

# Machine Learning Guide

En este notebook encontrarás una guía de cómo afrontar un problema de Machine Learning supervisado de clasificación o regresión. Está compuesto de explicaciones teóricas, ayudas en la toma de decisiones, código y enlaces de apoyo.

## 1. Carga datos

Formato de los datos y cantidad de archivos

## 2. Problema Machine Learning

Clasificación o Regresión

## 3. Divide en train y test

Guardamos los datos de test desde el principio

## 4. Target

Distribución del target. ¿Desbalanceado?

## 5. Comprensión de variables

Cómo son tus features

## 6. Feat. Red. Preliminar

Reducción de features antes de empezar la analítica

## 7. Análisis univariante

Primeras impresiones de las variables. Distribuciones

## 8. Análisis bivariate

Búsqueda de relaciones entre las variables

## 9. Eliminación de features

Features con muchos missings o alto grado de cardinalidad

## 10. Duplicados

Comprobamos si el DataFrame tiene duplicados

## 11. Missings

Tratamos los missings

## 12. Anomalías y errores

Detección de datos incoherentes

## 13. Outliers

Tratamos los outliers

## 14. Feature Engineering

14.1 Transformaciones

14.2 Encodings

14.3 Nuevas Features

14.4 Escalados

## 15. Feature Reduction

Filtrado de features por importancia

## 16. Escoger métrica del modelo

## 16. Elegir métrica del modelo

16.1 Métricas de clasificación

16.2 Métricas de regresión

## 17. Decidir qué modelos

Factores que influyen en esta decisión

## 18. Elegir hiperparámetros

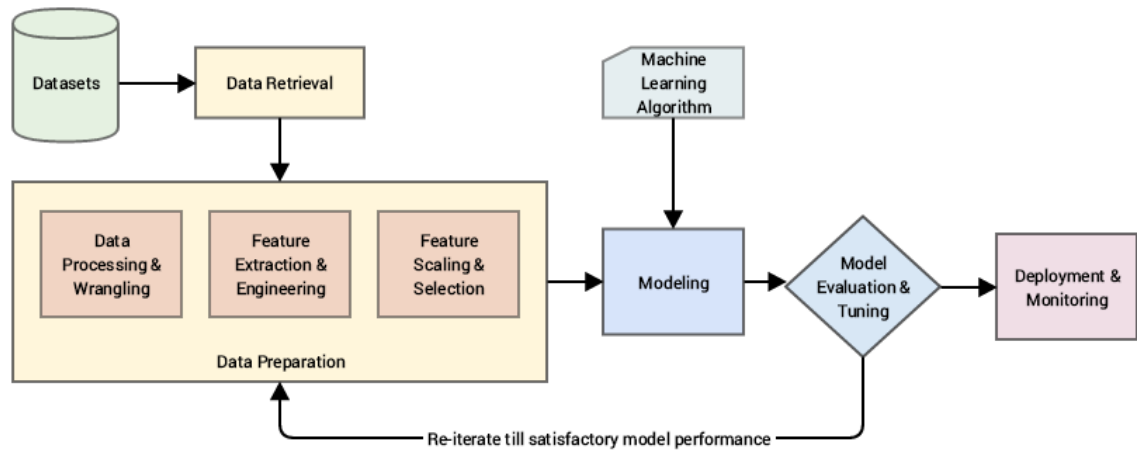
Según el volumen de datos y sus tipos

## 19. Definimos pipelines y probamos

Dependerá de cada modelo. Ejecutamos

## 20. Resultados

Comprobamos si el error se ajusta al problema



```
In [ ]: import warnings
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats

from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

warnings.filterwarnings("ignore")

iris_df = pd.DataFrame(load_iris()["data"], columns=load_iris()
["feature_names"])
iris_df["target"] = load_iris()["target_names"][load_iris()["target"]]

titanic_df = sns.load_dataset('titanic')

diamonds_df = sns.load_dataset('diamonds')

boston_df = pd.DataFrame(load_boston()["data"], columns=load_boston()
["feature_names"])
boston_df["target"] = load_boston()["target"]
```

[Volver al índice](#)

# 1. Carga datos

## Tipo de archivos

Los más típicos [csv](#), [txt](#), [tsv](#), [Excel \(xlsx, xls\)](#), [json](#), [html](#) o [xml](#)

Ten en cuenta que **el csv puede tener distintos separadores**, los datos del Excel pueden estar en una **hoja concreta**, y en una combinación de filas/columnas diferente al origen de la hoja. También podemos acceder a los archivos a través de **una URL**. Se usan las mismas funciones de pandas.

Se recomienda abrir el csv en texto plano para ver el separador.

```
import pandas as pd

df = pd.read_csv("folder/file.csv", sep=";")
df = pd.read_csv("folder/file.csv", sep="\t")
df = pd.read_excel("folder/file.csv", sheet_name="Sheet1")
df = pd.read_json("folder/file.json")

URL = "https://raw.githubusercontent.com/guipsamora/pandas_exercises/master/02_Filtering_%26_Sorting/Euro12/Euro_2012_stats_TEAM.csv"
df = pd.read_csv(URL)
```

## Encoding de los datos

### ► Descripción de encoding

¿Qué dificultades podemos encontrar?

- **En la lectura tengamos un error de encoding.** Habrá que modificar el parámetro `encoding`, adaptándolo al archivo.
  - **El archivo ya se ha creado con el encoding erróneo.** Tendremos caracteres raros en algunas palabras. Un mapeo y sustitución podría valer para solventar el problema.
  - **Encontrar el encoding del archivo.** Es complicado saberlo antes de leer los datos. Aquí tenemos varias opciones:
    - Leer con Python e ir probando los más habituales ( `unicode` , `utf-8` , `latin1` , `ANSI` ). Todas las [funciones de lectura de pandas](#) tienen el tipo de encoding como argumento
    - Abrir el archivo con el bloc de notas. El encoding viene en la esquina inferior derecha.
- Ver imagen

- Usar la librería [`chardet`](https://pypi.org/project/chardet/) para saber el encoding adecuado del archivo.

## ¿Cuántos DataFrames hay que cargar? train/test

Simplemente, dos opciones:

- **Todo el dataset junto:** necesitas un conjunto de test. Verás en el apartado [donde se divide en train y test](#) cómo hacer esta división.
- **Train/test por separado:** si tu conjunto de test no está etiquetado (típico submission file de Kaggle), tendrás que proceder como en el punto anterior. En caso contrario ya tendrás definido tu conjunto de test, y el de train será el set de datos utilizados para el *cross validation* de los modelos.

## Join de datos

No siempre tenemos los datos en un único dataframe, por lo que habrá que unir varios conjuntos de tal manera que haya una columna identificadora única para cada fila, con sus features y target asociados. **¿Cómo procedemos?**

1. **Identifica las claves de cruce:** Cuando juntas dos tablas necesitas al menos una columna en común en ambas, por ejemplo un id de cliente. Para que si en una tabla tienes datos de pedidos y en otra datos personales del cliente, mediante su id, podrás juntar toda esa información en una misma tabla.
2. **Escoge todas las columnas que vas a unir:** no siempre quieres quedarte con todos los datos de ambas tablas, por lo que habrá que elegir los datos a conservar de cada una.

```
left = df_pedidos[['id_cliente', 'pedido', 'descripcion']]
right = df_cliente[['id_cliente', 'dirección', 'edad']]
```

```
result = pd.merge(left, right, how='inner', on=['id_cliente'])
```

Describir cómo son los joins no es el objetivo de este notebook, pero te dejo [este enlace](#) un buen artículo con varios ejemplos de joins.

► Ver tipos de joins

## ¿Me vale esta muestra para entrenar al modelo?

Es muy sencillo cuando tenemos un dataset cerrado, que viene de un concurso de Kaggle, pero en un caso real eso se cumple pocas veces. Hay muchas bases de datos en la empresa, por no hablar de todos los sitios externos (web scraping o APIs) de donde podemos sacar datos. ¿Cómo sabemos que tenemos los datos suficientes para montar un modelo? Si vamos a obtener nuevos, ¿a qué fuentes acudo? ¿cómo es la calidad de estos datos?

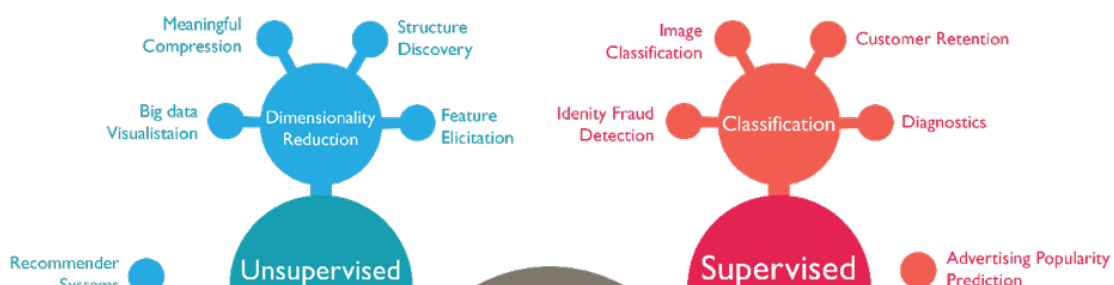
1. **Volumen:** Lo primero, necesitamos un buen volumen de datos. Menos de mil observaciones suele ser escaso para entrenar y testar un modelo de machine learning.

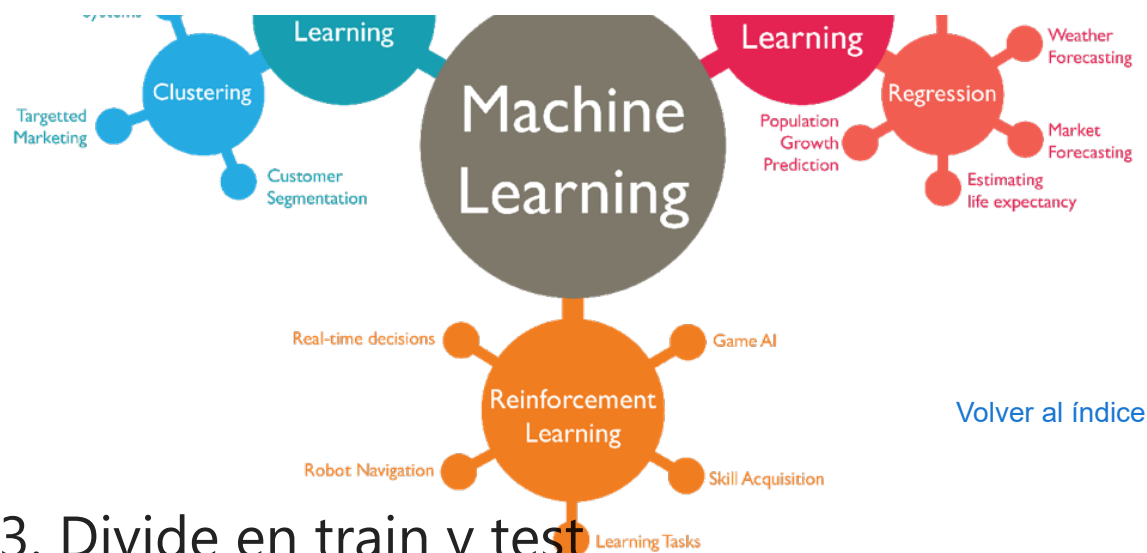
2. **Calidad:** la calidad de los datos siempre que mejor que cantidad. Es mejor encontrar unas pocas features predictivas, cuyos datos sean fiables, que una gran cantidad de features que no aporten nada al modelo. Asegúrate de que los datos conseguidos son buenos y no están manipulados, ni modificados por otros integrantes de la empresa. Y de ser así, si vas a utilizarlos piensa que cuando realices predicciones, las entradas de tu modelo tendrán que ser esos mismos datos modificados.
3. **Caso de uso:** piensa en el problema de negocio y planteate qué variables podrían ser predictivas, y si es factible conseguir esos datos.
4. **Población:** asegúrate de que la población/muestra utilizada para entrenar se asemeje a la población con la que harás predicciones. Por ejemplo, si creas un modelo de tratamiento de imágenes, con el que predigas si unos pulmones tienen cáncer o no, tu modelo no tendrá un buen performance si lo entrenas con muestras de pulmones asiáticos y pretendes predecir muestras de pulmones caucásicos. O si entrenas solo con pulmones de mujeres e intentas predecir sobre pulmones de hombres.
5. **Fuentes externas:** si tienes tiempo planteate acudir a fuentes externas a la empresa, mediante APIs, datasets de kaggle, páginas del gobierno... Por otro lado, no incluyas datos en el entrenamiento que luego no vayas a conseguir para las predicciones. Por ejemplo, si consigues una muestra concreta de datos que publicó una empresa, y ya no se van a publicar más, cuando vayas a hacer predicciones, no vas a poder contar con esos datos. Por último, piensa también que cuantas más fuentes externas, más dependencias tendrá tu modelo para realizar las predicciones y quizá no sea sencillo conseguir ese dato.
6. **Crea tus propios datos:** ¿no tienes datos? "*invéntatelos*". Si necesitas crear un software de reconocimiento de imágenes para saber si alguien lleva gafas o no, saca fotos de amigos o familiares y utilízalas para el modelo. Otra opción es realizar encuestas.

[Volver al índice](#)

## 2. Problema Machine Learning

Existen varios tipos de problemas supervisados de machine learning. Este notebook se centra en problemas de clasificación y regresión, sin series temporales ni RRNN.





### 3. Divide en train y test

#### Algoritmos supervisados

En todo problema de machine learning hay que reservar una porción de los datos para probar nuestros modelos. Es imposible saber a priori qué modelo y qué configuración es la que mejor se ajusta a nuestros datos, por lo que tendremos que probar varias combinaciones (entrenar con train) y después probar con test que modelo da menos errores.

1. **Clasificación:** el target del problema es un conjunto de valores discretos. Dos opciones:
  - Binaria: paga/no paga, fuga de cliente/no fuga, contrae enfermedad/no contrae y así elegir.
  - Multiclase: clasificar fruta (manzanas, peras, naranjas...), vinos (tinto, blanco, rosado)

#### ► 2. **Cómo funciona el train-test-split:**

2. **Regresión:** el objetivo es predecir una variable continua como: distancia, temperatura, ingresos.

¿Cuánto reservamos para test? Entre un 15-30% es buena cifra.

```
In [ ]: X_train, X_test, y_train, y_test =
train_test_split(boston_df.drop('target', axis=1),
                 boston_df['target'],
                 test_size=0.2,
                 random_state=42)
```

#### 2. **PCA:** reducción de dimensionalidad

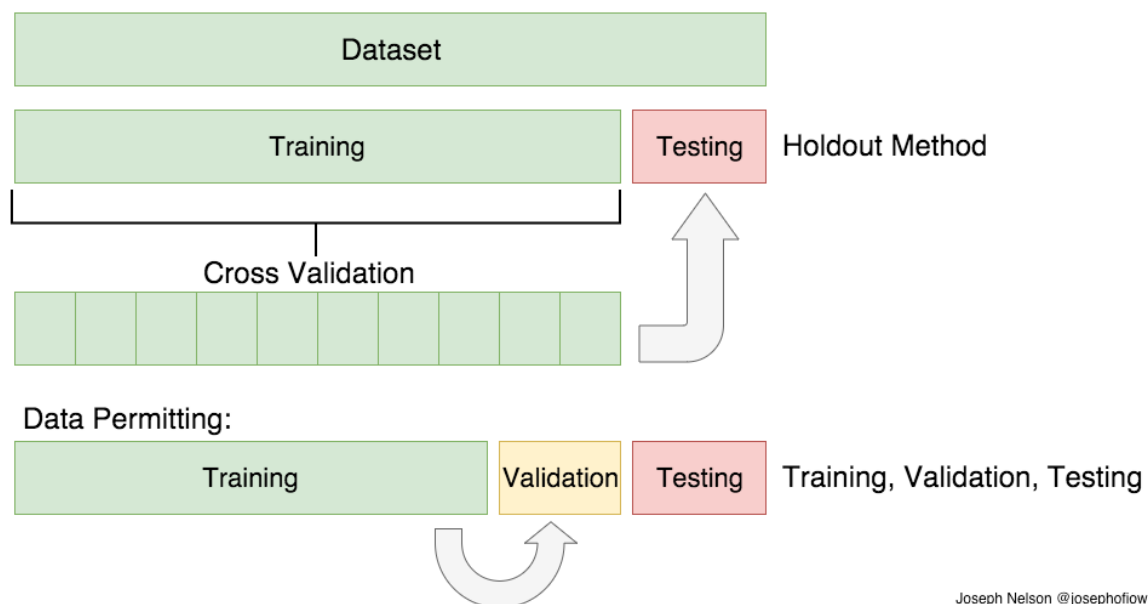
Dentro de la clasificación anterior tendríamos los casos especiales de regresión con series temporales, o clasificación de imágenes/video con redes neuronales. **Este notebook se centra en clasificación y regresión de datos tabulares.**

Es MUY IMPORTANTE conocer el tipo de problema de Machine Learning porque va a ser determinante para toda la analítica que viene a continuación.

Realmente hay que dividir los datos en train, validation y test (ver imagen abajo). El conjunto de validation lo utilizamos para escoger modelo, y el de test para comprobar que nuestro modelo escogido generaliza bien. De momento vamos a preocuparnos de **guardar unos datos para test, y no los tocaremos hasta el final. No podemos contaminar train con los datos de test**. Esto supone seguir ciertas normas:

1. **Scalers**: por ejemplo, si tenemos un StandardScaler en train, usar ese mismo StandardScaler en test (ver ejemplo abajo).
2. **Outliers**: Me temo que los outliers no los voy a poder eliminar en test ya que en un problema real, podría entrar un outlier como input del modelo una vez hayamos hecho la puesta en producción. Cuidado con eliminar registros si estamos en una competición de Kaggle porque la cantidad de muestras que van a test no puede variar.
3. **Missings**: si aplicábamos la media/mediana/moda en train, aplicar esa misma métrica en test. No apliques la media de test en train.
4. **Feature reduction**: eliminar las features que quitábamos en train.
5. **Feature engineering**: mismos cálculos que en train.

**NOTA:** si se trata de un concurso de Kaggle, en el que ya te dan el conjunto de test aparte, no hace falta que dividas tu train para obtener un nuevo conjunto de test, sobre todo si tienes pocos datos. Utiliza todo el conjunto de train en el *cross validation* para escoger el mejor modelo, y después el test proporcionado por Kaggle para realizar una submission.



Joseph Nelson @josephoflwa



```
In [ ]: # Ejemplo de código para un StandardScaler
        from sklearn.preprocessing import StandardScaler

        # Almaceno en el objeto scaler todo lo necesario para estandarizar, con
        # los datos de train
        scaler = StandardScaler()
        scaler.fit(X_train)

        # Utilizo los datos de train para escalar train y test.
        X_train_scaled = scaler.transform(X_train)
        X_test_scaled = scaler.transform(X_test)
```

[Volver al índice](#)

## 4. Target

Tenemos que echar un pequeño vistazo al target ya que hay ciertas casuísticas que no queremos tener, como por ejemplo un dataset desbalanceado o un target asimétrico.

Problema ML	Caso deseado	Caso no deseado
Clasificación	Dataset balanceado	Dataset desbalanceado
Regresión	Distribución normal	Distribución asimétrica

### Clasificación

Si intentamos predecir si una persona va a padecer un cáncer o no, y entrenamos un modelo en el que el 97% de los casos no ha padecido cáncer, si el modelo predice siempre que no tiene cáncer, conseguirá un 97% de precisión. Y no es lo que estamos buscando. Queremos profundizar más en ese 3%, y que el modelo sepa discriminar bien entre una clase u otra.

