results

Dep. Variable:	earn	**R-squared:**	0.231
Model:	OLS	**Adj. R-squared:**	0.230
Method:	Least Squares	**F-statistic:**	206.4
Date:	Sun, 08 Jul 2018	**Prob (F-statistic):**	4.42e-79
Time:	22:17:31	**Log-Likelihood:**	-15361.
No. Observations:	1374	**AIC:**	3.073e+04
Df Residuals:	1371	**BIC:**	3.074e+04
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2.064e+04	2613.174	-7.898	0.000	-2.58e+04	-1.55e+04
ed	2660.1759	192.327	13.832	0.000	2282.889	3037.462
male	1.352e+04	968.064	13.968	0.000	1.16e+04	1.54e+04

Omnibus:	829.868	**Durbin-Watson:**	1.943
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	12917.525
Skew:	2.515	**Prob(JB):**	0.00
Kurtosis:	17.154	**Cond. No.**	76.2

Para facilitar nuestro entendimiento respecto al modelo, debemos tener las siguientes consideraciones:

- Cada parámetro se interpreta de forma *individual* siguiendo el principio **ceteris paribus**: todas las demás variables consideradas en el modelo pero no interpretadas se *asumen* que se mantienen constantes en la media.

- La predicción de valores en nuestra variable dependiente se asume como *la suma de todos los coeficientes estimados del modelo*. Esto se conoce como la propiedad aditiva de la regresión lineal.

En este caso nuestra regresión considera una variable contínua y una variable binaria. Para este caso es útil separar nuestra ecuación detallada en dos posibles estimaciones:

- Una **Ecuación para Hombres** donde se considera el parámetro estimado `male`:

$$\texttt{earn}_i = \beta_0 + \beta_1 \times \texttt{ed} + \gamma_2 \times \texttt{male=1} + \varepsilon_i$$

- Una **Ecuación para Mujeres** donde la ausencia de atributo `male` implica que el parámetro estimado no se incluye en esa estimación:

$$\texttt{earn}_i = \beta_0 + \beta_1 \times \texttt{ed} + \gamma_2 \times \texttt{male=0} + \varepsilon_i \Rightarrow \texttt{earn}_i = \beta_0 + \beta_1 \times \texttt{ed} + \varepsilon_i$$

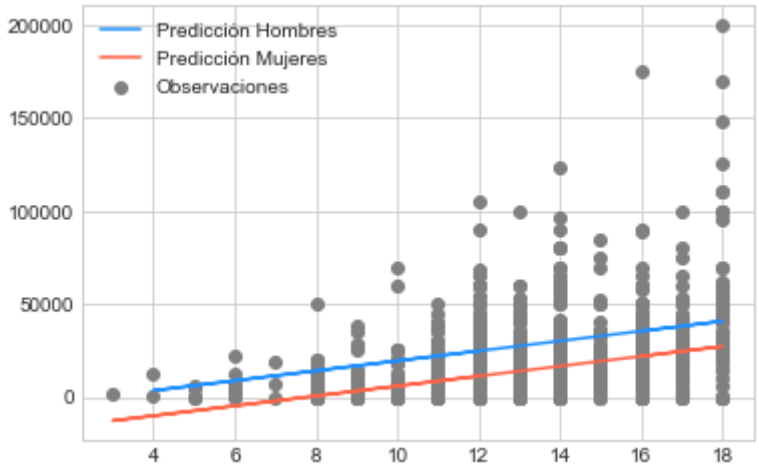
Siguiendo nuestro modelo, se aprecia que si bien la diferencia en los salarios entre dos personas con similares características pero que difieren en un año de educación es de 2660 dólares. De manera similar a nuestro modelo binario, la diferencia entre hombres y mujeres en los salarios es de 13520 dólares en promedio.

Para entender de una manera más clara el impacto de `male`, generaremos un gráfico de líneas paralelas.

```
# una buena práctica es generar copias de nuestro objeto para evitar modificación.
df_dummy = df.copy()

model_3 = smf.ols('earn ~ ed + male', df).fit()
# ahora guardemos los valores predichos de nuestro modelo en nuestra base.

df_dummy['yhat'] = model_3.predict()
df_dummy.head()

# output omitido
```

```
# comencemos por graficar todos los puntos en la relación
plt.scatter(df_dummy['ed'], df_dummy['earn'], color='grey',
            label = 'Observaciones')
# grafiquemos la proyección para hombres
plt.plot(df_dummy.query('male == 1').ed,
         df_dummy.query('male == 1').yhat,
         color = 'dodgerblue', label = 'Predicción Hombres')
# grafiquemos la proyección para mujeres
plt.plot(df_dummy.query('male == 0').ed,
         df_dummy.query('male == 0').yhat,
         color = 'tomato', label  = 'Predicción Mujeres')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1c1816dc50>
```