**Esto es lo que se conoce como dataset desbalanceado.** Lo ideal es que el dataset esté lo más equilibrado posible. ¿Cómo lidiar con ello?

1. **Cambiando la métrica:** recall, f1-score o ROC curve (ver apartado de métricas).
2. **Consiguiendo más datos:** Parece una tontería pero muchas veces ni nos lo planteamos
3. **Resampling:** poner más copias del target con menor cantidad (over-sampling) o al revés (under-sampling), quitar datos del caso con mayor cantidad de datos. Under-sampling está bien cuando tenemos muchos datos, mientras que over-sampling lo tendremos que considerar cuando sean escasos. Prueba varios ratios.
4. **Generar datos sintéticos:** existen funciones de sklearn que son capaces de generar nuevos datos, como SMOTE (Synthetic Minority Over-sampling Technique)
5. **Prueba nuevos algoritmos:** los árboles de decisión suelen funcionar bien en problemas desbalanceados.
6. **Penalización en los modelos:** modelos que presten más atención a la clase minoritaria.
7. **Weights:** Es posible aplicarle un peso diferente a cada clase, para que esté más balanceadas, para ello usamos el parámetro `class_weight`.

```
In [ ]: !pip install imblearn

from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler

df = pd.read_csv("data/balance-scale.csv")

rus = RandomUnderSampler(random_state=42)
X_rus, y_rus = rus.fit_resample(df.loc[:, 'B':'1.3'], df['balance'])

ros = RandomOverSampler(random_state=42, sampling_strategy=0.5)
X_ros, y_ros = ros.fit_resample(df.loc[:, 'B':'1.3'], df['balance'])
```

Requirement already satisfied: imblearn in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (0.0)

Requirement already satisfied: imbalanced-learn in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (from imblearn) (0.10.1)

Requirement already satisfied: numpy>=1.17.3 in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (from imbalanced-learn->imblearn) (1.21.6)

Requirement already satisfied: scipy>=1.3.2 in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (from imbalanced-learn->imblearn) (1.7.1)

Requirement already satisfied: scikit-learn>=1.0.2 in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (from imbalanced-learn->imblearn) (1.0.2)

Requirement already satisfied: joblib>=1.1.1 in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (from imbalanced-learn->imblearn) (1.2.0)

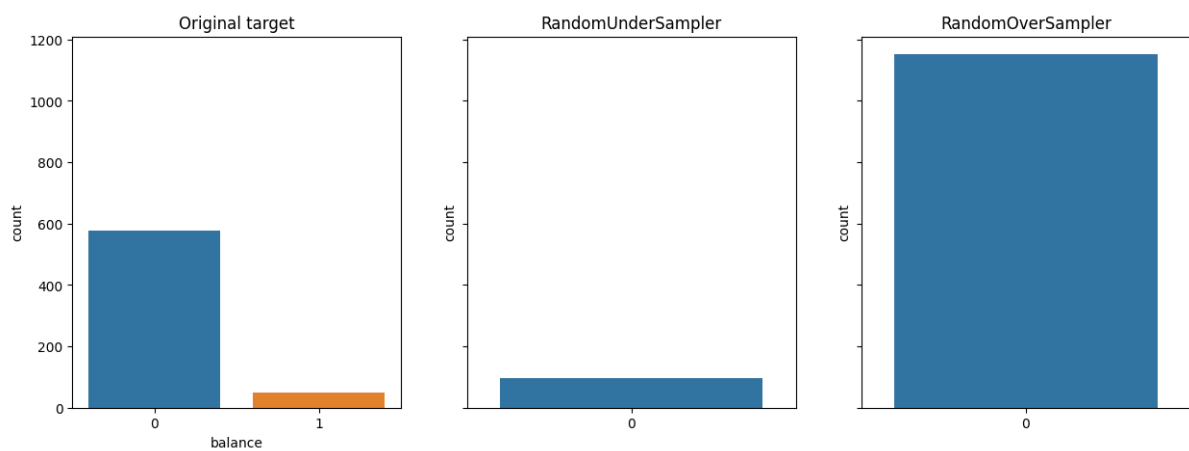
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\rafael ortega\appdata\local\programs\python\python37\lib\site-packages (from imbalanced-learn->imblearn) (3.1.0)

```
In [ ]: fig, axes = plt.subplots(1, 3, figsize=(15, 5), sharey=True)

# Original target
sns.countplot(x="balance", data=df, ax=axes[0])
axes[0].set_title("Original target")

# RandomUnderSampler
sns.countplot(y_rus, ax=axes[1])
axes[1].set_title("RandomUnderSampler")

# RandomOverSampler
sns.countplot(y_ros, ax=axes[2])
axes[2].set_title("RandomOverSampler");
```



### Algunas consideraciones antes de utilizar esta técnica

- Nunca testear sobre el conjunto sampleado. Siempre sobre el conjunto de test original
- Si realizamos CrossValidation, siempre samplear DURANTE el CrossValidation, no antes. Es decir, meterlo en el Pipeline.

## Regresión

Lo mejor para los modelos es que tengamos todos los datos escalados y con una distribución normal. Algunos de los algoritmos asumen ese tipo de distribuciones en los datos como las regresiones lineales o las redes neuronales. Ya que esto rara vez ocurre, hay que transformar las variables. Para estandarizar una distribución podemos aplicar varios tipos de transformaciones ([ver apartado transformaciones](#)):

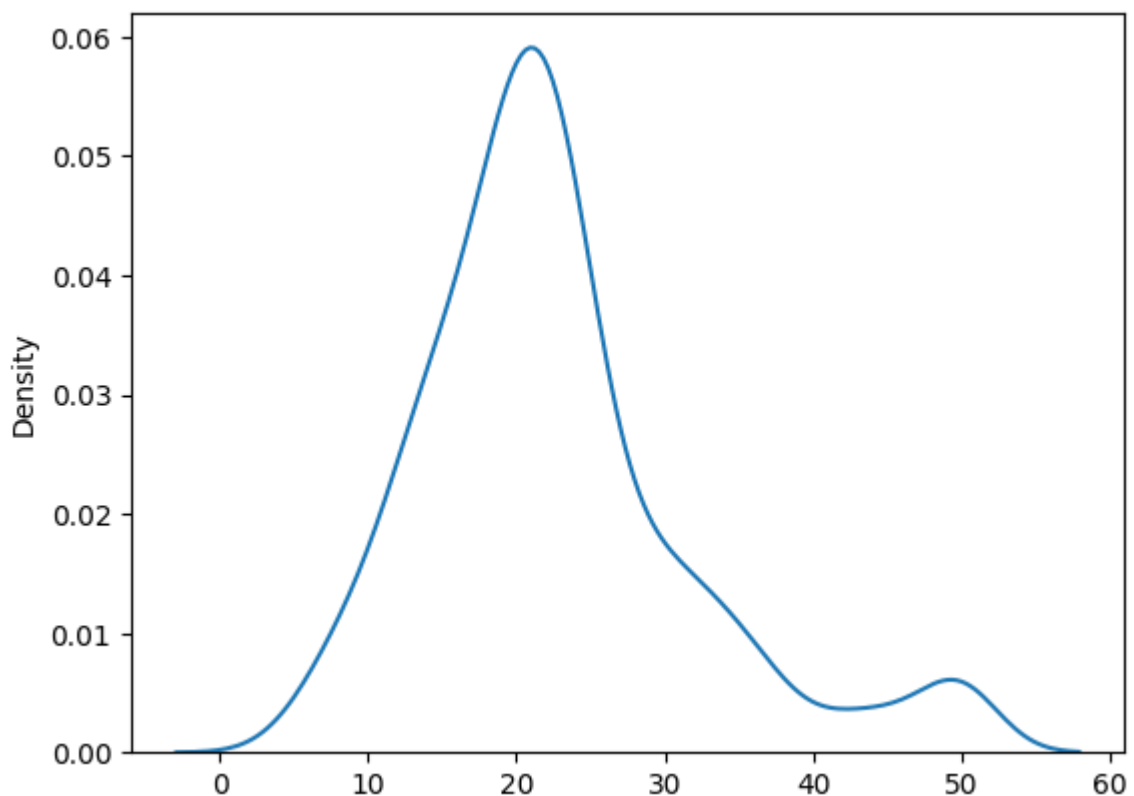
1. Logarítmica
2. Box-cox
3. Cuadrado/cúbica

Después puedes comprobar cuánto de normal es una variable mediante el **test de Shapiro** o un **Q-Q plot**.

### ► Q-Q plot

```
In [ ]: from sklearn.datasets import load_boston

boston_target = load_boston()["target"]
sns.kdeplot(boston_target);
```



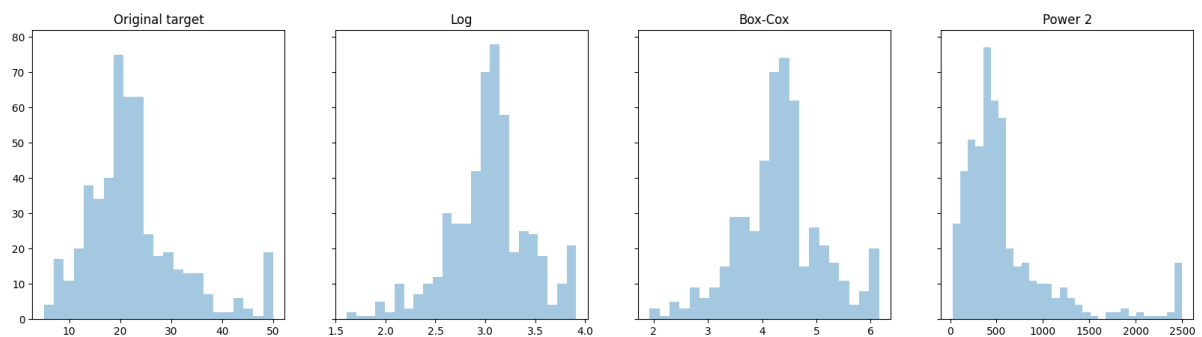
```
In [ ]: fig, axes = plt.subplots(1, 4, figsize=(20, 5), sharey=True)

# Original target
sns.distplot(boston_target, kde=False, ax=axes[0])
axes[0].set_title("Original target")

# Logaritmico
sns.distplot(np.log(boston_target),kde=False, ax=axes[1])
axes[1].set_title("Log")

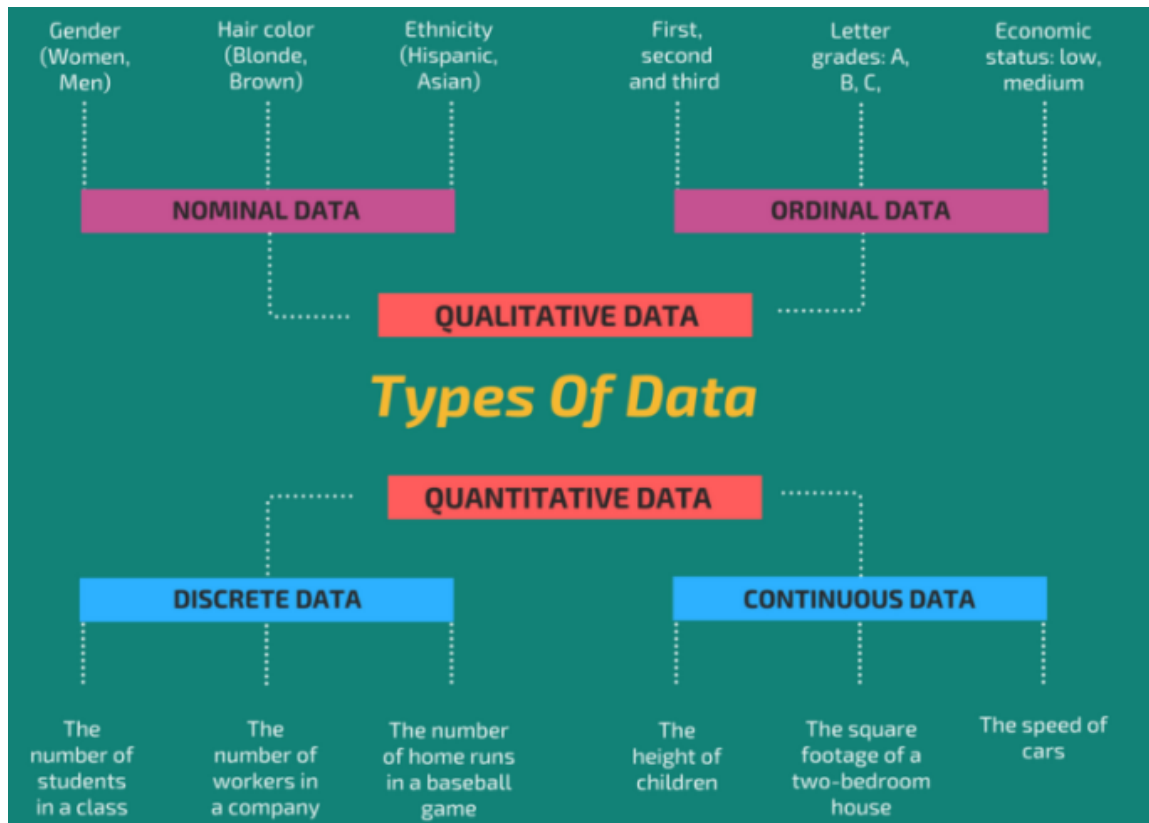
# Box-cox
sns.distplot(stats.boxcox(boston_target)[0],kde=False, ax=axes[2])
axes[2].set_title("Box-Cox");

# Power 2
sns.distplot(np.power(boston_target, 2),kde=False, ax=axes[3])
axes[3].set_title("Power 2");
```



**NOTA:** Después de aplicar una de estas transformaciones, habrá que volver a aplicar las transformaciones inversas sobre las predicciones. Si aplicamos logaritmo, luego irá `np.expm()`, box-cox con `scipy.special.inv_boxcox()` y la raíz enésima, en el caso de elevar a la potencia enésima.

## 5. Comprensión de variables



Antes de ponernos a programar tenemos que entender muy bien cuál es la problemática y de qué datos disponemos, ya que este momento puede ser un punto de inflexión importante. Quizá con un pequeño análisis previo nos demos cuenta de que no hace falta un modelo de machine learning para solventar nuestro problema, o que no tenemos suficientes datos para realizar las predicciones planteadas inicialmente.

Realiza una pequeña analítica de cada una de las variables. Esta analítica puede ir perfectamente en un Excel y nos va a ayudar a comprender la naturaleza de las variables y el sentido de las mismas respecto al target. Deberás recabar la siguiente información:

1. **Variable:** nombre variable/alias
2. **Data type:** cualitativa, cuantitativa, ordinal, continua...¿?
3. **Segmento:** clasificar las variables según su significado. Si son variables demográficas, económicas, identificadores, tiempo...
4. **Expectativas:** un pequeño indicador personal de si resultará útil la variable. ¿Necesito esta variable para la solución? ¿Cómo de importante será esta variable? ¿Esta info la recoge otra variable ya vista?
5. **Conclusiones:** después del análisis anterior, llegar a unas conclusiones sobre la importancia de la variable.

**¿Por qué hacemos esto?** Si nuestro problema tiene muchas features, realizar una selección "a ojo" puede resultar muy útil, siempre y cuando se haga con criterio de negocio. Además, podría darse el caso de empezar el EDA pre-modelo erróneamente sin darnos cuenta, y este pequeño análisis nos servirá para orientarnos ante discrepancias durante el análisis y posibles rectificaciones tempranas.

Típicos métodos para echar un primer vistazo al dataset:

```
In [ ]: # Estadísticos
diamonds_df.describe()
diamonds_df.describe(include='all')

# Tipos de los datos
diamonds_df.dtypes

# Tipos de los datos y missings
diamonds_df.info()

# Columnas del dataset
diamonds_df.columns

# dimensiones del dataset
print("Filas:", diamonds_df.shape[0])
print("Columnas:", diamonds_df.shape[1])
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype  
---  -
 0   carat       53940 non-null  float64
 1   cut         53940 non-null  category
 2   color       53940 non-null  category
 3   clarity     53940 non-null  category
 4   depth       53940 non-null  float64
 5   table       53940 non-null  float64
 6   price       53940 non-null  int64   
 7   x           53940 non-null  float64
 8   y           53940 non-null  float64
 9   z           53940 non-null  float64
dtypes: category(3), float64(6), int64(1)
memory usage: 3.0 MB
Filas: 53940
Columnas: 10
```



## Forzar el data type

Fuerza los tipos de los datos. Si sabes que un float es un float, un str es un str, fuérzalo en el código para evitar errores posteriores. Hay que tener bien identificados los tipos de los datos.

Por supuesto, probablemente haya missings en los datos. La conversión de estos missings da error, pero lo solventamos (de momento) con el argumento `errores='ignore'`.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 624 entries, 0 to 623
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype  
---  -
 0   Unnamed: 0   624 non-null   int64  
 1   B            624 non-null   object  
 2   1            624 non-null   int64  
 3   1.1          624 non-null   int64  
 4   1.2          624 non-null   int64  
 5   1.3          624 non-null   int64  
 6   balance      624 non-null   int64  
dtypes: int64(6), object(1)
memory usage: 34.2+ KB
```

```
In [ ]: from seaborn import load_dataset
df = load_dataset('titanic')

# Diccionario con tipo en clave y valor columnas
new_types = {
    int: ['survived', 'age'],
    float: ['fare'],
    str: ['embarked', 'embark_town'],
    bool: ['alone']
}

for key, value in new_types.items():
    for i in value:
        df[i] = df[i].astype(key, errors='ignore')

# df['fare'] = df['fare'].astype(float)
```

## Reporte de variables

Un pequeño reporte de las columnas, con sus tipos, % de missings y cardinalidad, resultará muy útil a lo largo de la analítica.

```
In [ ]: def data_report(df):  
    # Sacamos Los NOMBRES  
    cols = pd.DataFrame(df.columns.values, columns=["COL_N"])  
  
    # Sacamos Los TIPOS  
    types = pd.DataFrame(df.dtypes.values, columns=["DATA_TYPE"])  
  
    # Sacamos Los MISSINGS  
    percent_missing = round(df.isnull().sum() * 100 / len(df), 2)  
    percent_missing_df = pd.DataFrame(percent_missing.values, columns=  
["MISSINGS (%)"])  
  
    # Sacamos Los VALORES UNICOS  
    unicos = pd.DataFrame(df.nunique().values, columns=  
["UNIQUE_VALUES"])  
  
    percent_cardin = round(unicos['UNIQUE_VALUES']*100/len(df), 2)  
    percent_cardin_df = pd.DataFrame(percent_cardin.values, columns=  
["CARDIN (%)"])  
  
    concatenado = pd.concat([cols, types, percent_missing_df, unicos,  
percent_cardin_df], axis=1, sort=False)  
    concatenado.set_index('COL_N', drop=True, inplace=True)  
  
    return concatenado.T  
  
data_report(df)
```

Out[ ]:

COL_N	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who
DATA_TYPE	int32	int64	object	float64	int64	int64	float64	object	category	object
MISSINGS (%)	0.0	0.0	0.0	19.87	0.0	0.0	0.0	0.0	0.0	0.0
UNIQUE_VALUES	2	3	2	88	7	7	248	4	3	1
CARDIN (%)	0.22	0.34	0.22	9.88	0.79	0.79	27.83	0.45	0.34	0.34

[Volver al índice](#)

## 6. Feat. Red. Preliminar

En ocasiones es casi imposible llevar a cabo un análisis exploratorio si tienes una gran cantidad de features. Si ese es tu caso, deberías seguir alguna de las recomendaciones de este apartado.

Básicamente lo que haremos es eliminar algunas features. ¿Qué criterios seguiremos?

1. **Muchos missings:** a partir de un 40% de missings son features difícilmente salvables.
2. **Features repetidas:** columnas repetidas. Asegúrate bien de que sea así.
3. **Identificadores:** ids, nombres únicos. Son features que no aportan nada
4. **Feature selection**
  - 4.1 **Correlación lineal:** coeficientes de Pearson entre  $[-0.1, +0.1]$  respecto a nuestro target, suelen ser variables que no tiene nada que ver con el mismo.
  - 4.2 **Feature importance:** podemos tirar un modelo rápidamente que nos de la importancia de cada feature respecto al target: RandomForest se suele usar.
  - 4.3 **Varianza:** variables que no tienen varianza no nos interesan. El caso extremo sería una constante. No nos aporta nada. Es útil para ver que las variables pueden servir para el modelo pero no nos da una relación con el target.
  - 4.4 **Estadísticos:** la función de sklearn `SelectKBest` permite aplicar un test estadístico entre el target y cada uno de las features para determinar qué variables están relacionadas.

En este momento lo que se pretende es eliminar features que sabes con certeza que son muy inútiles, por lo que los criterios para eliminar features en este punto del análisis serán mucho menos restrictivos que en posteriores (no queremos cargarnos información sin conocer bien el dataset). De momento, será suficiente con seguir los puntos 1, 2 y 3. El punto 4 se aplicarán si la cantidad de features es muy grande, por ejemplo más de 50 features.

► **¿Qué es la correlación lineal?**

► **¿Qué es el feature importance de un modelo?**

**Te recomiendo que utilices la función del apartado de comprensión de variables para eliminar las que sean identificadores o tengan muchísimos missings**

Muchos missings

```
In [ ]: df = pd.read_csv('data/data with missings.csv')

precent_missing = df.isnull().sum()*100/len(df)
missing_value_df = pd.DataFrame({'column_name': df.columns,
                                'percent_missing':
precent_missing}).sort_values('percent_missing', ascending=False)
missing_value_df
```

```
Out [ ]:
```

	column_name	percent_missing
<b>PoolQC</b>	PoolQC	99.520548
<b>MiscFeature</b>	MiscFeature	96.301370
<b>Alley</b>	Alley	93.767123
<b>Fence</b>	Fence	80.753425
<b>FireplaceQu</b>	FireplaceQu	47.260274
...	...	...
<b>ExterQual</b>	ExterQual	0.000000
<b>Exterior2nd</b>	Exterior2nd	0.000000
<b>Exterior1st</b>	Exterior1st	0.000000
<b>RoofMatl</b>	RoofMatl	0.000000
<b>SalePrice</b>	SalePrice	0.000000

81 rows × 2 columns

```
In [ ]: cols_to_drop = missing_value_df[missing_value_df['percent_missing'] >
50].index.values
print("Cols:", cols_to_drop)

print("Columnas pre drop:", len(df.columns))

df.drop(columns=cols_to_drop, inplace=True)

print("Columnas post drop:", len(df.columns))
```

```
Cols: ['PoolQC' 'MiscFeature' 'Alley' 'Fence']
Columnas pre drop: 81
Columnas post drop: 77
```

Correlación lineal

```
In [ ]: df = pd.DataFrame(load_boston()["data"], columns=load_boston()
["feature_names"])
df["target"] = load_boston()["target"]

# Borra las columnas con una correlacion menor a 0.2
corr = np.abs(df.corr()['target']).sort_values(ascending=True)
print(corr)

bad_corr_feat = corr[corr < 0.2].index.values
print(bad_corr_feat)

df.drop(columns = bad_corr_feat, inplace = True)
```

```
CHAS      0.175260
DIS       0.249929
B         0.333461
ZN        0.360445
AGE       0.376955
RAD       0.381626
CRIM      0.388305
NOX       0.427321
TAX       0.468536
INDUS     0.483725
PTRATIO   0.507787
RM        0.695360
LSTAT     0.737663
target    1.000000
Name: target, dtype: float64
['CHAS']
```

Feature importance RandomForest

```
In [ ]: from sklearn.datasets import load_boston
        from sklearn.ensemble import RandomForestRegressor

        import numpy as np
        #Load boston housing dataset as an example
        boston = load_boston()
        X = boston["data"]
        Y = boston["target"]
        names = boston["feature_names"]
        rf = RandomForestRegressor(n_estimators = 100)
        rf.fit(X, Y)

        scores = sorted(zip(map(lambda x: round(x, 4), rf.feature_importances_),
                                names), reverse=True)
        pd.DataFrame(scores, columns=['Score', 'Feature'])
```

Out[ ]:

	Score	Feature
--	-------	---------

0	0.4110	RM
1	0.3943	LSTAT
2	0.0649	DIS
3	0.0373	CRIM
4	0.0242	NOX
5	0.0160	PTRATIO
6	0.0144	TAX
7	0.0144	AGE
8	0.0114	B
9	0.0065	INDUS
10	0.0035	RAD
11	0.0013	ZN
12	0.0008	CHAS

Feature importance estadístico

```
In [ ]: from sklearn.feature_selection import SelectKBest
```

```
print(X.shape)
sel = SelectKBest(k=5)
X_new = sel.fit_transform(X, Y)
print(X_new.shape)
print(sel.scores_)

pd.DataFrame({'column': names, 'score':
sel.scores_}).sort_values('score', ascending=False)
```

```
(506, 13)
```

```
(506, 5)
```

```
[3.41392257 1.69825318 2.15122      1.1023291  2.33899533 2.6759978
 2.32954454 1.50668687 1.91464853 2.21005614 1.77625065 1.81833191
 5.75215088]
```

```
Out[ ]:
```

	column	score
12	LSTAT	5.752151
0	CRIM	3.413923
5	RM	2.675998
4	NOX	2.338995
6	AGE	2.329545
9	TAX	2.210056
2	INDUS	2.151220
8	RAD	1.914649
11	B	1.818332
10	PTRATIO	1.776251
1	ZN	1.698253
7	DIS	1.506687
3	CHAS	1.102329



[Volver al índice](#)

## 7. Análisis univariante

Antes de meternos de lleno con los modelos, tenemos que realizar un análisis exploratorio para comprender qué datos tenemos entre manos.

### ¿Qué buscamos con este análisis?

1. Comprobar que las variables tienen sentido y sus unidades de medida también son correctas
2. Detectar outliers
3. Detección de errores en la variable
4. Posibles features a eliminar
5. Detectar distribuciones extrañas. Buscamos siempre distribuciones normales en los datos. Es como mejor trabajan los modelos. Si las distribuciones no son normales, habrá que transformarlas (ver apartado transformaciones).

**¿Qué hacemos si tenemos muchas variables en el dataset?** Si es el caso, deberíamos utilizar una matriz de correlación o correr algún modelo para que nos de un indicador de qué features serán las más predictivas. Podemos empezar por esas features. Otra manera de decidir qué features escoger es mediante un análisis de negocio. Si vas a predecir si un paciente padece una enfermedad, o la temperatura que hará mañana, o la probabilidad de fuga de un cliente en una telco, piensa qué variables necesitarías en un modelo de ese estilo, y cuáles de las que disponen cumplen con esos criterios.

#### ► Tipos de gráficas

#### ► Gráficas en función de tus datos

Clasificación, feature numérica

```
In [ ]: fig, axes = plt.subplots(2, 3, figsize=(20, 10))

# Funcion de densidad
sns.distplot(iris_df['sepal length (cm)'], hist = False, ax=axes[0, 0])
axes[0, 0].set_title("Función de densidad")

# Histograma
sns.distplot(iris_df['sepal length (cm)'],
             kde=False,
             color='slategray',
             ax=axes[0, 1]);

axes[0, 1].set_title("Histograma")

# Funcion de densidad + histograma
sns.distplot(iris_df['sepal length (cm)'],
             kde_kws={"color": "k", "lw": 3, "label": "KDE"},
             hist_kws={"histtype": "step", "linewidth": 3,
                       "alpha": 1, "color": "g"},
             ax=axes[0, 2])

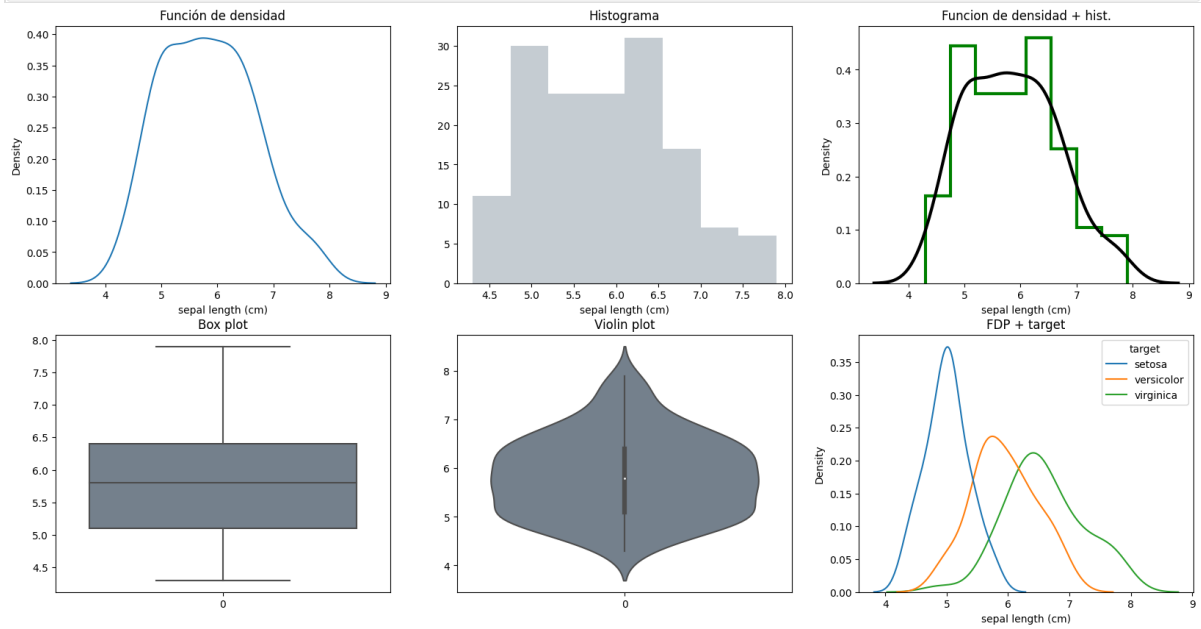
axes[0, 2].set_title("Funcion de densidad + hist.")

# Boxplot
sns.boxplot(iris_df['sepal length (cm)'], color="slategray", ax=axes[1,
0])
axes[1, 0].set_title("Box plot")

# Violin plot
sns.violinplot(iris_df['sepal length (cm)'], color="slategray",
ax=axes[1, 1])
axes[1, 1].set_title("Violin plot")

# Funcion de densidad + target
sns.kdeplot(data=iris_df, x='sepal length (cm)', hue = 'target',
ax=axes[1, 2])
```

```
axes[1, 2].set_title("FDP + target");
```



## Clasificación feature categórica

```
In [ ]: len(titanic_df["survived"])
```

```
Out[ ]: 891
```

```
In [ ]: fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Conteo de categorica
sns.countplot(titanic_df["embark_town"], ax=axes[0])
axes[0].set_title("Variable categórica")

# Categórica vs target
sns.countplot(titanic_df["embark_town"], hue=titanic_df['survived'],
ax=axes[1], dodge=True)
axes[1].set_title("Categórica vs target");
```

```

-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16640\2253441205.py in <module>
      2
      3 # Conteo de categorica
----> 4 sns.countplot(titanic_df["embark_town"], ax=axes[0])
      5 axes[0].set_title("Variable categórica")
      6

c:\Users\Rafael Ortega\AppData\Local\Programs\Python\Python37\lib\site-packages\seaborn\categorical.py in countplot(data, x, y, hue, order, hue_order, orient, color, palette, saturation, width, dodge, ax, **kwargs)
    2945     estimator, errorbar, n_boot, units, seed,
    2946     orient, color, palette, saturation,
-> 2947     width, errcolor, errwidth, capsize, dodge
    2948 )
    2949

c:\Users\Rafael Ortega\AppData\Local\Programs\Python\Python37\lib\site-packages\seaborn\categorical.py in __init__(self, x, y, hue, data, order, hue_order, estimator, errorbar, n_boot, units, seed, orient, color, palette, saturation, width, errcolor, errwidth, capsize, dodge)
    1529     """Initialize the plotter."""
    1530     self.establish_variables(x, y, hue, data, orient,
-> 1531                             order, hue_order, units)
    1532     self.establish_colors(color, palette, saturation)
    1533     self.estimate_statistic(estimator, errorbar, n_boot, seed)

c:\Users\Rafael Ortega\AppData\Local\Programs\Python\Python37\lib\site-packages\seaborn\categorical.py in establish_variables(self, x, y, hue, data, orient, order, hue_order, units)
    514
    515         # Convert to a list of arrays, the common representation
--> 516         plot_data = [np.asarray(d, float) for d in plot_data]
    517
    518         # The group names will just be numeric indices

c:\Users\Rafael Ortega\AppData\Local\Programs\Python\Python37\lib\site-packages\seaborn\categorical.py in <listcomp>(.0)
    514
    515         # Convert to a list of arrays, the common representation
--> 516         plot_data = [np.asarray(d, float) for d in plot_data]
    517
    518         # The group names will just be numeric indices

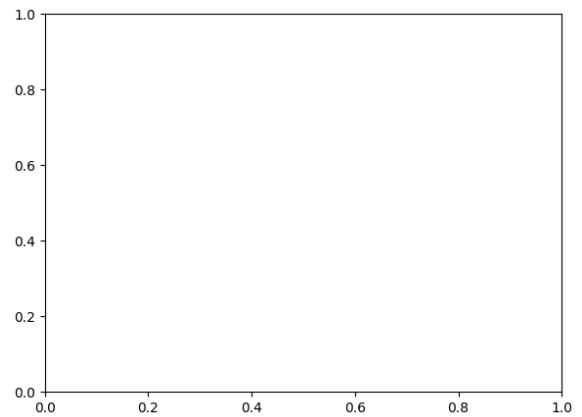
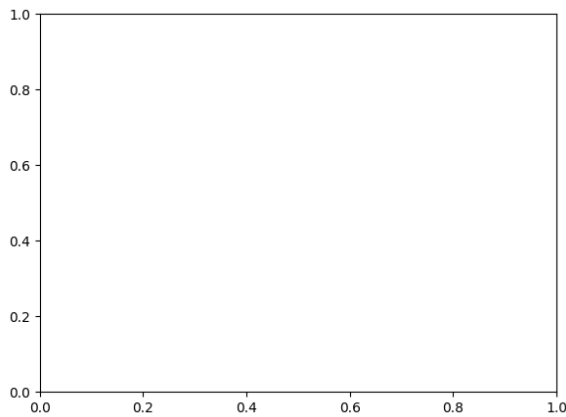
c:\Users\Rafael Ortega\AppData\Local\Programs\Python\Python37\lib\site-packages\pandas\core\series.py in __array__(self, dtype)
    855         dtype='datetime64[ns]')
    856         """
--> 857         return np.asarray(self._values, dtype)
    858
    859         #

```

```

ValueError: could not convert string to float: 'Southampton'

```



## Regresión (numérica y categórica)

```
In [ ]: # Target de precio en el dataset de diamonds
fig, axes = plt.subplots(1, 3, figsize=(20, 5))

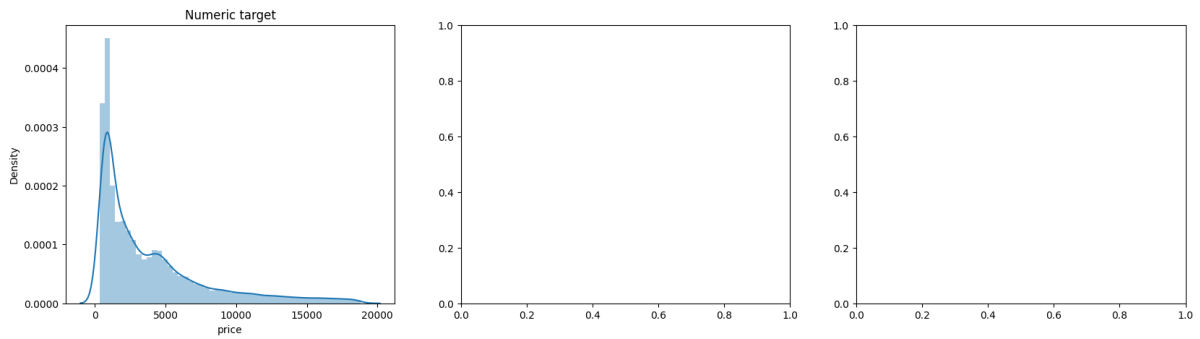
# Numeric target
sns.distplot(diamonds_df['price'], ax = axes[0])
axes[0].set_title("Numeric target")

# Numeric vs Numeric
sns.scatterplot(diamonds_df['price'], diamonds_df['carat'], ax =
axes[1])
axes[1].set_title("Numeric vs Numeric")

# Categorical vs target
sns.boxplot(data=diamonds_df, x='cut', y='price', ax=axes[2])
axes[2].set_title("Numeric vs categorical");
```

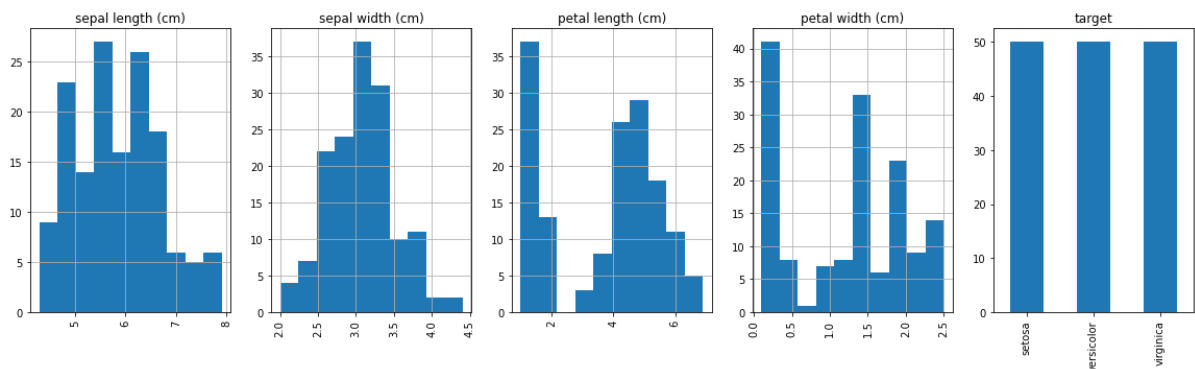
```
-----
TypeError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16640\2322037341.py in <module>
7
8 # Numeric vs Numeric
----> 9 sns.scatterplot(diamonds_df['price'], diamonds_df['carat'], ax = axes[1])
10 axes[1].set_title("Numeric vs Numeric")
11
```

**TypeError:** scatterplot() takes from 0 to 1 positional arguments but 2 positional arguments (and 1 keyword-only argument) were given



## Código para análisis rápido univariante