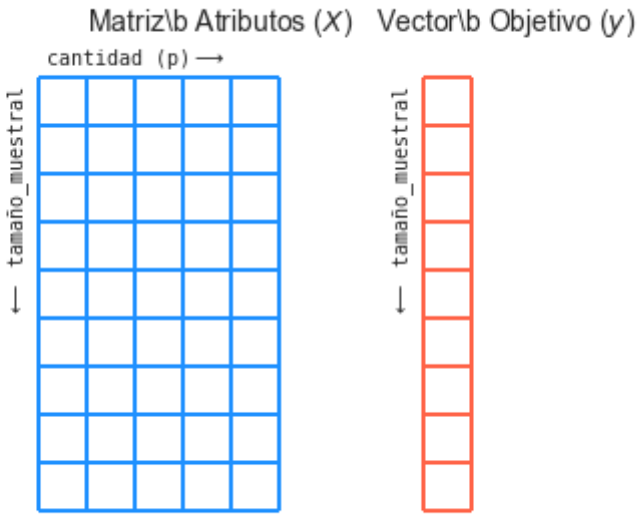


El efecto estimado de ser hombre en el salario se mantiene de forma **constante** en la medida que cambiamos el valor del educación.

Machine Learning

Una de las principales diferencias entre la estadística, machine learning e inteligencia artificial es cómo se refieren a las variables, Mientras que en la estadística (y la tradición más académica) se habla de variables *dependientes* (el fenómeno a estudiar) e *independientes* (atributos que pueden explicar la variable dependiente), en machine learning se habla de *atributos* y *objetivos*.

```
gfx.feature_target()
```



En machine learning se suele considerar la matriz de atributos (X) y el vector objetivo (y) como entidades separadas. El por qué hace referencia a la finalidad de generar buenas predicciones a partir de la matriz de atributos, no el discriminar entre distintas variables pertenencientes a X.

El flujo de trabajo desde Machine Learning

Primer elemento: Conocer los datos

Los datos son observaciones de fenómenos empíricos. Cada uno de ellos provee información (limitada) sobre un aspecto de nuestro fenómeno a estudiar. La colección de todos los datos nos permite dar una panorámica (aún incompleta) sobre el fenómeno.

Es sano asumir que todos los datos vienen con información faltante y ruido de medición.

Segundo elemento: Determinar los objetivos de Trabajo

El tipo de datos a disposición limita el tipo de objetivos de trabajo.

Más que un fin en sí, los objetivos permiten fijar las preguntas de investigación.

Los objetivos de trabajo también permiten definir la arquitectura y modelos que se utilizarán.

Tercer elemento: Diseñar e implementar los Modelos

La elección del modelo viene determinada parcialmente por los objetivos de trabajo y los datos disponibles.

Algunos elementos a considerar:

- ¿Qué esperamos como resultado?
- ¿Qué parámetros estimaremos?
- ¿Qué hiperparámetros consideraremos?

Los *parámetros* son aquellos elementos que se generan mediante los algoritmos o métodos de estimación. Éstos no son gobernados directamente por el investigador.

Los *hiperparámetros* son los elementos que se determinan por el investigador e influyen en los parámetros a estimar. Ejemplos de hiperparámetros son el porcentaje de datos que se distribuyen en los subconjuntos *train* y *test*, la inclusión de un parámetro basal como β_0, la normalización de la matriz de datos, entre otros.

Nuestro primer modelo desde Machine Learning con `scikit-learn`

Para nuestro primer modelo utilizaremos la misma base de datos `earnings.csv`, donde buscaremos predecir los ingresos de individuos a partir de una serie de características registradas.

Utilizaremos `scikit-learn` [1] , una librería diseñada para implementar modelos de minería de datos y machine learning de manera simple y efectiva. Opera con componentes de `numpy`, `scipy` y `matplotlib` para facilitar la labor del investigador.

En énfasis en la facilidad de uso, rigurosidad en el código, amplia comunidad de apoyo y buena documentación la han transformado en la principal librería de análisis enfocado a la predicción. Para más información sobre `scikit-learn`, vea la página oficial https://http://scikit-learn.org/.

Paso 1: Importación de módulos

Dado la vasta cantidad de funciones en la librería, se sugiere importar los módulos específicos que utilizaremos. Esto conlleva un trabajo de identificar todos los argumentos requeridos en las funciones.

Para generar un modelo lineal necesitamos el módulo `linear_model`, que importaremos mediante `from <...> import <...>`.

Posteriormente importamos las métricas de evaluación alojadas en `sklearn.metrics`.

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
```

Paso 2: Dividiendo la muestra

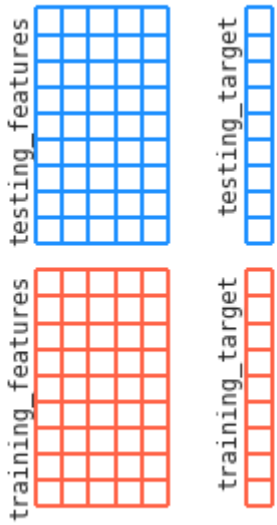
Otra de las características del machine learning es la divisón de las muestras entre dos grupos:

1. **Training Set**: Donde implementamos el modelo.
2. **Test Set**: Donde probamos el modelo.

El siguiente paso es poder dividir la muestra en cuatro partes. La división se puede ejemplificar en la figutra `gfx.train_testing()`. En ella generamos una división de nuestra matriz ingresada al ambiente de trabajo, teniendo en cuenta los siguientes aspectos:

1. La división debe realizarse entre una muestra que utilizaremos para generar una representación fidedigna del fenómeno.
2. Una muestra donde vamos a poner a prueba nuestra estimaciones y contrastaremos lo predecido con las observaciones reales.