```
In [ ]: fig = plt.figure(figsize=(20, 5))
cols = 5
rows = int(np.ceil(float(iris_df.shape[1]) / cols))
for i, column in enumerate(iris_df.columns):
    ax = fig.add_subplot(rows, cols, i + 1)
    ax.set_title(column)
    if iris_df.dtypes[column] == np.object:
        iris_df[column].value_counts().plot(kind="bar", axes=ax)
    else:
        iris_df[column].hist(axes=ax)
        plt.xticks(rotation="vertical")
plt.subplots_adjust(hspace=0.7, wspace=0.2)
```



[Volver al índice](#)

## 8. Análisis bivalente

Utilizamos este tipo de análisis para ver cómo son las relaciones entre nuestros datos, dos a dos. Los objetivos de este análisis son:

1. **Relación con el target:** ver qué tipo de relación hay entre cada feature con el target. Con el tipo de relación intuiremos el modelo que mejor le vendrá a nuestros datos.
2. **Eliminar algunas features:** comprobar si hay features con una alta correlación lineal entre ellas. Eso significa dos cosas:
  - Si hay dos features que se parecen mucho, sobra una. ¿Cuál? La que tenga mas missings o menor correlación con el target
  - Si dos features tienen una correlación lineal alta podemos sufrir de multicolinealidad. Te dejo [este artículo](#) para más información.

Podemos llevar a cabo dos tipos de análisis bivalente, uno mediante visualizaciones, y otro con tests y medidas estadísticas. Las visualizaciones serían el método más artesanal de ver esas relaciones entre los datos, mientras que con las medidas estadísticas puedo establecer un criterio algo más automatizado.

### ¿Qué visualizaciones puedo hacer?

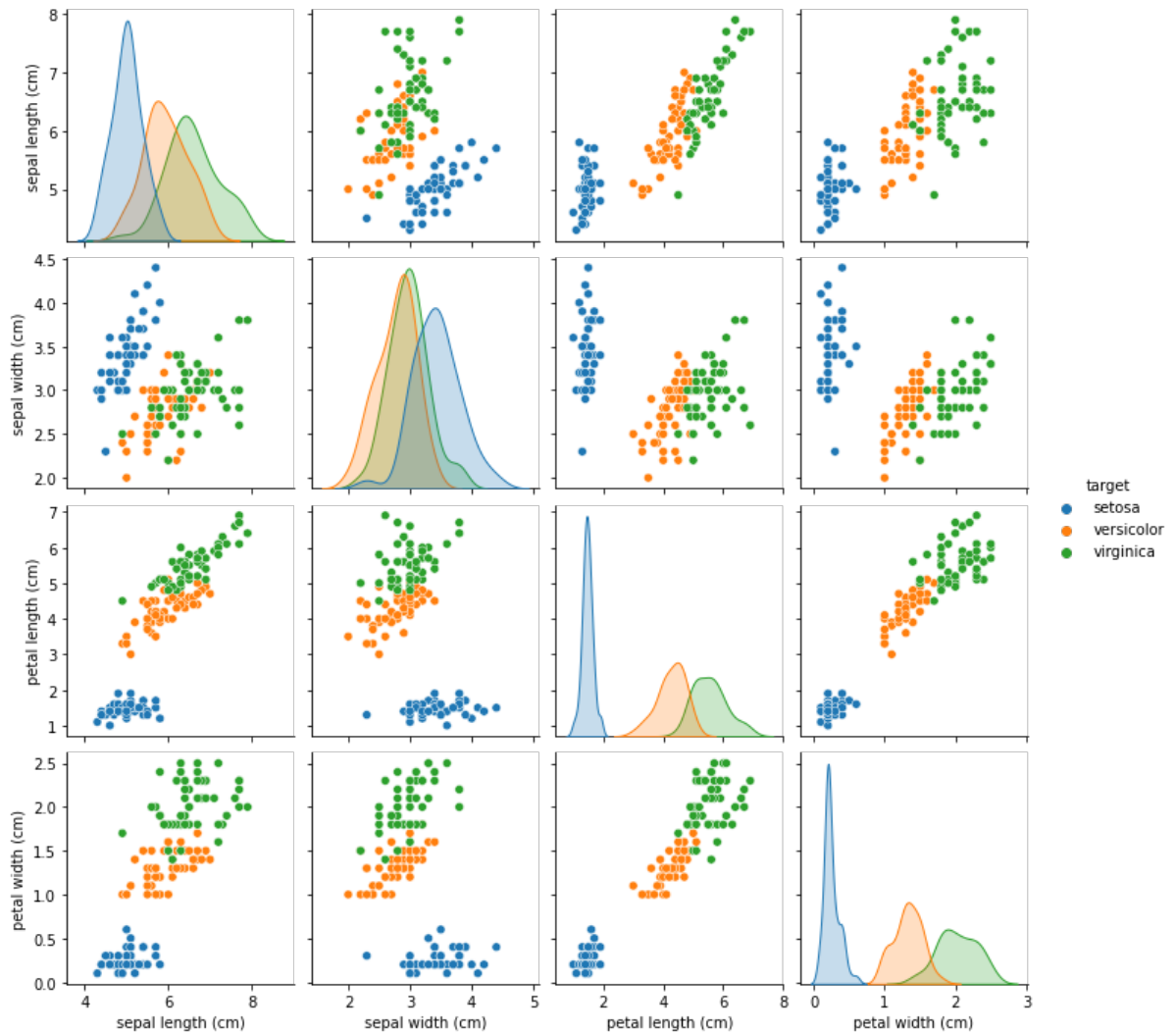
Lo mejor es hacer un grid de scatterplots. Suena muy rimbombante, pero en seaborn no es más que una línea de código. Resulta útil si tenemos todo variables numéricas, pero para categóricas no nos vale, tendremos que acudir a tablas y diagramas de barras stacked o agrupadas.

### ¿Qué medidas estadísticas puedo utilizar?

Lo habitual es utilizar el coeficiente de pearson (lo que te da una matriz de correlación) para cada dos variables, ya que es un valor normalizado entre  $[-1, 1]$ . Si el valor está en torno a 0, las dos variables tienen muy poca correlación lineal, y cuanto más cercano a 1 o -1 mejor será su correlación lineal directa o inversa respectivamente. A partir de  $\pm 0.6$  o  $0.7$  suele ser una buena correlación. No obstante un coeficiente de pearson alto [no siempre asegura una buena relación entre los datos](#)

El problema viene cuando queremos ver relaciones entre variables categóricas. En este caso ya no es tan útil el coeficiente de pearson. **Para detectar este tipo de relaciones recomiendo utilizar el coeficiente de Phik o Cramér's V.** En la [documentación de pandas profiling](#) tienes algunos ejemplos.

```
In [ ]: # Variables numericas vs target categorico
sns.pairplot(iris_df,
             kind='scatter',
             hue='target');
```



```
In [ ]: # Categórica vs categorica
pd.crosstab(titanic_df['survived'],
            titanic_df['sex'])
```

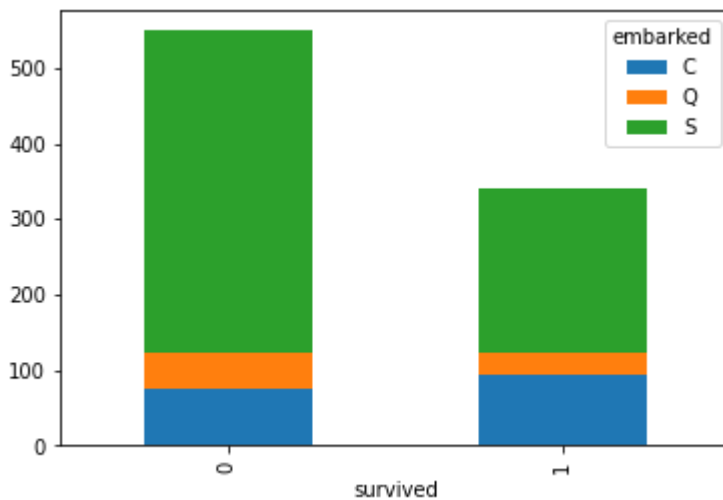
```
Out[ ]:      sex  female  male
survived
0         81    468
1        233    109
```



```
In [ ]: titanic_df.groupby(['embarked',
    'survived']).size().reset_index().pivot(columns='embarked',
    index='survived', values=0)
```

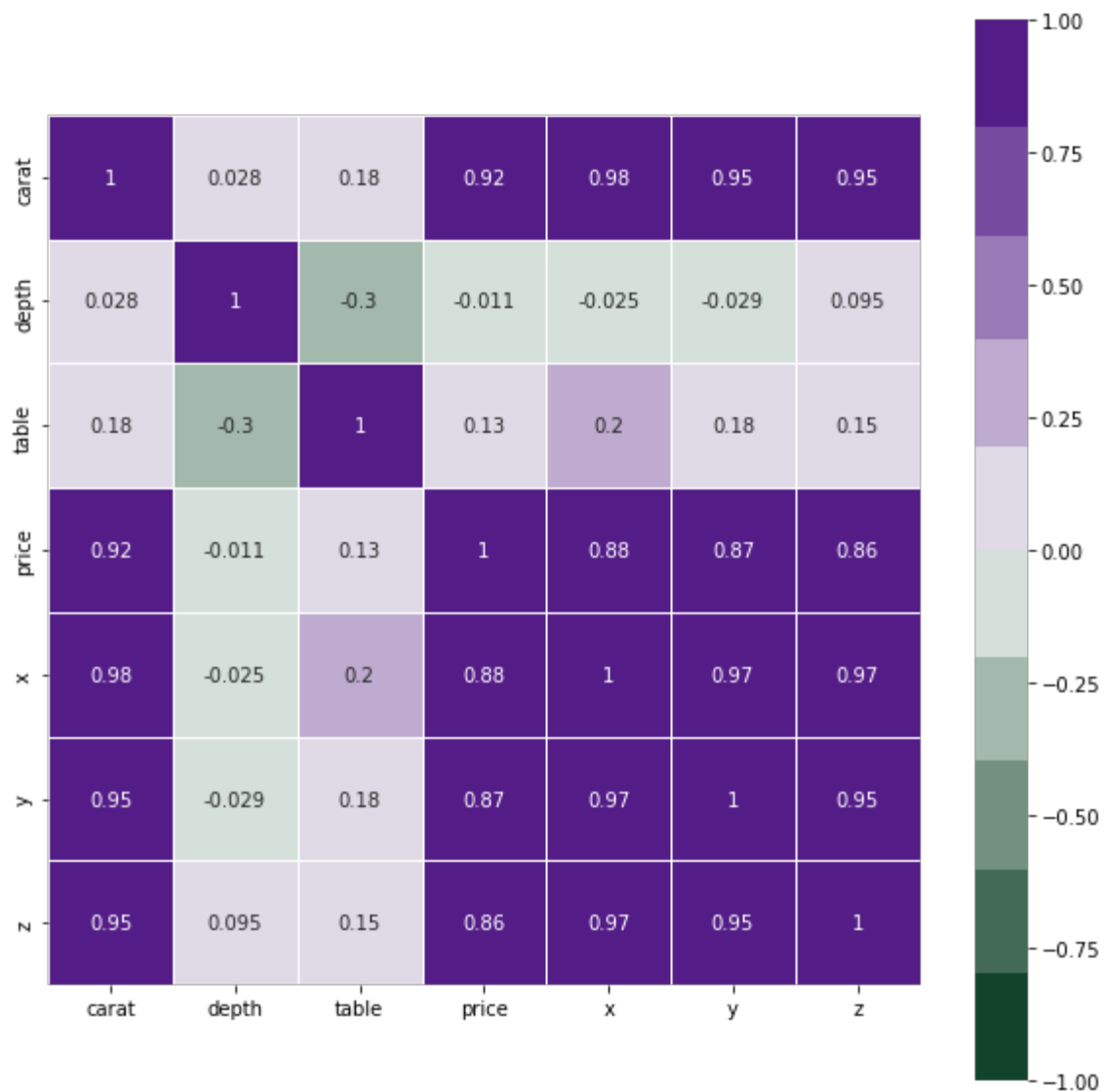
```
Out[ ]: embarked  C   Q   S
survived
0      75  47  427
1      93  30  217
```

```
In [ ]: df_plot = titanic_df.groupby(['embarked',
    'survived']).size().reset_index().pivot(columns='embarked',
    index='survived', values=0)
df_plot.plot(kind='bar', stacked=True);
```



### Correlation matrix/heatmap

```
In [ ]: # Establece los límites de colores entre [-1, 1], así como un rango de
    colores de oscuro a oscuro, pasando por claro (0 correlación)
plt.figure(figsize=(10,10))
sns.heatmap(diamonds_df.corr(),
            vmin=-1,
            vmax=1,
            center=0,
            cmap=sns.diverging_palette(145, 280, s=85, l=25, n=10),
            square=True,
            annot=True,
            linewidths=.5);
```



### Phik matrix/heatmap

Esta matriz es un muy buen indicador sobre la relación entre los datos, teniendo en cuenta variables categóricas.

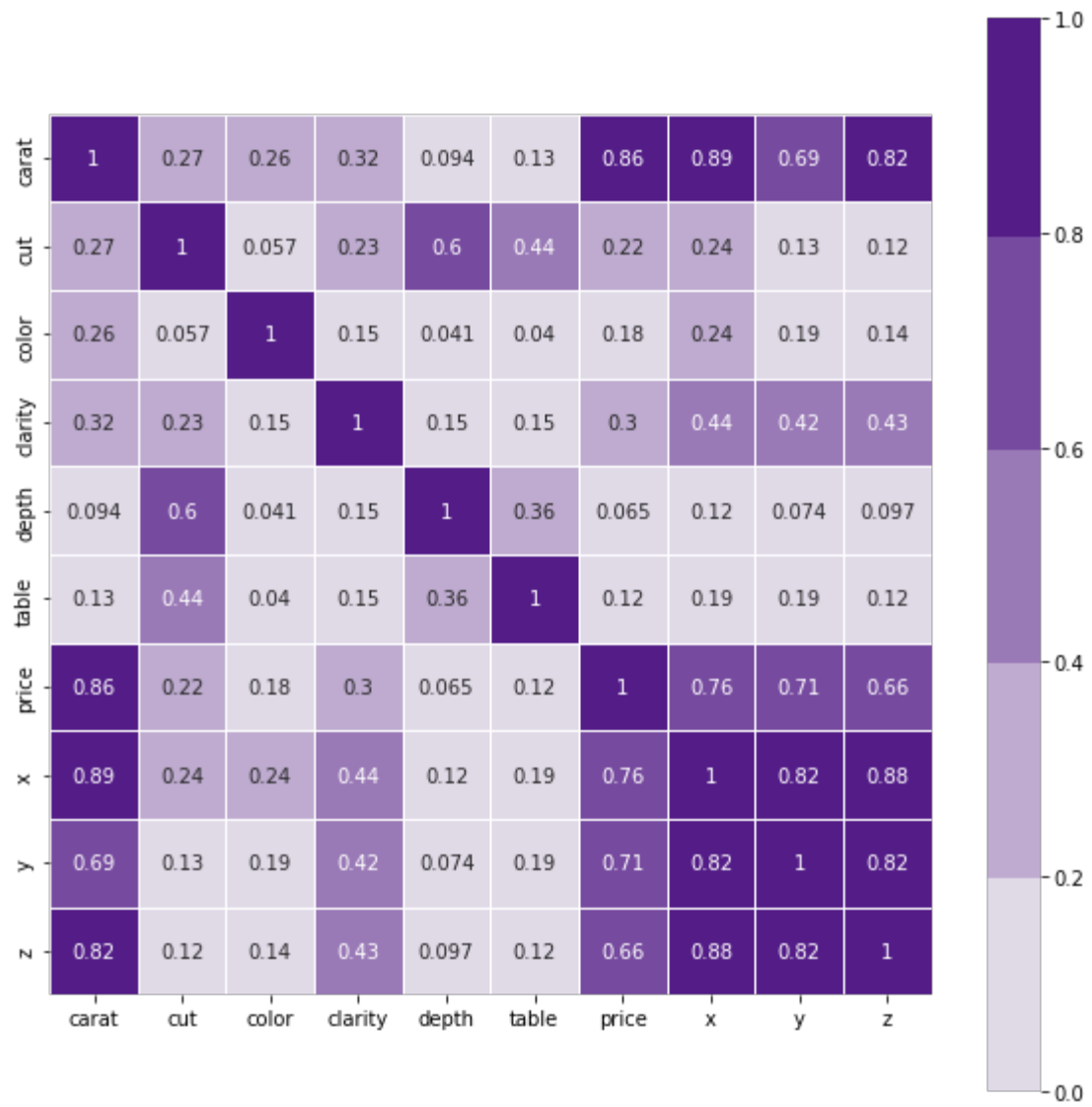
```
In [ ]: diamonds_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0   carat       53940 non-null  float64
 1   cut         53940 non-null  category
 2   color       53940 non-null  category
 3   clarity     53940 non-null  category
 4   depth       53940 non-null  float64
 5   table       53940 non-null  float64
 6   price       53940 non-null  int64
 7   x           53940 non-null  float64
 8   y           53940 non-null  float64
 9   z           53940 non-null  float64
dtypes: category(3), float64(6), int64(1)
memory usage: 3.0 MB
```

```
In [ ]: # !pip install phik
import phik
phik_matrix = diamonds_df.phik_matrix()

plt.figure(figsize=(10,10))
sns.heatmap(phik_matrix,
            vmin=0,
            vmax=1,
            center=0,
            cmap=sns.diverging_palette(145, 280, s=85, l=25, n=10),
            square=True,
            annot=True,
            linewidths=.5);
```

```
interval columns not set, guessing: ['carat', 'depth', 'table', 'price', 'x', 'y',
'z']
```



[Volver al índice](#)

## 9. Eliminación de features

Una vez hecho un primer análisis de nuestras variables, llega el momento de limpiar el dataset. Lo primero que nos planteamos es si sobran features, en función de toda la información ya recabada. Parecido al primer apartado de eliminación de features, pero en este caso no seremos tan restrictivos, ya que eliminaremos features con más información de la que disponíamos antes.

- **Features con valores constantes.**
- **Features con alto porcentaje de missings:** En general más de un 20%-30% de missings en una sola feature es bastante, por lo que podría considerarse la eliminación de la misma.
- **Features con identificadores.** Todo lo que sean ids y nombres tienen poco poder predictivo. Podría darse el caso de que un id corrale mucho con una fecha. Sería el caso de pedidos de una empresa, cuyo id va aumentando según pasan los días, pero igualmente suele aportar poco.
- **Features de strings largos:** Se podría aplicar algún tratamiento de texto con expresiones regulares o incluso NLP (Natural Language Processing), pero en general suelen eliminarse este tipo de features.
- **Features con alta cardinalidad:** viene a ser parecido a los identificadores. Se suelen eliminar.
- **Correlación o Phik matrix:** podríamos aplicar estos indicadores para eliminar algunas features. Recuerda que la matriz de correlción sólo te da información sobre la relación LINEAL entre los datos numéricos.

Estas decisiones no se toman tampoco a la ligera y dependerá mucho de la cantidad de features que tengamos disponible. Si es poca, andaremos con más cuidado a la hora de eliminar información, mientras que si son muchas las features no habrá problema en eliminar algunas tipo identificador o que contengan gran cantidad de missings.

*# Elimina features pero no sustituye*

```
df.drop(columns=['feature1', 'feature2'])
```

*# Elimina features y sustituye en el dataframe*

```
df.drop(columns=['feature1', 'feature2'], inplace=True)
```

```

In [ ]: # Con este código podrás eliminar de manera automática features con
        mucha cardinalidad o gran cantidad de missings

        # Max (%) cardinalidad que permitiremos en una feature
        cardi = 20

        # Max (%) de missings que permitiremos en una feature
        max_miss = 30

        def drop_cols(df_, max_cardi=20, max_miss=30):
            df = df_.copy()
            delete_col = []

            for i in df.columns:
                missings = df[i].isnull().sum() * 100 / len(df)

                # Elimina por missings
                if missings >= max_miss:
                    df.drop(i, 1, inplace=True)
                    continue

                # Elimina por cardinalidad en variables categoricas
                if df[i].dtype.name in ('category', 'object'):
                    if df[i].nunique()*100/len(df) >= max_cardi:
                        df.drop(i, 1, inplace=True)

            return df

        drop_cols(titanic_df).head()

```

```

Out[ ]:

```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	em
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	So
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	So
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	So
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	So

[Volver al índice](#)

## 10. Duplicados

Datos duplicados en general no aportan nada, a no ser que hayamos remuestreado el target, en el caso de tener un dataset desbalanceado ([ver apartado de análisis del target](#)). Por tanto, en general se suelen eliminar.

**¿Qué es un duplicado?** Pregunta obvia, pero en ocasiones no está tan claro. ¿Sabrías identificar cuál es tu clave única en los datos? Un identificador único puede ser los clientes, o los pedidos hechos en tu tienda, o tus pacientes. Pero si tu clave son tus clientes, podrías tener varios clientes "duplicados" porque dispones de datos concretos de ese cliente a fecha 14 y 15 de marzo. O tienes datos únicos de clientes, o tienes datos únicos por cliente y fecha, las dos cosas no puedes.

**¿Cómo elimino los duplicados?** Lo más fácil es un `drop_duplicates()` de pandas, que eliminará los registros duplicados teniendo en cuenta TODAS las columnas. Pero hay ocasiones en las que quiero eliminar registros duplicados en función de unas pocas columnas (ver ejemplo abajo).

**NOTA:** cuidado con eliminar datos en test si estamos en una competición de Kaggle. A la muestra de test no le podemos quitar registros

```
In [ ]: df1 = pd.DataFrame({'Clientes': ['Alba', 'Carolina', 'Alberto',  
                                     'Alberto'],  
                          'Fecha': ['2020-01', '2020-01', '2020-01', '2020-02'],  
                          '¿Compró?': [1, 1, 1, 1]})  
  
df1
```

```
Out[ ]:
```

	Clientes	Fecha	¿Compró?
0	Alba	2020-01	1
1	Carolina	2020-01	1
2	Alberto	2020-01	1
3	Alberto	2020-02	1

Alberto en principio está duplicado. Pero si aplico un `drop_duplicates` no me lo elimina ya que TODAS las filas son únicas. Para ello habrá que decirle a pandas qué columna/as queremos que utilice para discriminar si un registro es un duplicado o no. En este caso será únicamente el nombre. [Aquí tienes la documentación del `drop\_duplicates\(\)`](#)

```
In [ ]: print("len original", len(df1))

print("len drop_duplicates():", len(df1.drop_duplicates()))

de_new = df1.drop_duplicates(subset = 'Clientes', keep = 'last')
print("len drop_duplicates() por cliente:", len(de_new))

len original 4
len drop_duplicates(): 4
len drop_duplicates() por cliente: 3
```

```
In [ ]: de_new
```

```
Out[ ]:
```

	Clientes	Fecha	¿Compró?
0	Alba	2020-01	1
1	Carolina	2020-01	1
3	Alberto	2020-02	1



[Volver al índice](#)

## 11. Missings

Suponen un problema ya que la mayoría de los modelos no saben tratar los missings y por tanto tendremos que inferir sus valores. Por aclarar, un missing no es un 0, ni un string vacío, ni un False. Es un hueco en los datos, el vacío, nos falta un dato en una de las features. Eso es un missing.

¿Por qué aparecen missings?

- **Missings en la extracción de los datos:** estos se suelen identificar rápido, por ejemplo, que no se ha leído bien un CSV, y todos los valores se los tome como uno solo.
- **Missings completamente aleatorios:** la probabilidad de que haya un valor missing es la misma para todas las observaciones.
- **Missings aleatorios:** aleatoriamente hay missings en una columna. Por ejemplo, datos de edades de una página web, vemos que hay más missings de mujeres que de hombres.
- **Missings que dependen de los inputs:** Por ejemplo, en un estudio médico, si un diagnóstico es dudoso, hay una alta probabilidad de ser descartado. Cuando estos missings se les pone a "Dudoso", dejan de ser valores aleatorios.
- **Missings que dependen de su propio valor:** por ejemplo, ingresos de las personas, los que tienen mucho o muy poco, se tenderá a no poner el valor.
- **Missings en tratamiento de datos:** pueden aparecer missings debido a una incorrecta lectura o escritura, tratamiento de los datos y merge con otras tablas.

Métodos para tratar los missings

- **Imputación sobre caso similar:** tengo en cuenta otra variable, por ejemplo si tengo la altura de alumnos de una clase, podría calcular los missings teniendo en cuenta si son chicos o chicas. O si hay missings de densidad de población, pero tengo la población y el área, podría recalcularlo. Esta opción es lo ideal, razonando los missings con los propios datos.
- **Borrado:** Aquí hay dos opciones: si hay un missing en una columna, nos cargamos toda la fila. El borrado se realiza cuando tenemos pocos missings y muchas observaciones. De lo contrario, eliminaríamos muchos datos.
- **Imputación por valor concreto:** No es lo más habitual, pero puede ocurrir que sepamos de antemano por qué hay missings en esa feature. Quizá todos los días hay problemas en la carga de la base de datos con el carácter Ñ, y cuando debería aparecer España, en realidad está en missing. Este método también se utiliza cuando hay missings en la variable categórica, le ponemos un nombre genérico y equivaldría a añadir una categoría nueva a la feature.
- **Imputación de Media/Mediana/Moda:** es el método más frecuente, sobretodo si

tenemos poco tiempo. Para variables numéricas imputamos normalmente la media, aunque si la variable tiene asimetría, suele dar mejores resultados la mediana. En el caso de que sea una variable categórica lo más habitual es utilizar la moda.

- **Imputacion + flag:** es igual que el caso anterior, pero además añadimos una nueva columna binaria por cada feature con missings imputados para indicarle al modelo que en ese lugar había un missing.
- **Modelo:** es lo más sofisticado. Dividimos el dataset en dos: uno con missings y el otro sin, que se usa para entrenar al modelo. Este método se suele comportar bien, a no ser que no tengan mucha relación los datos. Lo habitual es usar KNN, regresión logística o lineal. El KNN rellena los missings teniendo en cuenta el resto de variables del dataset, buscando similitudes con registros que tengan las variables parecidas. Se puede usar para cualitativas y cuantitativas, las variables con muchos missings se pueden tratar fácilmente. Lo malo es que es muy lento y la elección del k-value es crítica.

Vamos al código para imputar missings. Lo primero que necesitas saber es los missings que tienes en los datos. Para ello lo mejor es utilizar la función que hemos creado en el apartado de comprensión de variables

```
In [ ]: data_report(titanic_df)
```

```
Out[ ]:
```

	COL_N	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who
<b>DATA_TYPE</b>		int64	int64	object	float64	int64	int64	float64	object	category	object
<b>MISSINGS (%)</b>		0.0	0.0	0.0	19.87	0.0	0.0	0.0	0.22	0.0	0.0
<b>UNIQUE_VALUES</b>		2	3	2	88	7	7	248	3	3	
<b>CARDIN (%)</b>		0.22	0.34	0.22	9.88	0.79	0.79	27.83	0.34	0.34	0.3

```
In [ ]: # En los ejemplos con drop NO estamos sobrescribiendo el dataset

# Para eliminar una columna, si tiene muchos missings
iris_df.drop(columns = ['sepal length (cm)'])

# Eliminamos las filas si encuentra missing en cualquier columna del
dataset
iris_df.dropna()

# Elimina las filas donde todos sus elementos sean missing
iris_df.dropna(how='all')

# Elimina filas si encuentra missings en las siguientes columnas
iris_df.dropna(subset=['sepal length (cm)', 'sepal width (cm)'])

# Imputar con un valor
iris_df['sepal length (cm)'] = iris_df['sepal length (cm)'].fillna(0)

# Imputar con la media o mediana
iris_df['sepal length (cm)'] = iris_df['sepal length
(cm)'].fillna(iris_df['sepal length (cm)'].mean())
iris_df['sepal length (cm)'] = iris_df['sepal length
(cm)'].fillna(iris_df['sepal length (cm)'].median())

# Interpolación. Imputa missings en función del valor anterior y el
siguiente. Tendrá sentido en series temporales, es decir, con datos
ordenados
iris_df['sepal length (cm)'] = iris_df['sepal length
(cm)'].interpolate()
```

```
In [ ]: # Imputacion mediante KNN
        from sklearn.impute import KNNImputer

        imputer = KNNImputer(n_neighbors=2)
        imputer.fit(titanic_df[['pclass', 'age', 'sibsp', 'parch', 'fare']])
        titanic_df[['pclass', 'age', 'sibsp', 'parch', 'fare']] =
        imputer.transform(titanic_df[['pclass', 'age', 'sibsp', 'parch',
        'fare']])
```

```
In [ ]: data_report(titanic_df)
```

```
-----
NameError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16640\1332495200.py in <module>
----> 1 data_report(titanic_df)
```

```
NameError: name 'data_report' is not defined
```

[Volver al índice](#)

## 12. Anomalías y errores

También podemos encontrarnos casos extraños en los datos que no siempre se corresponden con un outlier. Un outlier es un valor atípico en una variable, pero que puede tener cierto sentido. Sin embargo, un error es un valor que no tiene nada que ver con la variable, como que aparezcan edades en negativo, o haya ciertos datos a 99999. También podría ocurrirnos en texto, un texto mal decodificado, o unas fechas erróneas. Posibles errores que habría que corregir:

1. **Numeros positivos que aparecen como negativos:** variables numéricas que siempre deberían tener numeros positivos y en algún caso (o en muchos) presente datos negativos, habrá que tratarlo como si fuesen missings.
2. **Datos de fecha incorrectos:** asegúrate que estás leyendo bien la fecha en el formato americano o europeo. Suele cambiarse la posición del día por la del mes. Comprueba que los años de las fechas son coherentes, ya que una lectura errónea puede llevar a fechas del año 9999. El método que no te va a fallar (pero también es el más costoso) es importar la fecha en string, y luego parsear cada uno de sus elementos mediante la función `split()`.
3. **Encoding del texto:** si ves caracteres raros, sospecha del encoding del texto. Abre el archivo de los datos en texto plano y comprueba que no tenga caracteres raros. Prueba varios encodings en la lectura. Los más habituales: `ascii`, `ansi`, `utf-8`, `latin1`. Tienes más información en [el apartado de carga de datos](#)

[Volver al índice](#)

## 13. Outliers

### ¿Qué es un outlier?

Se trata de un valor atípico dentro de nuestros datos. Un valor que se desvía mucho de las métricas estadísticas de centralidad (media, moda, mediana). En el análisis exploratorio de datos suele ser algo a estudiar, el por qué tengo algunos valores atípicos en los datos. No obstante, en machine learning los outliers en los datos implican penalizaciones en los modelos, sobretodo los que trabajan con distancias y con Gradient Descent. Por tanto, hay que lidiar con ellos.

### ¿Cómo detecto los outliers?

Lo mejor es gráficamente, aunque también puedes parametrizar una serie de umbrales en tus features, a partir de los cuales consideras que son outliers. Hay dos maneras de verlos gráficamente, mediante análisis:

1. **Univariante**
2. **Bivariante**
3. **Medidas estadísticas**: cuánto me desvío de la media, o cuánto de lejos estoy del IQR.

### ¿Qué gráficas utilizo para visualizar los outliers?

Lo mejor son boxplots, histogramas, diagramas de densidad, scatter plots y count plots para categóricas.

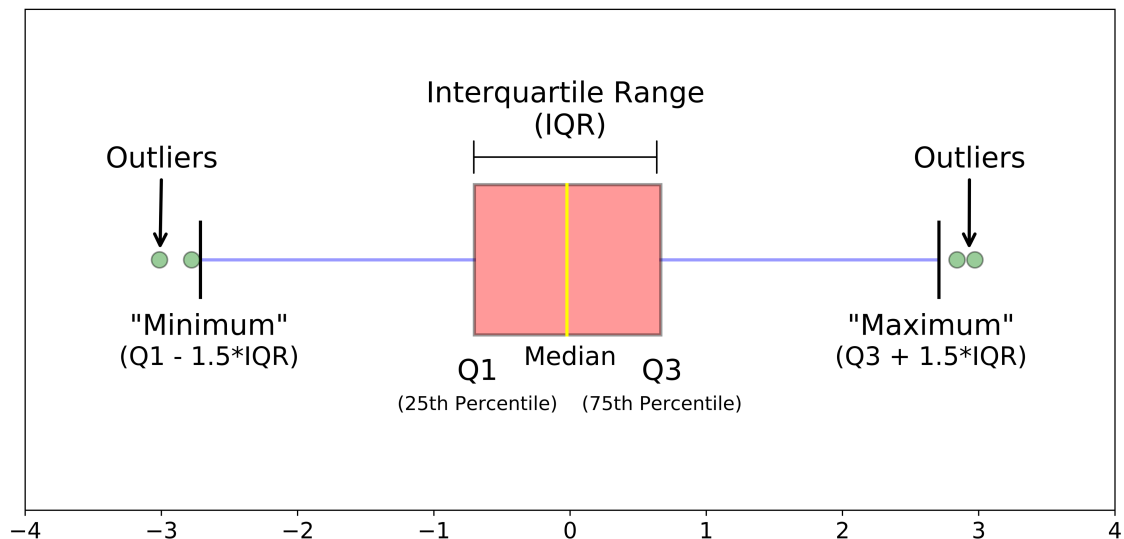
### ¿Qué técnicas hay para detectar outliers?

1. **Gráficamente**: Datos que se desvien mucho.
2. **Cuartiles**: se ve en un diagrama de caja. Datos que caigan fuera del rango  $\pm 1.5 \cdot \text{IQR}$ . Este 1.5 es muy restrictivo por lo que se suelen probar valores del 3 al 5. Dependerá mucho de cada feature.
3. **Desviación estándar**: todo lo que caiga fuera de  $(\text{media} \pm N \cdot \text{std})$  de la variable. Normalmente N es un valor de 3 a 5.

Ten en cuenta siempre el contexto de negocio para considerar los outliers, no el puramente numérico. El propio negocio del dataset que estés analizando es el que define qué es un outlier y qué no.

Detectar gráficamente es la técnica más habitual, pero si quieres hacerlo de una manera más automatizada puedes probar con las otras dos opciones. Además ten en cuenta que **los outliers siempre dependen del negocio** por lo que deberás analizarlos bien antes de eliminarlos

eliminarlos.



### ► Causas de los outliers

¿Qué hago con los outliers?

Tenemos varias opciones:

1. **Eliminarlos:** es la técnica más habitual y sencilla
2. **No hacer nada:** si no son exagerados. Los árboles de decisión y SVM (en este orden) son robustos frente a outliers.
3. **Transformaciones logarítmicas:** elimina asimetría en las features, y por tanto reduce el efecto de los outliers. Para más info [ver el apartado de transformaciones](#).
4. **Binning:** discretiza la variable en varios grupos. Esto me va a permitir incluir los outliers en un grupo donde haya otros datos no considerados como outliers (ver ejemplo abajo)
5. **Imputación:** igual que con los missings, sustituir los valores. Esto tendrá sentido si los outliers están bien analizados, y desde el punto de vista de negocio conviene sustituirlos por un valor concreto.
6. **Tratamiento por separado:** si es un número significativo de outliers quizá merezca la pena separar los datos y tratarlos como otro modelo aparte.

¿Qué modelos son más sensibles a outliers?

Los modelos a los que MENOS afectan los outliers son los árboles de decisión, por lo que todo lo que sean árboles y derivados tendrán un comportamiento robusto frente a outliers. Los SVM también funcionan bien. A los algoritmos que más les afectan los outliers son a los basados en distancia (KNN) o Gradient Descent (regresiones).

**NOTA** Mucho cuidado al eliminar outliers en  $X_{train}$ . Hay que eliminar los mismos registro en el target ( $y_{train}$ ). Para ello lo mejor es juntarlos, eliminarlos y cuando vayas a modelar separar  $X$  de  $y$ .

```
In [ ]: # Target de precio en el dataset de diamonds
```

```
# Target de precio en el dataset de diamantes
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
plt.subplots_adjust(hspace = 0.3)

# Outliers con boxplot
sns.boxplot(diamonds_df['depth'], ax=axes[0, 0])
axes[0, 0].set_title("Outliers con boxplot")

# Feature susceptible de transformación
sns.boxplot(diamonds_df['price'], ax=axes[0, 1])
axes[0, 1].set_title("Feature susceptible de transformación")

# Transformacion Logaritmica
```

```
In [ ]: from scipy.stats import iqr

def outliers_quantile(df, feature, param=1.5):

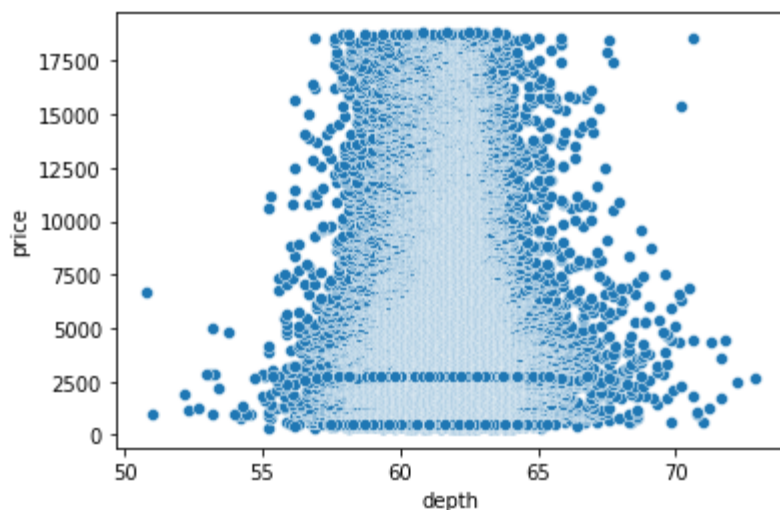
    iqr_ = iqr(df[feature], nan_policy='omit')
    q1 = np.nanpercentile(df[feature], 25)
    q3 = np.nanpercentile(df[feature], 75)

    th1 = q1 - iqr_*param
    th2 = q3 + iqr_*param

    return df[(df[feature] >= th1) & (df[feature] <=
th2)].reset_index(drop=True)

diamonds_df2 = outliers_quantile(diamonds_df, 'depth', 7)
print("Len original:", len(diamonds_df))
print("Len sin outliers en depth:", len(diamonds_df2))
sns.scatterplot(data=diamonds_df2, x='depth', y='price');
```

```
Len original: 53940
Len sin outliers en depth: 53933
```

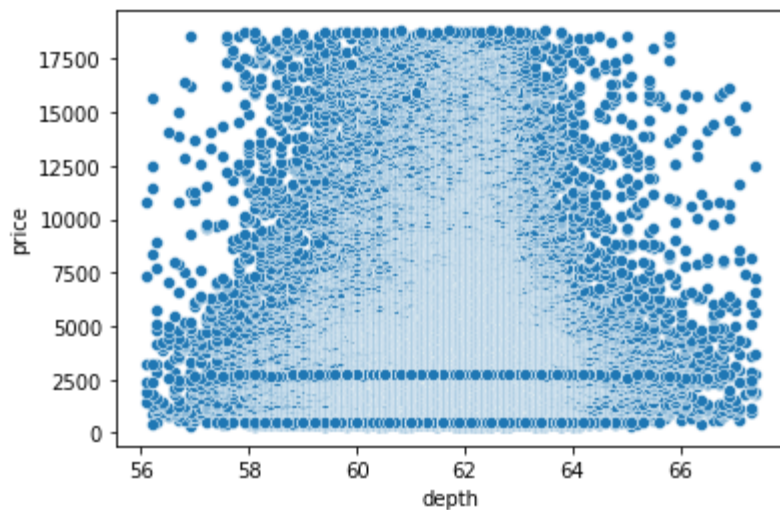