```
gfx.train_testing()
```



Necesitamos generar las muestras mediante el operador slice de pandas. Los cuatro grupos que debemos crear son:

1. Matriz de Atributos en Training Set: `X_mat_train`
2. Vector Objetivo en Training Set: `y_train`
3. Matriz de Atributos en Testing Set: `X_mat_test`
4. Vector Objetivo en Testing Set: `y_vec`

Para generar nuestra matriz de atributos, es necesario excluir el vector objetivo.

```
# Generemos un nuevo objeto llamado attr_mat que excluya Unnamed: 0 y earn
attr_mat = df.drop(['Unnamed: 0', 'earn'], axis =1)
```

Ahora generamos la división entre muestra de entrenamiento del modelo y muestra de prueba del modelo. Por ahora nuestra aproximación es excluir las últimas 30 filas en nuestra muestra de entrenamiento, y agregarlas a nuestra muestra de prueba. Repetimos exactamente el mismo proceso para el vector objetivo.

```
# dividir matriz de atributos
X_mat_train = attr_mat[:-30]
X_mat_test = attr_mat[-30:]

# dividir vector objetivo
y_train = df['earn'][:-30]
y_test = df['earn'][-30:]
```

El método que acabamos de utilizar para separar nuestros conjuntos de entrenamiento y prueba se le conoce comúnmente como '*máscara estática*', la razón de esto es porque podemos ejecutar el bloque de código anterior las veces que queramos y siempre obtendremos los mismos valores en cada conjunto, en nuestro caso el conjunto de entrenamiento se compone de todas las filas excepto las últimas 30, mientras que el de pruebas se compone solo de esas 30 últimas.

En rigor, no deberíamos utilizar una máscara estática para seleccionar nuestro conjunto de entrenamiento/pruebas, deberíamos seleccionar los registros que estarán en cada conjunto de forma aleatoria, la razón de esto es porque los datos pueden estar ordenados bajo algún criterio que desconocemos (ya sea por la forma en la que se ingresaron al momento de registrarlos o, por ejemplo, producto de algún mecanismo de índice en la base de datos en la que fueron almacenados) y al entrenar nuestro modelo lo estaremos sesgando hacia esos valores. Más adelante en este mismo módulo introduciremos un método que nos provee la librería `scikit-learn` para obtener esta partición aleatoria. En esta ocasión utilizaremos la máscara estática para hacer el ejemplo menos aparatoso.

Paso 3: Generar objeto y modelo

El siguiente paso es la construcción de un objeto que contenga las instrucciones a ejecutar en el modelo. Para el caso de la regresión lineal, esto se genera con la clase `LinearRegression()` del módulo `linear_model`.

A diferencia de `statsmodels`, donde definíamos la fórmula y los datos para crear el objeto, en `scikit-learn` se defininen los hiperparámetros definidos por el usuario. Por hiperparámetros entendemos todas las instrucciones hacia el modelo que permiten definir la tolerancia, método de optimización o criterios de estimación. Esto a diferencia de los parámetros estimables que dependen exclusivamente del método de estimación.

Lo más comunes son:

- `fit_intercept`: Incluimos la estimación del intercepto (β_0) en el modelo.
- `normalize`: Reescalar cada uno de los atributos en la muestra de prueba mediante la norma euclídea.

```
modelo = linear_model.LinearRegression(fit_intercept=True, normalize=False)
```

Al generar el objeto con el método `LinearRegression()`, podemos inicializar la clase de similar manera a como lo hicimos con `statsmodels`. Para ello, ingresamos el training set `X_mat_train` e `y_train` en el método `.fit()` del objeto `modelo`.

Cuando pasamos los datos por la función `.fit()` no es necesario asignar de nuevo el objeto, dado que estamos generando una instancia específica de éste.

```
modelo.fit(X_mat_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Ahora `modelo` contiene información sobre los parámetros estimados como el intercepto y las pendientes para cada atributo en la matriz.

```
# imprimamos la lista de coeficientes
print("La lista de pendientes: :", modelo.coef_)
# imprimamos el intercepto del modelo
print("El intercepto del modelo es :", modelo.intercept_)
```

```
La lista de pendientes: : [   14.92703623   132.30589935 -6360.94719277  -661.37164603
  1688.11542562  2428.20840913  -886.76446618   311.43033413
   886.76446618 -3038.69811608   185.21329516  6360.94719277
   -15.73447966]
El intercepto del modelo es : 13833.688816521604
```

Cuando solicitamos las pendientes estimadas con `modelo.coef_`, Python nos devuelve un array con cada una de ellas. Con `modelo.intercept_` obtenemos información sobre el punto de partida en las estimaciones cuando **todas nuestras variables están situadas en 0**.