```
In [ ]: def outlier_meanSd(df, feature, param=3):  
    media = df[feature].mean()  
    desEst = df[feature].std()  
  
    th1 = media - desEst*param  
    th2 = media + desEst*param  
  
    return df[((df[feature] >= th1) & (df[feature] <= th2)) |  
              (df[feature].isnull())].reset_index(drop=True)  
  
diamonds_df2 = outlier_meanSd(diamonds_df, 'depth', 4)  
print("Len original:", len(diamonds_df))  
print("Len sin outliers en depth:", len(diamonds_df2))  
sns.scatterplot(data=diamonds_df2, x='depth', y='price');
```

Len original: 53940

Len sin outliers en depth: 53739

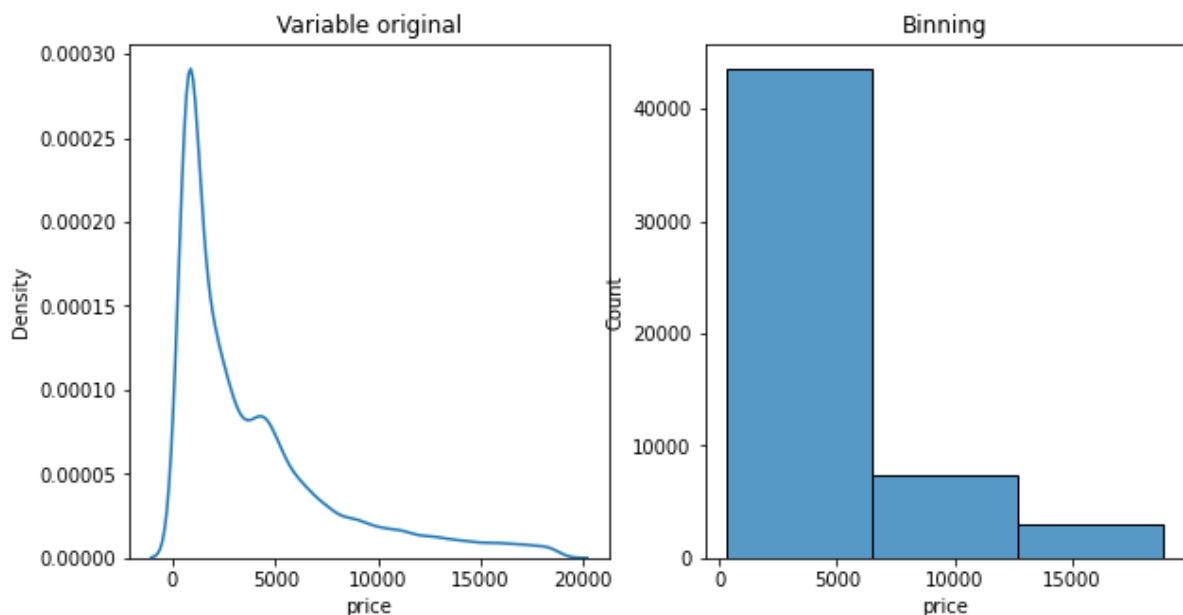




```
In [ ]: # Tecnicas de binning
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
plt.subplots_adjust(hspace = 0.3)

sns.kdeplot(diamonds_df["price"], ax=axes[0])
axes[0].set_title("Variable original")

sns.histplot(diamonds_df["price"], bins=3, ax=axes[1])
axes[1].set_title("Binning");
```



```
In [ ]: hist, bin_edges = np.histogram(diamonds_df["price"], bins=3,
density=True)
bin_edges
```

```
Out[ ]: array([ 326.          , 6491.66666667, 12657.33333333, 18823.          ])
```

[Volver al índice](#)

## 14. Feature Engineering

### 14.1 Transformaciones

Como ya se ha mencionado varias veces a lo largo del notebook, lo ideal es tener distribuciones normales en todas las variables (target incluido) de un modelo. Por desgracia no siempre es así, y variables como el salario de una población suelen tener una larga cola hacia la derecha, es decir, presentan asimetría positiva.

¿Cómo solucionamos esto?

Aplicando transformaciones logarítmicas, cuadradas o cúbicas. Las logarítmicas suelen dar mejor resultado.

Para comprobar si tenemos una distribución normal, lo mejor es graficarla, pero siempre tenemos la opción de calcular un valor numérico que nos indique si la distribución es normal o no. Para ello podemos aplicar el test de Shapiro, cuya hipótesis nula es que la variable sigue una distribución normal, mientras que la alternativa sostiene que no sigue una distribución normal. Si el p-value es mucho menor que su nivel de significación (0.05), tendremos que rechazar la hipótesis nula y asegurar que la variable NO sigue una distribución normal.

Otra forma de visualizar la normalidad de una variable es mediante un Q-Q plot

#### ► Q-Q plot

```
In [ ]: from scipy.stats import shapiro
        from sklearn.datasets import load_boston
        from scipy.stats import skew

        boston_target = load_boston()["target"]

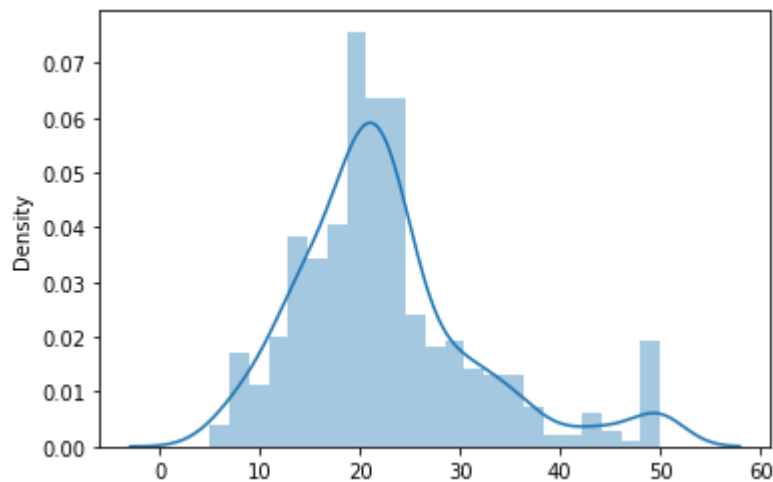
        # Muy por debajo del nivel de significación (0.05) no se considera
        # distribución normal.
        print("Saphiro:", shapiro(boston_target).pvalue)

        # Para comprobar la asimetría de una variable siempre podemos calcular
        # su valor skew
        # 0 si es simetrica, >0 cola hacia la derecha, <0 cola hacia la
        # izquierda
        print("Asimetria:", skew(boston_target))
```

```
Saphiro: 4.940195786292171e-16
Asimetria: 1.104810822864635
```

```
In [ ]: sns.distplot(boston_target)
```

```
Out[ ]: <AxesSubplot:ylabel='Density'>
```



```
In [ ]: import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

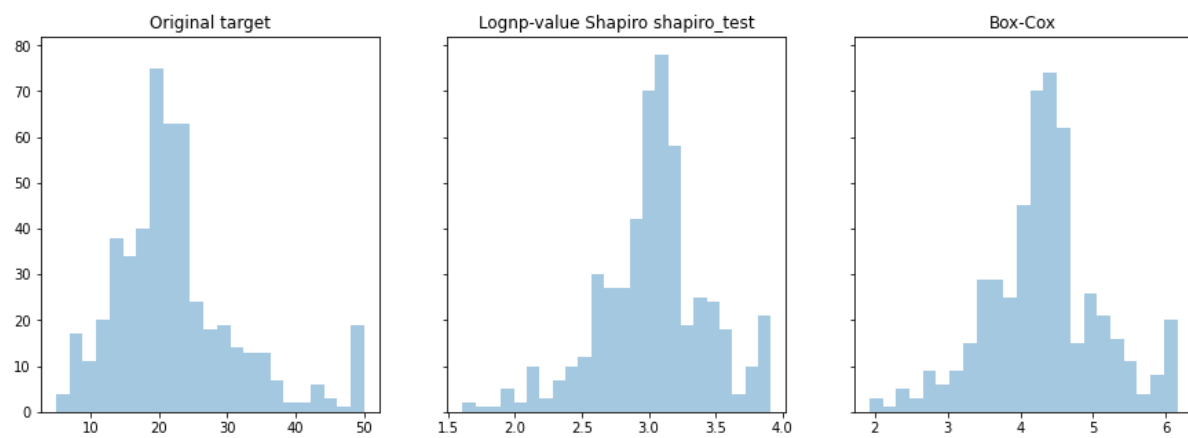
fig, axes = plt.subplots(1, 3, figsize=(15, 5), sharey=True)

# Original target
print("p-value Shapiro test Original: ", shapiro(boston_target).pvalue)
sns.distplot(boston_target, kde=False, ax=axes[0])
axes[0].set_title("Original target")
print("Asimetria:", skew(boston_target))

# Logaritmica
print("p-value Shapiro test Logaritmica: ",
      shapiro(np.log(boston_target)).pvalue)
sns.distplot(np.log(boston_target), kde=False, ax=axes[1])
axes[1].set_title("Log" + 'np-value Shapiro ' + "shapiro_test")
print("Asimetria:", skew(np.log(boston_target)))

# Box-cox
print("p-value Shapiro test Box-cox: ",
      shapiro(stats.boxcox(boston_target)[0]).pvalue)
sns.distplot(stats.boxcox(boston_target)[0], kde=False, ax=axes[2])
axes[2].set_title("Box-Cox");
print("Asimetria:", skew(stats.boxcox(boston_target)[0]))
```

```
p-value Shapiro test Original: 4.940195786292171e-16
Asimetria: 1.104810822864635
p-value Shapiro test Logaritmica: 1.9355394442754914e-07
Asimetria: -0.32934127453151935
p-value Shapiro test Box-cox: 2.0399854747665813e-06
Asimetria: 0.01583478691187103
```



## 14.2 Encodings

¿En qué dataset no vamos a encontrar variables cualitativas? Variables categóricas que se componen de un conjunto de valores finito, pequeño, y normalmente en formato string. Aportan muchísima información, pero necesitamos transformarlas a valores numéricos para que los modelos puedan trabajar con ellas, ya que no admiten texto.

Una primera aproximación podría ser asignar un número del 1 a  $n$  a cada categoría, siendo  $n$  la cantidad de valores únicos de esa variable. Esto no es del todo correcto ya que si tuviésemos países en una variable y asignamos 1 a China, 2 a Marruecos y 3 a Brasil, estamos diciendo que China está a 2 de distancia de Brasil y que Marruecos es un país a medio camino entre China y Brasil. Estamos cometiendo un error y es que le estamos asignando un orden a variables que originalmente no tienen orden.

Por tanto, estas son las preguntas que deberías plantearte cuando vayas a codificar variables categóricas:

**¿La variable es binaria?** En este caso nos dará igual el orden. Asignamos 1 y 0 indistintamente y listo.

**¿La variable está ordenada?** Si estuviese ordenada, nuestra primera aproximación sería perfectamente válida. Eso sí, aplicando el mapeo correcto. Si tenemos calificaciones tipo ['Baja', 'Media', 'Alta', 'Muy Alta'], asegúrate de codificarlas en el orden correcto: [0, 1, 2, 3].

**¿Y si la variable no está ordenada?** Aplicaremos un dummy encoder o OneHotEncoder. Básicamente consiste en crear una columna nueva con cada feature. Todas las columnas a 0, excepto cuando aparezca esa categoría concreta. En ese caso pondremos un 1.

### ► OneHotEncoder

Para más detalle sobre la diferencia entre un OneHot o dummy encoder visita [este enlace](#)

**¿Y si tenemos una gran cardinalidad en la variable?** No es muy viable aplicar un OneHot o dummy encoder, ya que crearía demasiadas features nuevas. Otra posible opción es emplear un HashingEncoder. Establecemos un número  $m$  de columnas, y traduce mediante una función hash todas las categorías de la feature en solo las  $m$  columnas, por supuesto ya no van a ser binarias. En [este enlace](#) tienes información sobre cómo funciona.

**Consejos para trabajar con train y test.** Recuerda no contaminar train con los datos de test. Esto quiere decir que si usas un OneHotEncoder sobre una categoría de colores, donde en test hay colores que no tienes en train, NO crees columnas dummy de esos colores en train porque en teoría train no sabe que existen esos colores de test. Crea los dummies en train con los datos de train, y utiliza la misma codificación para test. Si alguna de las categorías de train no coincide con las de test y viceversa, simplemente aparecerá todo el registro a 0s.

Método	Descripción
--------	-------------

DummyEncoding/OneHotEncoder	Sustituye la variable por 1s y 0s en varias columnas Por sencillez, se recomienda usar la función de dummies de pandas, en vez del OneHotEncoder de sklearn
LabelEncoding	Sustituye cada categoría por un valor numérico creciente
Count encoding	Sustituye cada categoría por la frecuencia de aparición de la misma
Mean encoding	Para problemas de regresión. Sustituye cada categoría por la media del target para esa categoría
Hashing	Utilizado cuando tenemos una alta cardinalidad en los datos

Los métodos mean encoding o el count encoding no son tan efectivos como el resto.

```
In [ ]: import pandas as pd
from sklearn.preprocessing import OneHotEncoder

df = pd.DataFrame({'City': ['SF', 'SF', 'NYC', 'NYC', 'Seattle',
                             'Seattle'],
                   'Rent': [3999, 4000, 3499, 3500, 2499, 2500]})

dummy_df = pd.get_dummies(df, prefix=['city'])
dummy_df
```

```
Out[ ]:
```

	Rent	city_NYC	city_SF	city_Seattle
0	3999	0	1	0
1	4000	0	1	0
2	3499	1	0	0
3	3500	1	0	0
4	2499	0	0	1
5	2500	0	0	1

```
In [ ]: # Si queremos aplicar un mapeo personalizado
# Para encodig binario se recomienda aplicar un mapeo, para no
equivocarnos.
def mapping(x):
    if x == 'SF':
        return 1
    elif x == 'NYC':
        return 2
    elif x == 'Seattle':
        return 3
    else:
        return 9999

df['Custom map'] = df['City'].apply(mapping)
df
```

```
In [ ]: # Si tenemos una alta cardinalidad en los datos, utilizamos un hasher
from sklearn.feature_extraction import FeatureHasher

df = pd.DataFrame({'CA': ['Madrid', 'Cataluña', 'Andalucia', 'Pais
Vasco', 'Andalucia', 'Madrid', 'Valencia', 'Andalucia']})
df
```

```
Out[ ]:
```

	CA
0	Madrid
1	Cataluña
2	Andalucia
3	Pais Vasco
4	Andalucia
5	Madrid
6	Valencia
7	Andalucia

```
In [ ]: h = FeatureHasher(n_features=3, input_type='string')
h
```

```
Out[ ]: FeatureHasher(input_type='string', n_features=3)
```



```
In [ ]: f = h.transform(df['CA'])  
        f.toarray()
```

```
Out[ ]: array([[ -1.,  4.,  1.],  
               [ -2.,  2.,  4.],  
               [ -2.,  3.,  2.],  
               [ -2.,  1.,  1.],  
               [ -2.,  3.,  2.],  
               [ -1.,  4.,  1.],  
               [ -1.,  1.,  2.],  
               [ -2.,  3.,  2.]])
```

## 14.3 Nuevas features

Llega la parte más creativa de la analítica, en la que intentamos sacarle jugo a los datos. El objetivo de este apartado es crear nuevas features a partir de las que ya tenemos. Muy habitual cuando no tenemos muchas features y hay que buscar nuevas.

Algunas técnicas que podemos aplicar:

1. **Si tenemos fechas y horas**, podemos obtener nuevos datos como:
  - Año, trimestre, mes, día, hora, minuto, segundo.
  - Un indicador de si es de día o de noche. O incluso si es por la mañana, horario laboral...
  - Otro indicador de si es un día de fin de semana o laboral.
  - Decadas o épocas.
2. **Concatenar campos**: si tenemos variables categóricas, puede resultar interesante realizar otras agrupaciones con otras categóricas. Por ejemplo, si tenemos ventas por cliente, segmentadas por sexo y tipo de producto. Una nueva feature puede ser combinar ambos, para tener un indicador agregado de tipo de producto y sexo.
3. **Agrupar**: sería el binning que ya hemos visto. Por ejemplo agrupar edades en varios bins.
4. **Suma/Resta/Multiplicación...**: los ratios suelen ser útiles. Cuidado con las combinaciones lineales entre variables, sobretodo si estamos con regresión lineal.
5. **Agregaciones de otros datasets**: por ejemplo, si estamos viendo pedidos de clientes, sus datos agregados de lo que suelen gastar, o la media de pedidos que hacen al mes, se podrían añadir al dataset.
6. **Datos espaciales**: si tuviésemos varias localizaciones geográficas, podríamos calcular por ejemplo distancias.
7. **Texto**: podemos sacar info de ciertas palabras clave. Por ejemplo, si tenemos nombres con Mr, Mis, Mrs... es posible extraer la variable sexo de manera sencilla, empleando únicamente el nombre.
8. **Datos externos macro**: si estamos trabajando con países o poblaciones, podemos agregar nuevos datos al dataset como el PIB, cantidad de población, distribución por sexo...acudiendo a fuentes como el INE.

## 14.4 Escalados

Dependiendo del modelo que vayamos a utilizar, resultará útil escalar las features. **¿Qué significa escalar una variable?** Consiste en cambiar el orden de magnitud de la variable, conservando su distribución. Estas transformaciones se aplican por separado a cada feature. Se suele calcular unas métricas como media, desviación estándar, y mediante una fórmula aplicamos la transformación a cada valor de la feature.

¿Por qué realizamos transformaciones?

Hay ciertas características que le vienen bien a los modelos. Tener todos los datos en la misma escala es una de ellas. Los modelos trabajan mejor y entrenan más rápido cuando la escala de todos los datos es la misma. Esto lo vamos a conseguir con un `StandardScaler` o un `MinMaxScaler`. Otra de las características que deberían tener los modelos es tener datos con distribuciones gaussianas. Esto no siempre es así ya que muchas veces las distribuciones son bimodales o asimétricas. Para solucionar esto aplicamos logaritmos, elevamos al cuadrado o al cubo (veremos en el siguiente apartado).

`StandardScaler`

► **Explicación del `StandardScaler`**

```
In [ ]: from sklearn import datasets
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Cargamos datos
iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.20, random_state=42)
print(X_train[0])

# Creo el scaler con los datos de train
scal = StandardScaler() # Declaro el scaler
scal.fit(X_train) # Lo "entreno". Calculo su media y std para cada
feature
X_train = scal.transform(X_train) # Aplico el scaler y sobrescribo los
datos de train
print(X_train[0])

# Aplico el mismo scaler con los datos de test
X_test = scal.transform(X_test)

# Si quiero recuperar la anterior escala
X_train = scal.inverse_transform(X_train)
print(X_train[0])

[4.6 3.6 1.  0.2]
[-1.47393679  1.20365799 -1.56253475 -1.31260282]
[4.6 3.6 1.  0.2]
```

## MinMaxScaler

### ► Explicación del MinMaxScaler