Paso 3.1: Generar predicciones

Nuestro objetivo con Machine Learning es el poder utilizar nuestras explicaciones sobre un fenómeno en datos donde no tenemos la información completa, o en datos donde no tenemos la información completa y deseamos predecir cuál sería su comportamiento, asumiendo que nuestro modelo está correcto.

El primer paso para medir el desempeño predictivo del modelo es generar nuevas predicciones en nuestra muestra de prueba. Para ello nos valemos de la función `.predict()` del modelo.

La función `.predict()` del modelo resulta crucial de entender. La función necesita como argumento una matriz para generar predicciones. Acá es donde ingresamos la matriz de atributos del testing set.

```
earn_yhat = modelo.predict(X_mat_test)
print("La cantidad de predicciones realizadas en X_mat_test son: ", len(earn_yhat))
print("las predicciones son ", earn_yhat)
```

```
La cantidad de predicciones realizadas en X_mat_test son:  30
las predicciones son  [25598.08771501 22456.39927869 25859.43110455  6927.53079027
 19541.84095374 23007.85181165 15953.0359177  20722.82694069
 30815.51755312 42107.27749179 17374.42556013 27321.46818842
 16039.6979205  42630.94412833 18053.08347602 25336.99601278
 21507.10037955  6230.39947764 20825.36789164 -4891.86653372
 20939.55454343 31366.3369532  41244.16654204 44883.74058834
 24000.40337208 24107.37666572 18000.07128118 12600.30094273
 29170.58626159 22120.37934468]
```

Las predicciones de `X_mat_test` realizadas con nuestra función las guardamos en un nuevo objeto llamado `earn_yhat` (*yhat* hace referencia a la estimación de $\hat{y}$). Si solicitamos las nuevas observaciones, nos devolverá un array con cada uno de los puntos estimados en base a los datos de `X_mat_test`.

Paso 4: Generar métricas

Finalmente, es necesario juzgar qué tan bien se comportó nuestro modelo al predecir los valores del vector objetivo de nuestro testing set. `sklearn` ofrece una serie de funciones para comparar las diferencias entre el vector objetivo de la prueba y las estimaciones del vector objetivo en base a nuestro modelo.

Utilizaremos medidas que buscan resumir qué tan grande es la diferencia entre la predicción y los datos empíricos, y el R^2, una medida convencional proveniente de la econometría:

- **Promedio del Error Cuadrático** (Mean Squared Error): Representa la expectativa del error cuadrático. Es un indicador de calidad con valores no negativos, donde menores valores indican mejores niveles de ajuste. Una medida similar que se presenta en algunos trabajos es el *Root Mean Squared Error* que representa la raíz cuadrada del MSE.
- **R-cuadrado**: Representa la capacidad explicativa de nuestro conjunto de atributos en la variabilidad de nuestro vector objetivo. Tiene la misma interpretación que la enseñada desde la econometría.

Guardaremos los resultados en objetos nuevos (`m1_mse` y `m1_r2`) para su posterior uso.

```
m1_mse = mean_squared_error(y_test, earn_yhat).round(1)
m1_r2 = r2_score(y_test, earn_yhat).round(2)
print("Mean Squared Error: ", m1_mse)
print("R-cuadrado: ", m1_r2)
```

```
Mean Squared Error:  393102326.6
R-cuadrado:  0.31
```

La interpretación del R cuadrado es relativamente simple y similar a la econometría, donde esperamos que nuestra función explique en un 31% la variabilidad de nuestra variable objetivo.

El problema con el Promedio del Error Cuadrático es que necesitamos compararlo con otro modelo para ver cuál reduce más el error cuadrático. Este es un punto que volveremos a visitar al final de la lectura.

¿Qué fue lo que hicimos?

La regresión es un problema de ***aprendizaje supervisado***, dado que buscamos solucionar problemas donde existen *inputs* y *outputs* que son manejados *a priori* por el investigador. El vector objetivo en los problemas de regresión es un vector *numérico*. Se dice que es un problema supervisado cuando el investigador tiene suficiente información como para calibrar el modelo (en detrimento de los modelos no supervisados donde gran parte de la responsabilidad recae en el algoritmo y en el computador). Visto de otra forma, antes de entrenar el modelo ya se conocen los valores de la variable objetivo que el modelo tiene que ser capaz de predecir (en nuestro ejemplo desde un comienzo conocíamos los valores de los salarios que queriamos que el modelo predijiese), en el aprendizaje no supervisado no tenemos la posibilidad de comparar lo que predice el modelo con la realidad y esperamos que el modelo sea capaz de encontrar estos valores/clases (por ejemplo, en problemas de clustering).

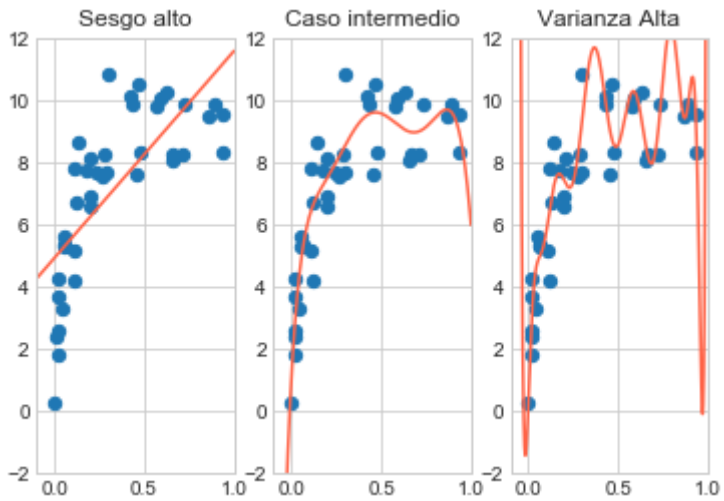
En el problema de regresión buscamos controlar la **complejidad** del modelo, que viene dada por la cantidad de *inputs* que asignamos a nuestra función. De esta manera, los contratiempos asociados a éste problema son el **exceso de ajuste** (presentado cuando tenemos muchos inputs que impiden la generalización predictiva) también llamado *overfitting*, y la **falta de ajuste** (presentada cuando los inputs ingresados a la función no son suficientes como para caracterizar las predicciones *dentro de la muestra*) también llamado *underfitting*.

Dividir los datos en conjuntos de prueba y muestra, y el comparar las predicciones responde a la necesidad de probar la *capacidad de generalización* del modelo.

Trueque entre sesgo y varianza

Uno de los principales criterios para analizar la capacidad de generalización del modelo es ver cómo se comportan el sesgo, la varianza y la complejidad del modelo implementado. En la figura generada con `gfx.bias_variance()`, se grafica la recta de ajuste cuando establecemos una regresión con polinomios de 1, 7 y 15 grados respectivamente. Esto genera tres situaciones:

```
gfx.bias_variance()
```



En la figura de la izquierda encontrarmos la situación donde se presenta un sesgo alto. Por **Sesgo** entendemos la tendencia de un estimador de elegir un modelo incorrecto dado que utiliza supuestos incorrectos sobre el proceso generativo de los datos. Un ejemplo es si utilizamos un modelo lineal en una función cúbica. El modelo **subrrepresentará** los valores verdaderos en la muestra, independiente del tamaño de ésta.

En la figura de la derecha encontramos un estimador muy flexible, caracterizado por una varianza alta. Por **Varianza** se entiende la sensibilidad del estimador a fluctuaciones en el training set. Altos niveles de varianza en la muestra pueden generar mucho ruido en el estimador, generando predicciones fuera del rango empírico de los datos. En este caso el modelo **sobrerrepresentará** los valores verdaderos de la muestra.

El objetivo del aprendizaje supervisado es especificar los parámetros de forma tal de encontrar un modelo que sea lo suficientemente flexible con el caso estudiado, pero que también pueda ser extrapolado a nuevos datos y tener un buen desempeño.

Métricas de ajuste de modelo

Para juzgar cuándo un modelo de machine learning permite generalizar predicciones, nos valemos de *métricas* y *pruebas* para su validación.

Si desde la econometría nos valemos de medidas de variabilidad explicada como el R^2, desde machine learning nos guíamos por el principio rector de la *minimización de pérdidas*.

La mayoría de las métricas de un modelo de regresión buscan estimar el error promedio. Una de las más utilizadas es la *mean-square error* (MSE), que se define de forma:

$$\mathrm{MSE}(\hat{f}, \mathrm{datos}) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{f}\left(\mathbf{x_i}\right) \right)^2$$

De forma alternativa también está el *root-mean-square error* (RMSE), que es la raíz de nuestra métrica MSE. Las diferencias entre ambas son arbitrarias, puesto que RMSE reescala MSE.

Train-Test Split

Un problema de estimar un modelo con la muestra completa es que el cálculo del MSE depende de la cantidad de atributos ingresados en el modelo. De esta manera, el error tenderá a bajar o en el peor de los casos a mantenerse invariante.

Si desde la **econometría** el problema era estimar la relación entre Y e X mediante la función lineal $Y = f(x) + \varepsilon$, para **machine learning** el problema es estimar la función $f(\cdot)$ con un estimado de nuestra función $\hat{f}$.

Otro tema con la validación del modelo y su capacidad de generalizar predicciones, es la necesidad de ver cómo se comportan las estimaciones con nuevos datos. Este es una característica crucial de los modelos de machine learning más complejos: **realizar predicciones en tiempo real de nuevos datos ingresados**.