```
In [ ]: from sklearn.preprocessing import MinMaxScaler, StandardScaler
iris = datasets.load_iris()
X = iris.data
y = iris.target

minmax = MinMaxScaler()
minmax.fit(X)
X_minmax = minmax.transform(X)

stdscaler = StandardScaler()
stdscaler.fit(X)
X_std = stdscaler.transform(X)
```

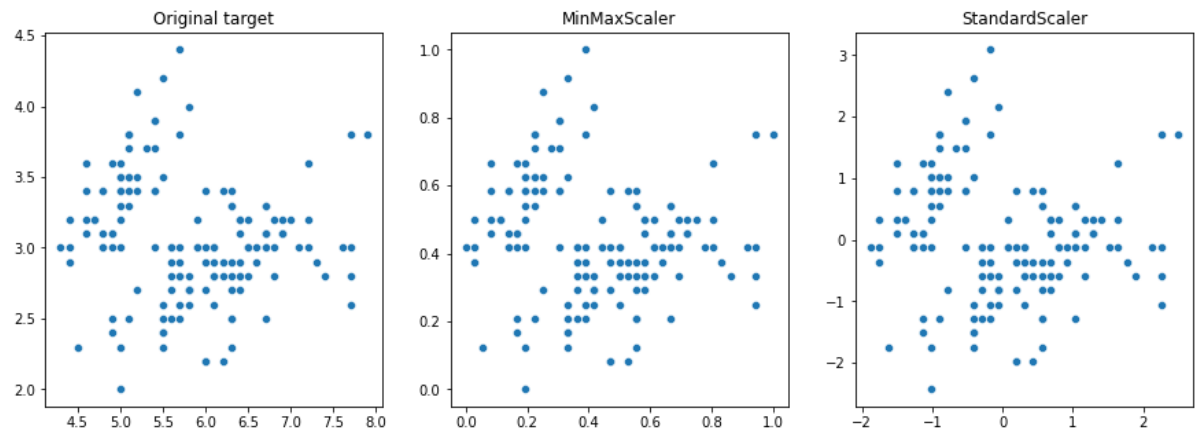
```
In [ ]: # La forma de las distribuciones no cambia.
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Original target
sns.scatterplot(X[:, 0], X[:, 1], ax=axes[0])
axes[0].set_title("Original target")

# MinMaxScaler
sns.scatterplot(X_minmax[:, 0], X_minmax[:, 1], ax=axes[1])
axes[1].set_title("MinMaxScaler")

# StandardScaler
sns.scatterplot(X_std[:, 0], X_std[:, 1], ax=axes[2])
axes[2].set_title("StandardScaler");
```



[Volver al índice](#)

## 15. Feature Reduction

[Ver apartado 6](#)

[Volver al índice](#)

## 16. Escoger métrica del modelo

Por desgracia los modelos no son perfectos y siempre cometen cierto error. De no ser así, probablemente no sería necesario utilizar Machine Learning para solucionar el problema y tengamos que acudir a soluciones más sencillas.

Por tanto, tendremos que definir una métrica para evaluar el comportamiento del modelo por dos motivos:

1. Primero, para probar varios modelos y poder **comparar** entre ellos cuál es el mejor
2. Segundo, para presentar **resultados del modelo**.

Prácticamente todas las métricas tratadas en este notebook están implementadas en `sklearn`, salvo MAPE. Consulta [la documentación](#) para más información.

### 16.1 Métricas para clasificación

#### Accuracy

La métrica más habitual. Calcula el % de acierto teniendo en cuenta todas las clases del algoritmo de clasificación. Esta métrica es muy fácil de entender, pero no profundiza en el % de acierto de cada clase, que puede ser algo interesante, dependiendo del problema que queramos tratar.

$$\text{Accuracy} = (\text{TP} + \text{TN}) / \text{Total}$$

#### Matriz de confusión

Confusión o *error matrix* es una tabla que describe el rendimiento de un modelo supervisado de Machine Learning en los datos de test, donde se desconocen los verdaderos valores. Se llama “matriz de confusión” porque hace que sea fácil detectar dónde el sistema está confundiendo dos clases.

- **True Positives (TP)**: cuando la clase real del punto de datos era 1 (Verdadero) y la predicha es también 1 (Verdadero)
- **Verdaderos Negativos (TN)**: cuando la clase real del punto de datos fue 0 (Falso) y el pronosticado también es 0 (Falso).
- **False Positives (FP)**: cuando la clase real del punto de datos era 0 (Falso) y el pronosticado es 1 (True).
- **False Negatives (FN)**: Cuando la clase real del punto de datos era 1 (Verdadero) y el valor predicho es 0 (Falso).

#### Recall o Sensibilidad

Los positivos que he clasificado bien vs todos los positivos que había. Métrica que se utiliza cuando queremos hacer foco en minimizar los FN (Falsos Negativos). Claro ejemplo puede ser un test de COVID. No me importa tanto que haya FP, ya que las consecuencias son aislamientos preventivos, mientras que tener FN, es decir, personas con COVID, cuyo resultado del test es negativo, si es grave ya que pueden producir más contagios.

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

### Precision

De los que ha predicho como 1, cuántos en realidad ha acertado. Precision, a diferencia del recall, pone foco en minimizar los FP. Como ejemplo podemos poner un filtro anti-spam. Que se cuele algún correo de spam (FN) no me importa. Ahora bien, si se clasifica como spam un correo importante del jefe (FP) y no lo leemos, sí es más grave.

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

### Specifity

Es el número de ítems correctamente identificados como negativos sobre el total de negativos. Es lo opuesto al Recall.

$$\text{Specifity} = \text{TN} / (\text{TN} + \text{FP})$$

### F1-Score

Combinación de las métricas Precision y Recall. Esta métrica se utiliza para comparar clasificadores, ya que es algo más compleja de entender, pero muy útil cuando tenemos clasificadores cuyos valores de recall y precision se intercalan unos con otros y no está muy claro cuál es mejor, a no ser que tengamos claro que hay que centrarse o bien en el recall o en el precision. El rango del F1-score va de 0 a 1, como las métricas anteriores.

$$\text{F1-score} = 2 * \text{Precision} * \text{Recall} / (\text{Precision} + \text{Recall})$$

### Precision vs Recall plot

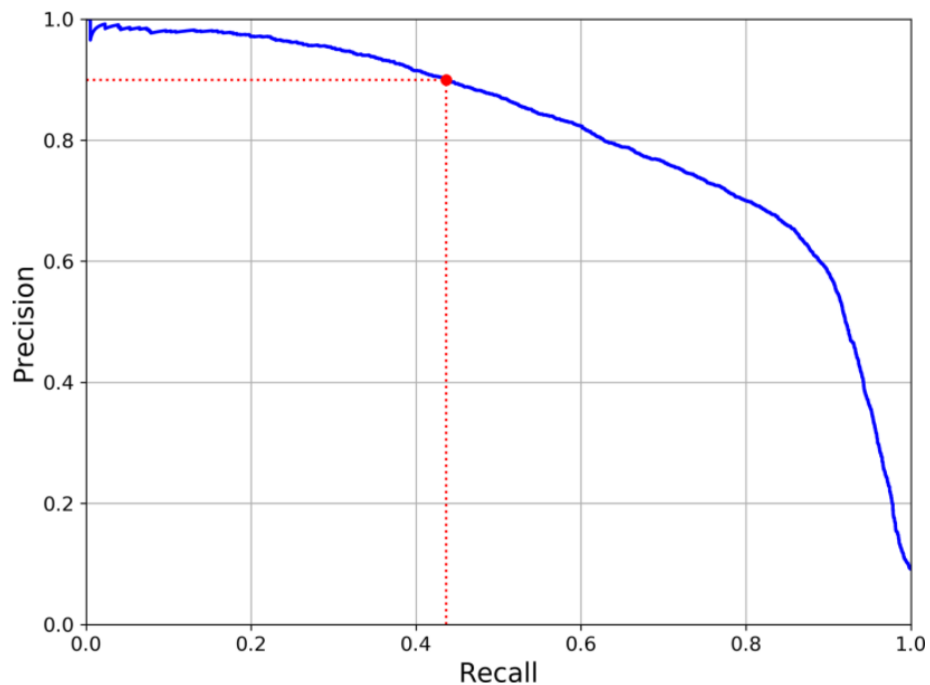
Recuerda que el modelo por defecto calcula las probabilidades de pertenecer a una clase, y que por defecto, si por ejemplo el clasificador es binario, y la probabilidad cae por encima del 50% se clasificaría como 1, como positivo. Ahora bien, modificar ese threshold cambia completamente la matriz de confusión. Si subimos el threshold somos más restrictivos con los 1s, y por tanto empezará a clasificar más valores como 0s y como FN. En cambio si lo bajamos, somos más permisivos con los 1s, los clasificaríamos casi todos, pero subirían los FP.

Es por ello que existen curvas como ROC o Precision vs Recall con las que podremos ver diferentes comportamientos del clasificador, teniendo en cuenta varios thresholds, y todo ello en una misma gráfica.

Mediante esta curva obtendremos el punto óptimo de precision vs recall. Para usar esta gráfica hay que comprender bien qué significan las métricas de precision y de recall



gracias hay que comprender bien que significan las medidas de precisión y de recall.

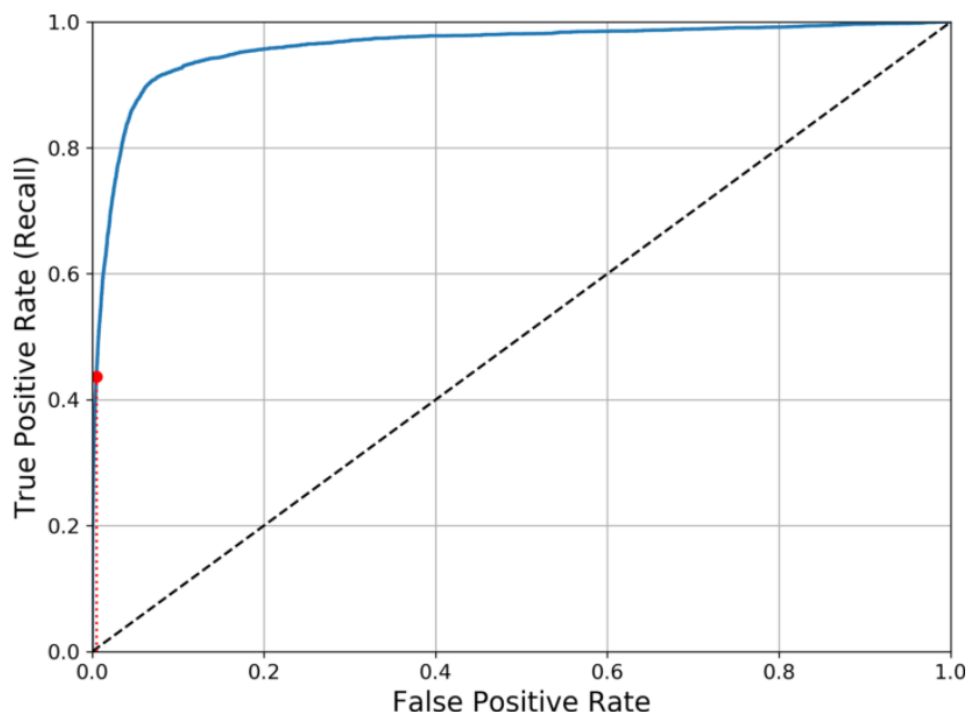


### ROC Curve/AUC

Similar a la anterior curva. Nos sirve para comparar el recall con  $FPR = FP/(FP + TN)$

De esta curva podemos obtener una medida, que es el AUC (Area Under the Curve). El AUC cuanto más cercano a 1, mejor será el clasificador. Si la curva es una línea recta estaremos ante un clasificador muy malo, que no sería muy diferente a un clasificador aleatorio.

**Normalmente se usa la ROC curve cuando tenemos datasets balanceados, mientras que la curva precision-recall es más propia de datasets con una clase mayoritaria.**



```
In [ ]: # Métricas de clasificación
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score,\
roc_auc_score, roc_curve,
precision_recall_curve, confusion_matrix

# Modelo rapido sin dividir en train/test ni ndada
titanic_df2 = titanic_df.drop(columns=['deck', 'alive']).copy()
titanic_df2 = titanic_df2.dropna()
titanic_df2 = pd.get_dummies(titanic_df2)

lr = LogisticRegression()
X = titanic_df2.iloc[:, 1:]
Y = titanic_df2.iloc[:, 0]

lr.fit(X, Y)
preds = lr.predict(X)

print("Score del modelo (accuracy):", round(lr.score(X, Y), 3))
print("Accuracy score:", round(accuracy_score(preds, Y), 3))
print("Recall score:", round(recall_score(preds, Y), 3))
print("Precision score:", round(precision_score(preds, Y), 3))
print("F1 score:", round(f1_score(preds, Y), 3))
print("AUC:", round(roc_auc_score(preds, Y), 3))

y_pred_prob = lr.predict_proba(X)[:, 1]
fpr, tpr, thresholds = roc_curve(Y, y_pred_prob)
plt.plot(fpr, tpr)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.title('ROC curve for titanic classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.grid(True)
```

Score del modelo (accuracy): 0.827

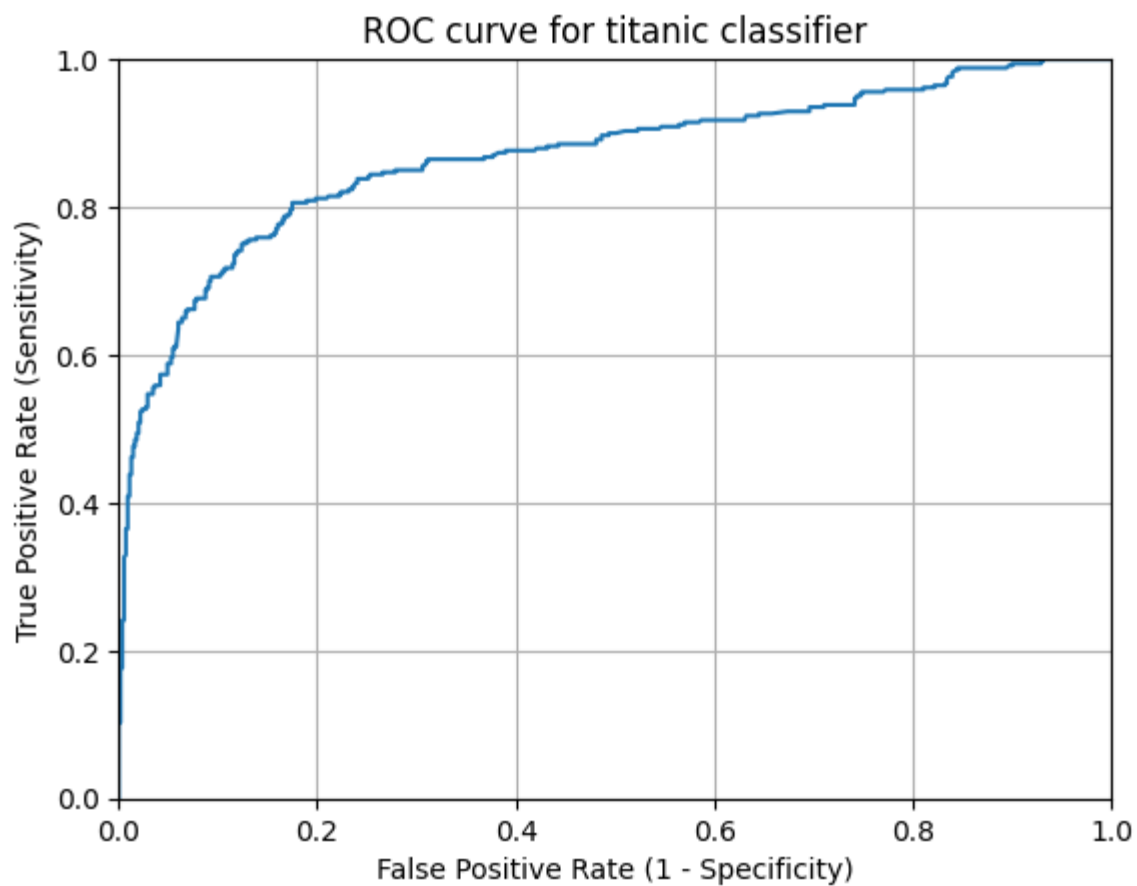
Accuracy score: 0.827

Recall score: 0.796

Precision score: 0.735

F1 score: 0.765

AUC: 0.82



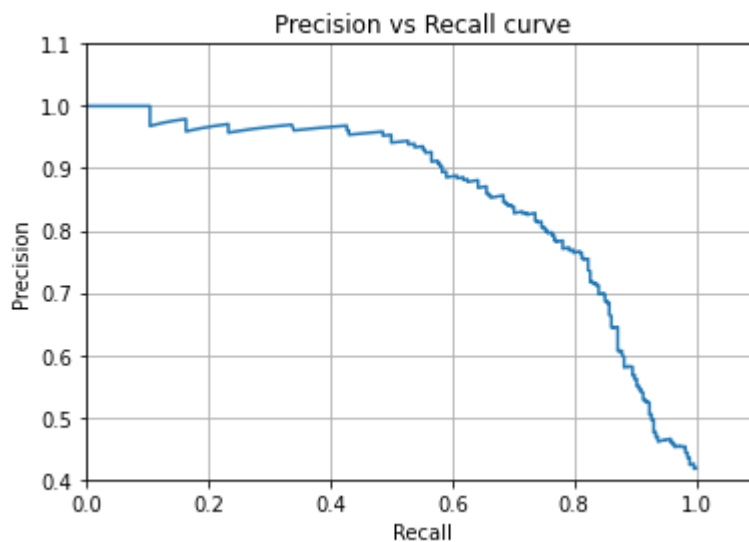
In [ ]: titanic\_df2

```
Out[ ]:
```

	survived	pclass	age	sibsp	parch	fare	adult_male	alone	sex_female	sex_male	...	e
0	0	3.0	22.0	1.0	0.0	7.2500	True	False	0	1	...	
1	1	1.0	38.0	1.0	0.0	71.2833	False	False	1	0	...	
2	1	3.0	26.0	0.0	0.0	7.9250	False	True	1	0	...	
3	1	1.0	35.0	1.0	0.0	53.1000	False	False	1	0	...	
4	0	3.0	35.0	0.0	0.0	8.0500	True	True	0	1	...	
...	...	...	...	...	...	...	...	...	...	...	...	
886	0	2.0	27.0	0.0	0.0	13.0000	True	True	0	1	...	
887	1	1.0	19.0	0.0	0.0	30.0000	False	True	1	0	...	
888	0	3.0	33.0	1.0	2.0	23.4500	False	False	1	0	...	
889	1	1.0	26.0	0.0	0.0	30.0000	True	True	0	1	...	
890	0	3.0	32.0	0.0	0.0	7.7500	True	True	0	1	...	

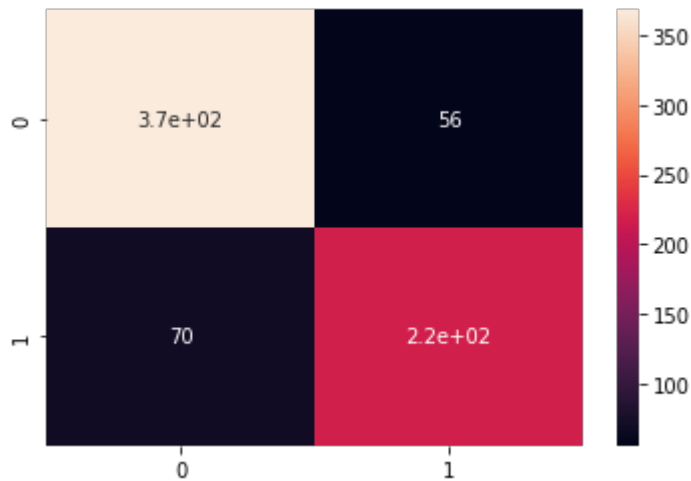
889 rows × 22 columns