La limitación de esto es que rara vez tenemos acceso a más datos sobre nuestro fenómeno. Las buenas prácticas indican que para tener una aproximación sobre la generalización, podemos dividir la muestra en dos submuestras:

Para estimar nuestra función candidata $\hat{f}$, debemos seguir lo siguiente:

1. Recolectamos una serie de datos que tengan $p \geq 1$ atributos (o variables independientes) y un vector objetivo y. Esta serie de datos conocida la denominamos **Training Set** y provee un campo empírico para implementar nuestros modelos. En la figura, éste campo se representa en los cuadrantes **X** e **y**.
2. Con los datos a mano, elegimos un método (o múltiples métodos) para generar una estimación $\hat{f}$.
3. Utilizamos nuestro training set para *entrenar* o *ajustar* nuestra función predictiva $\hat{f}$. Por entrenar se entiende que el algoritmo/método capture **información existente** sobre el fenómeno.
4. Con nuestra función entrenada con los datos, buscamos generar predicciones $\hat{y} = \hat{f}(x)$ en un conjunto de datos que no fueron considerados en el conjunto de entrenamiento. En la figura, esto se representa con $\tilde{X}$. En este último conjunto de datos se conoce como **Testing Set**. Es necesario que éste contenga los mismos atributos que en el training set:
    - Si los datos del testing set son una submuestra proveniente de la misma fuente que el training set, se consideran como datos de validación *dentro de la muestra*. Éstos datos permiten comparar las predicciones de nuestro método en contraste al vector objetivo.
    - Si los datos del testing set son nuevos datos, se considera como datos de validación *fuera de la muestra* y permite medir el desempeño de nuestras estimaciones.

Digresión: No free lunch theorem

Del punto 2 se desprende una máxima a considerar: **no existe un método o algoritmo que provea de una solución óptima fuera de la empíricamente probada**. Por ello es que siempre debemos establecer múltiples modelos para obtener un criterio informado. Éstos métodos no implican nuevos algoritmos de forma obligatoria, pero probar con distintas espeficiaciones de los parámetros e hiperparámetros. Se puede encontrar más información y detalles formales de éste teorema en http://www.no-free-lunch.org/.

Con nuestros **train** y **test** definidos, podemos medir la capacidad de reducir el error mediante *MSE*. Así calcularemos un MSE_{train} y un MSE_{test}.

$$\text{MSE}_{Train} = \text{MSE}(\hat{f}, Train) = \frac{1}{n_{Tr}} \sum_{i \in Train} \left(y_i - \hat{f}(\mathbf{x_i}) \right)^2$$

donde n_{Tr} es la cantidad de observaciones en el subconjunto $train$. MSE_{train} siempre tenderá a bajar (o a estancarse en un valor) en la medida que la complejidad aumenta. Esto implica que no será tan útil para comparar modelos.

$$\text{MSE}_{Test} = \text{MSE}(\hat{f}, Test) = \frac{1}{n_{Te}} \sum_{i \in Test} \left(y_i - \hat{f}(\mathbf{x_i}) \right)^2$$

En este caso n_{Te} es el número de observaciones en el subconjunto $test$. MSE_{test} servirá para medir *¿Qué tan bien predice nuestro modelo en datos previamiente desconocidos?*.

Algunos comportamientos de la complejidad y el tamaño de los modelos

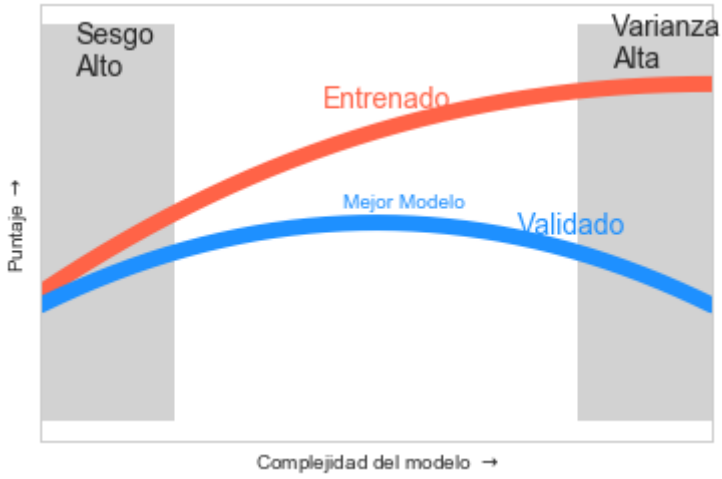
En la figura generada con `gfx.validate_curve()` se presenta la visualización entre un puntaje (llámese bondad de ajuste, verosimilitud, etc..) y la complejidad del modelo. Mediante este gráfico buscamos evaluar cómo se comporta nuestra métrica de evaluación respecto a la complejidad del modelo. Por complejidad del modelo se entiende de la cantidad de parámetros que se agregan al modelo para estimar.

Encontramos tres escenarios:

1. Un escenario caracterizado por una baja complejidad del modelo, que resulta en sesgos altos. Esto implica que el modelo falla en capturar de forma clara los parámetros verdaderos y la caracterización es insuficiente dado la baja cantidad de parámetros.
2. Otro escenario por una excesiva complejidad del modelo, lo que resulta en alta varianza del modelo. Bajo esta situación, el modelo genera una muy buena representación de los datos de la muestra entrenada, pero falla en replicar la generalización de las predicciones en la muestra de testing.
3. Otro escenario donde encontramos la mejor estimación del puntaje para la muestra de validación. Bajo este escenario la complejidad del modelo es óptima. Este es el objetivo del aprendizaje supervisado

Cabe destacar que mientras que en la muestra de entrenamiento el puntaje siempre incrementará en la medida que agregemos más atributos, esto será ineficiente en la medida que generaremos una función no parsimoniosa que no podremos aplicar a otro conjunto de datos.

```
gfx.validate_curve()
```



Ya entendiendo que la cantidad de atributos agregados puede dañar la capacidad de generalizar un modelo en nuevos datos, es meritorio mencionar el efecto del tamaño muestral en el aprendizaje del modelo.

Por aprendizaje del modelo hacemos referencia a la capacidad de obtener información sobre los parámetros verdaderos a estimar.

La figura generada con `gfx.learning_curve()` muestra la relación entre el tamaño de las muestras de entrenamiento y prueba, y el puntaje a optimizar. Se aprecia que en situaciones donde el tamaño muestral del modelo entrenado es bajo, los puntajes serán sobrestimados en relación a los puntajes de la muestra validada.

Esperamos que ambos modelos tiendan a converger en la medida que el tamaño muestral aumenta.

Esto está asociado a lo que vimos en sesiones pasadas con los aspectos asintóticos de las variables aleatorias, donde una mayor cantidad de observaciones de una muestra permite asimilar de mejor manera las características poblacionales.

```
gfx.learning_curve()
```

Existen métodos más sofisticados para generar subconjuntos como la validación cruzada (*crossvalidation*) y la estimación 'deja-uno-afuera' (*Leave-one-out*) que veremos la próxima semana.

Refactorización de nuestro modelo

Feature engineering

Resulta que el modelo que generamos con `sklearn` incluye todas las variables dentro de la matriz de datos. Un inconveniente relacionado es el hecho que nuestro modelo puede que no sea el óptimo, dado que incluimos regresores que no aportan poder predictivo y utilizan grados de libertad de forma ineficiente. Nuestro objetivo es encontrar un número adecuado de regresores a incluír en nuestro modelo predictivo.

Una manera simple de determinar cuáles van a ser los principales atributos a incluir en el modelo es mediante la generación de correlaciones parciales entre cada atributo y el vector objetivo.

Este procedimiento es un ejemplo de *feature engineering*, el seleccionar y/o preprocesar atributos correspondiente a los datos que permiten resumir de manera precisa un modelo. Está asociado al concepto de *parsimonía* proveniente de la academia, donde la explicación más adecuada es aquella que no agregue información irrelevante (el principio de la navaja de Occam).

El código de abajo es una versión extremadamente simplificada de una búsqueda de correlaciones. Para tener un algoritmo más depurado, es necesario diferenciar entre la naturaleza de las variables ingresadas y el tipo de correlación que se genera. Para ello se pueden ocupar correlaciones poliseriales para la asociación categórica-contínua y punto biserial para ordinales-contínuas.

```
# extraemos los nombres de las columnas en la base de datos
columns = df.columns

# generamos 3 arrays vacíos para guardar los valores
# nombre de la variable
attr_name = []
# correlación de pearson
pearson_r = []
# valor absoluto de la correlación
abs_pearson_r = []

# para cada columna en el array de columnas
for col in columns:
    # si la columna no es la dependiente
    if col != "earn":
        # adjuntar el nombre de la variable en attr_name
        attr_name.append(col)
        # adjuntar la correlación de pearson
        pearson_r.append(df[col].corr(df['earn']))
        # adjuntar el absoluto de la correlación de pearson
        abs_pearson_r.append(abs(df[col].corr(df['earn'])))

# transformamos los arrays en un DataFrame
features = pd.DataFrame({
    'attribute': attr_name,
    'corr':pearson_r,
    'abs_corr':abs_pearson_r
})
```

```
# generamos el index con los nombres de las variables
features = features.set_index('attribute')
# ordenamos los valores de forma descendiente
features.sort_values(by=['abs_corr'], ascending=False)
```

attribute	abs_corr	corr
sex	0.352354	-0.352354
male	0.352354	0.352354
ed	0.349334	0.349334
height	0.302707	0.302707
height1	0.188762	0.188762
height2	0.105633	0.105633
age_category	0.075321	0.075321
yearbn	0.068475	-0.068475
age	0.068475	0.068475
hisp	0.055187	0.055187
eth	0.055139	0.055139
race	0.031410	-0.031410
age_sq	0.026209	0.026209
Unnamed: 0	0.024675	-0.024675

Implementando un modelo con menos regresores

En base a la información obtenida con nuestra tabla de correlaciones parciales, podemos refinar nuestro primer modelo para reducir el error cuadrático.

Para ello seleccionaremos aquellas correlaciones que sean superior al .30.

Resulta que nuestra versión más parsimoniosa incluye sólo a Male, Ed y Height.

```
# separemos los vectores a trabajar
y_vec= df.loc[:, 'earn']
X_mat = df.loc[:, ['male', 'ed', 'height']]
```

Ahora procedemos a dividir la muestra en conjuntos de entrenamiento y prueba. A diferencia del ejemplo anterior, donde nuestra división se realizó mediante la separación de las últimas 30 observaciones, en este caso utilizaremos `train_test_split`, un método que permite realizar separaciones de muestras aleatorizadas. Cabe destacar que éste método es el preferido, dado que mediante la aleatorización podemos cancelar el efecto de sesgo por variables intervinientes en la muestra.

El método toma como argumentos la matriz de atributos (`X_mat`) y el vector objetivo a dividir (`y_vec`), el porcentaje de datos que serán asignados al conjunto de muestra (`test_size`), y una semilla pseudoaleatoria para asegurar replicación de resultados (`random_state`).

`train_test_split` devuelve como objetos la matriz de atributos para entrenar (`X_train`), la matriz de atributos para validar (`X_test`), el vector objetivo para entrenar (`y_train`) y el vector objetivo para contrastar las predicciones (`y_test`). Cabe destacar que el método las devuelve en ese orden, así que es mejor seguirlo para evitar confusiones entre objetos.

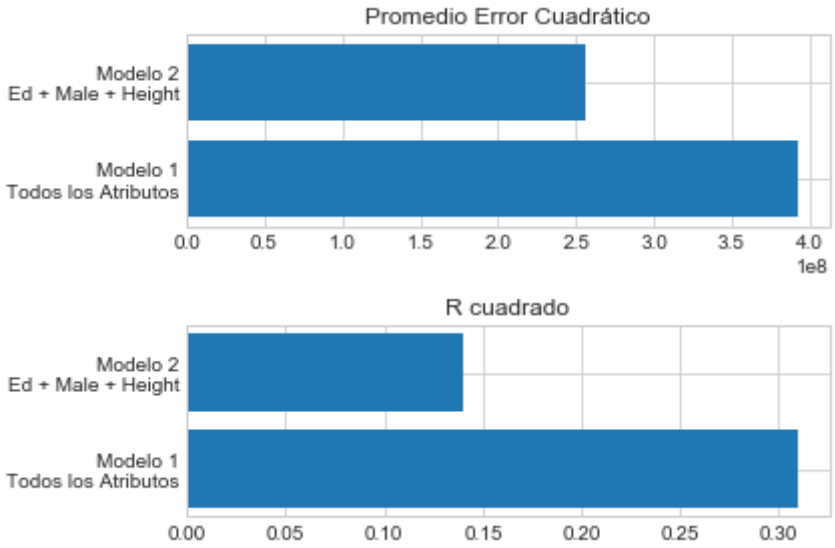
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_mat, y_vec, test_size=.30, random_state=11238)
```

Con la muestra ya dividida, podemos implementar el modelo de regresión con los mismos hiperparámetros anteriores.

```
purge_model = linear_model.LinearRegression(fit_intercept=True, normalize=True)
purge_model.fit(X_train, y_train)
purge_model_yhat = purge_model.predict(X_test)
```

Ahora podemos comparar el poder predictivo entre los dos modelos mediante el promedio del error cuadrático y el r cuadrado. Por motivos prácticos, se sugiere presentar toda esta información de manera visual.

```
m2_mse = mean_squared_error(y_test, purge_model_yhat).round(0)
m2_r2 = r2_score(y_test, purge_model_yhat).round(2)
plt.subplot(2,1,1)
plt.barh(['Modelo 1\nTodos los Atributos', 'Modelo 2\nEd + Male + Height'], [m1_mse, m2_mse])
plt.title("Promedio Error Cuadrático")
plt.subplot(2,1, 2)
plt.barh(['Modelo 1\nTodos los Atributos', 'Modelo 2\nEd + Male + Height'], [m1_r2, m2_r2])
plt.title("R cuadrado");
plt.tight_layout()
```



Si comparamos el promedio del error cuadrático para ambos modelos, existe evidencia para preferir el segundo modelo, dado que presenta una reducción significativa del error cuadrático.

Resulta contraintuitivo el hecho que el R cuadrado indica evidencia favorable para el primer modelo. ¿Por qué sucede esto?

Resulta que la forma en que se calcula el R cuadrado no penaliza por la inclusión de variables. En la medida que nosotros agregamos más variables al modelo, el R cuadrado sólo puede aumentar en su "poder explicativo" o mantenerse constante. Este comportamiento engaña el criterio de selección de modelos.

El promedio del error cuadrático hace más sentido para justificar el poder predictivo de un modelo de regresión lineal en el sentido que es una medida directa del método de optimización (mínimos cuadrados ordinarios).