```
In [ ]: precision, recall, thresholds = precision_recall_curve(Y, y_pred_prob)
plt.plot(recall, precision)
plt.xlim([0.0, 1.1])
plt.ylim([0.4, 1.1])
plt.title('Precision vs Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.grid(True);
```

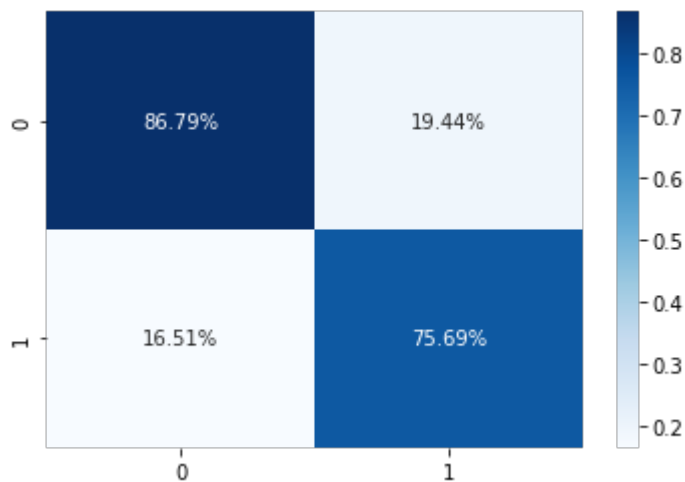


```
In [ ]: c_matrix = confusion_matrix(Y, preds)
print(c_matrix)
import seaborn as sns
sns.heatmap(c_matrix, annot=True);
```

```
[[368  56]
 [ 70 218]]
```



```
In [ ]: sns.heatmap(c_matrix/np.sum(c_matrix,axis=1), annot=True,
                    fmt='.2%', cmap='Blues');
```



## 16.2 Métricas para regresión

$R^2$

El  $R^2$  es el coeficiente de determinación, representa el porcentaje de variación de la variable dependiente vs la independiente para un modelo lineal. Se mide entre 0 y 1. El problema es que  $R^2$  no nos indica si mi modelo se ajusta bien a mis datos. Un buen modelo puede tener un  $R^2$  bajo, e incluso un modelo mal entrenado podría llegar a tener un  $R^2$  alto. [En este artículo tienes más detalles](#)

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

Mean Squared Error (MSE)

La media de los errores al cuadrado. ¿Por qué al cuadrado? Porque al calcular la suma no queremos que se nos cancelen unos errores con otros. ¿Qué esté al cuadrado nos sirve como medida de errores? Perfectamente, es más, esta métrica magnifica los errores grandes, por lo que si esos son los errores que no corregimos, seguiremos teniendo un MSE alto.

MSE es una métrica muy utilizada ya que nos sirve para hacer foco en los errores grandes. Como los datos están al cuadrado, no suele ser una métrica para presentar resultados ya que los datos están "distorsionados". **Se utiliza mayormente para comparar modelos.**

$$MSE = \frac{1}{n} \sum \underbrace{(y - \hat{y})^2}_{\substack{\text{The square of the difference} \\ \text{between actual and} \\ \text{predicted}}}$$

Root Mean Squared Error (RMSE)

Igual que en el caso anterior, pero ahora en unidades del target. Esta métrica nos servirá tanto para comparar modelos, como para presentar resultados.

$$RMSE = \sqrt{\sum_{i=1}^n \frac{(\hat{y}_i - y_i)^2}{n}}$$

Mean Absolute Error (MAE)

Muy útil cuando queremos medir errores en las unidades de la variable. Te dice qué error puedes esperar en la predicción de la variable. Muy bueno cuando hay outliers, ya que lo vamos a ver claramente reflejado en la medida.

$$MAE = \frac{1}{n} \sum_{i=1}^n |x_i - \hat{x}_i|$$

Mean Absolute Percentage Error (MAPE)

Es la única métrica acotada entre 0 y 1. Proporciona, en media, el % de error para cada punto respecto a su true label. Cuando queremos comparar modelos con diferentes targets, como por ejemplo en series temporales, las otras métricas no nos sirven porque van en función de las unidades de cada target, sin embargo mediante el MAPE podremos comparar con la misma unidad de medidas.

$$\text{MAPE} = \frac{100\%}{N} \sum_{i=1}^N \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

```
In [ ]: # Métricas de regresión
from sklearn.metrics import r2_score, mean_absolute_error,
mean_squared_error
from sklearn.linear_model import LinearRegression

diamonds_df2 = pd.get_dummies(diamonds_df)

lr = LinearRegression()
X = diamonds_df2.drop(columns = ['price'])
Y = diamonds_df2['price']

lr.fit(X, Y)
preds = lr.predict(X)

print("Score del modelo (R^2):", round(lr.score(X, Y), 4))
print("R^2 score:", round(r2_score(preds, Y), 4))
print("MAE score:", round(mean_absolute_error(preds, Y), 4))
print("MSE score:", round(mean_squared_error(preds, Y), 4))
print("RMSE score:", round(np.sqrt(mean_squared_error(preds, Y)), 4))

def mean_absolute_percentage_error(y_pred, y_true):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

print("MAPE score:", round(mean_absolute_percentage_error(preds, Y), 4))
```

Score del modelo ( $R^2$ ): 0.9198  
 $R^2$  score: 0.9128  
MAE score: 740.476  
MSE score: 1276866.7661  
RMSE score: 1129.9853  
MAPE score: 39.1852



[Volver al índice](#)

## 17. Decidir qué modelos

Existen una gran cantidad de modelos que nos solventarán problemas de todo tipo. Ahora bien, ¿qué modelo se adapta mejor a los datos? Va a depender de los siguientes factores:

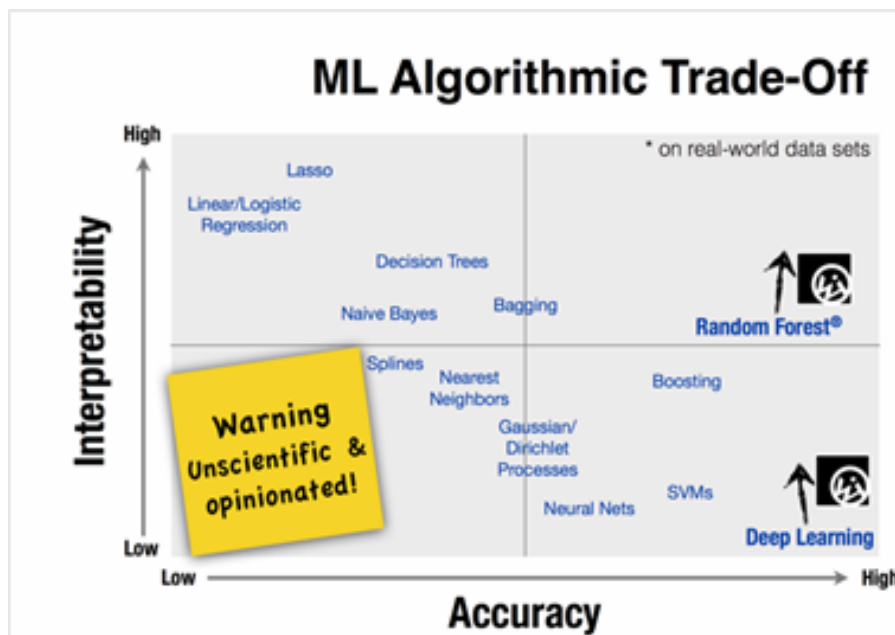
### Volumen de datos

Lo ideal es cuantos más datos mejor, pero si tenemos:

1. **Pocos datos y muchas features:** elige algoritmos con alto bias y low variance.  
Algoritmos que tienen algo más de error, pero generalizan mejor. Linear regression, naïve bayes, Linear SVM
2. **Muchos datos y pocas features:** elige algoritmos con low bias y high variance: KNN, Decision trees, resto de kernels de SVM

### Accuracy vs interpretability

Lo más importante a la hora de elegir modelo. ¿Cuánto de explicable tiene que ser mi modelo? Si tenemos que dar una justificación detallada de las decisiones que toma mi modelo, necesitare un algoritmo de caja blanca, de lo contrario, será un caja negra. Normalmente los algoritmos caja negra tienen una mejor precisión. El problema que presentan es que resulta muy difícil seguir la traza de los outputs del modelo y no sabemos muy bien qué está haciendo.



### Velocidad de entrenamiento

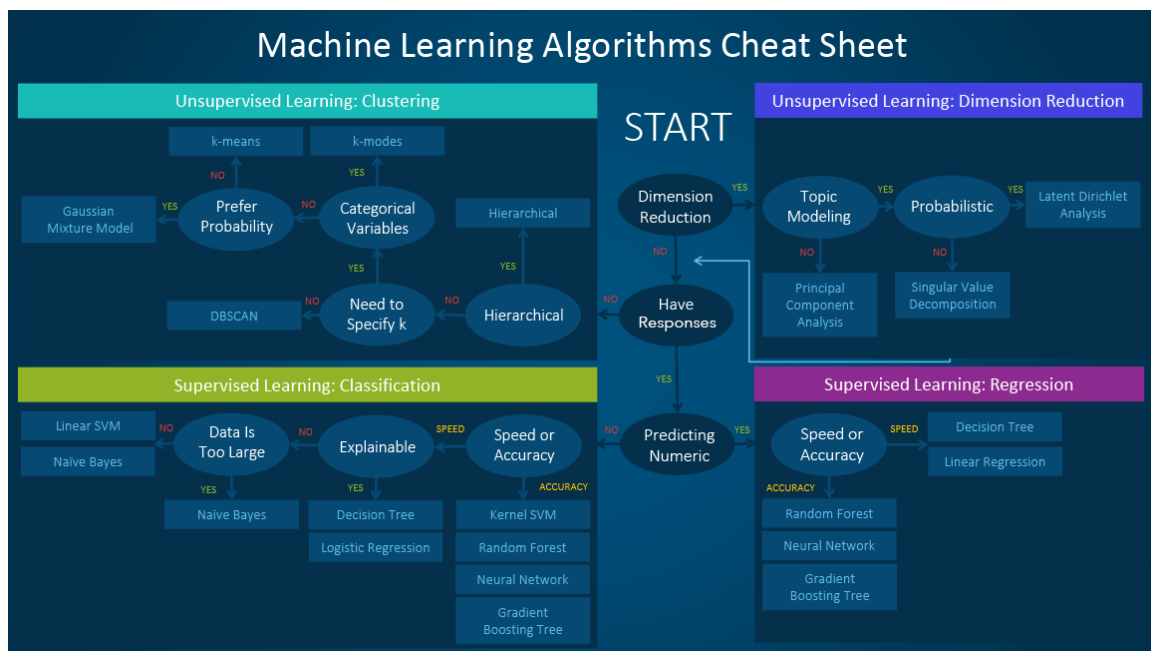
Normalmente los algoritmos más precisos suelen tener un tiempo de entrenamiento alto.

1. Rápidos: Logistic regression, linear models, naive bayes.
2. Lentos: SVM (por el tuning de hiperparámetros), RRNN, ensembles.

## Relaciones lineales entre los datos

La importancia del análisis exploratorio. Si la relación con el target es lineal, va a predecir bien un modelo lineal como logistic regression o SVM lineal. Si no tendremos que acudir a otros modelos que permitan modelar relaciones no lineales, como por ejemplo random forest, KNN, kernel SVM o redes neuronales.

¿Cómo comprobamos esto? Mediante el análisis exploratorio. Aunque otra opción sería entrenar una regresión lineal y analizar la aleatoriedad de los residuos. Deberían caer en una nube alrededor del 0, sin patrones tipo una parábola.



[Volver al índice](#)

## 18. Elegir hiperparámetros

Como ya sabes, cada dataset es de su padre y de su madre, y por tanto es imposible determinar el modelo con sus hiperparámetros que mejor se ajusten a los datos. Por tanto, tendremos que probar varias combinaciones. Por suerte `sklearn` tiene una función llamada `GridSearchCV` que permite probar varias combinaciones de una manera automatizada.

Empieza iterando unos pocos hiperparámetros y luego ve subiendo, según los resultados de esa ejecución.

En este apartado veremos posibles hiperparámetros a emplear en los algoritmos más

En este apartado veremos posibles hiperparámetros a emplear en los algoritmos más

```
In [ ]: # REGRESION LOGISTICA

grid_logreg = {

    "penalty": ["l1", "l2"], # Regularizaciones L1 y L2.
    "C": [0.1, 0.5, 1.0, 5.0], # Cuanta regularizacion
                                queremos

                                "max_iter": [50,100,500], # Iteraciones del
                                Gradient Descent. No suele impactar mucho
                                                                # pero en ocasiones
                                aparecen warnings diciendo que se aumente

    "solver": ["liblinear"] # Suele ser el más rápido
}

# KNN
grid_neighbors = {"n_neighbors": [3,5,7,9,11],          # Pares acepta
                  "weights": ["uniform", "distance"]    # Ponderar o no las
                  # función de la
                  inversa de la distancia a cada vecino
                  }

# ARBOL DE DECISION
grid_arbol = {"max_depth": list(range(1,10)) # Profundidades del árbol.
              # Cuanto más profundo, mas posibilidades de overfitting,
              # pero mas preciso en
              entrenamiento.
              }

# RANDOM FOREST
grid_random_forest = {"n_estimators": [120], # El Random Forest no suele
                                                empeorar por exceso de
                                                # estimadores. A partir de
                                                cierto numero no merece la pena
                                                # perder el tiempo ya que
```

no mejora mucho más la precisión.

# Entre 100 y 200 es una

buena cifra

"max\_depth": [3,4,5,6,10,15,17], # No le afecta  
tanto el overfitting como al decision tree.

# Podemos probar

mayores profundidades

"max\_features": ["sqrt", 3, 4] # Numero de features  
que utiliza en cada split.

# cuanto más bajo,

mejor generalizará y menos overfitting.

}

# SVM

grid\_svm = {"C": [0.01, 0.1, 0.3, 0.5, 1.0, 3, 5.0, 15, 30], # Parametro  
de regularizacion

"kernel": ["linear","poly","rbf"], # Tipo de kernel, probar  
varios

"degree": [2,3,4,5], # Cuantos grados queremos para el  
kernel polinómico

"gamma": [0.001, 0.1, "auto", 1.0, 10.0, 30.0] # Coeficiente  
de regularizacion para los kernels rbf, poly y sigmoid

}

# GRADIENT BOOSTING

grid\_gradient\_boosting = {"loss": ["deviance"], # Deviance suele ir  
mejor.

"learning\_rate": [0.05, 0.1, 0.2, 0.4, 0.5],

# Cuanto más alto, más aporta cada nuevo árbol

"n\_estimators": [20,50,100,200], # Cuidado con  
poner muchos estimadores ya que vamos a

#

*sobreajustar el modelo*

```
"max_depth": [1,2,3,4,5], # No es necesario  
poner una profundidad muy alta. Cada nuevo
```

```
# arbol va
```

```
corrigiendo el error de los anteriores.
```

```
"max_features": ["sqrt", 3, 4], # Igual que en  
el random forest
```

```
}
```

[Volver al índice](#)

## 19. Definimos pipelines y probamos

Ya tenemos los modelos y sus hiperparámetros elegidos, ahora solo queda configurar los pipelines para cada modelo. Por norma general, todo el preprocesado y limpieza de datos ya tiene que estar hecho en este punto, por lo que no será necesario cargar en exceso los pipelines. ¿Qué tienes que configurar en los pipelines?

1. **El modelo:** como último paso del pipeline. Puedes configurarle hiperparámetros que no vayas a iterar en la propia declaración del modelo.
2. **Escalados:** StandardScaler o MinMaxScaler. Menos en árboles.
3. **Feat. reduction:** prueba algún PCA o SelectKBest en algunos de los algoritmos, iterando sus hiperparámetros.
4. **Missings:** se suelen tratar antes, pero siempre tienes la opción de incluir un imputador de missings que afecte a todo el dataset.

**NOTA:** ten en cuenta que cuando se ejecute la validación cruzada, si imputaste missings con la media, por ejemplo, esta media está calculada sobre todo el conjunto de train, y en la validación cruzada (divide train en train+validation) estamos contaminando los datos de train (de la validación cruzada) con los de validación. No es algo grave, pero lo correcto es para cada kfold calcular sus estadísticos de train y aplicarlos en train y validation. Esto lo hace solo GridSearchCV siempre y cuando el imputador de missings esté incluido en el pipeline. El problema es que hacer una configuración tan personalizada y adaptada a tus datos en el pipeline conlleva muchísimo trabajo y al final se suelen dejar los pipelines para operaciones de escalado y de feature selection.

En el siguiente código tendrás varios ejemplos de declaración de pipelines.

```
In [ ]: from sklearn.preprocessing import StandardScaler
        from sklearn.impute import SimpleImputer
        from sklearn.pipeline import Pipeline
        from sklearn.feature_selection import SelectKBest
        from sklearn.model_selection import GridSearchCV

        from sklearn.svm import SVC
        from sklearn.linear_model import LogisticRegression
        from sklearn.ensemble import RandomForestClassifier

        # Si solo es el modelo, no hará falta meterlo en un pipeline
        rand_forest = RandomForestClassifier()

        svm = Pipeline([("scaler",StandardScaler()),
                        ("selectkbest",SelectKBest()),
                        ("svm",SVC())
                        ])

        reg_log = Pipeline([("imputer",SimpleImputer()),
                            ("scaler",StandardScaler()),
                            ("reglog",LogisticRegression())
                            ])

        ...

        Para iterar hiperparámetros de varios elementos del pipeline, le ponemos
        un nombre
        a cada elemento en el pipeline, por ejemplo 'selectkbest' y 'svm', para
        luego en el
        grid de hiperparametros identificar sus respectivos parametros mediante
        el nombre
        que le hayamos puesto en el pipeline, dos guines bajos y el nombre del
        hiperparámetro.
        ...

        grid_random_forest = {"n_estimators": [120],
                              "max_depth": [3,4,5,6,10,15,17],
                              "max_features": ["sqrt", 3, 4]
```

```
}

svm_param = {
    'selectkbest__k': [1,2,3],
    'svm__C': [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9],
    'svm__kernel': ["linear", "poly", "rbf"],
    'svm__coef0': [-10., -1., 0., 0.1, 0.5, 1, 10, 100],
    'svm__gamma': ('scale', 'auto')
}

reg_log_param = {
    "imputer__strategy": ['mean', 'median',
    'most_frequent'],
    "reglog__penalty": ["l1", "l2"],
    "reglog__C": np.logspace(0, 4, 10)
}
```

```
In [ ]: from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
```

```
In [ ]: # Almaceno en una lista de tuplas los modelos (nombre que le pongo, el
        # modelo, hiperparametros)
        models = [('rand_forest', rand_forest, grid_random_forest),
                   ('svm', svm, svm_param),
                   ('reg_log', reg_log, reg_log_param)]

        # Declaro en un diccionario los pipelines e hiperparametros
        models_gridsearch = {}

        for i in models:
            models_gridsearch[i[0]] = GridSearchCV(i[1],
                                                    i[2],
                                                    cv=10,
                                                    scoring="accuracy",
                                                    verbose=1,
                                                    n_jobs=-1)

            models_gridsearch[i[0]].fit(X_train, y_train)
```

Fitting 10 folds for each of 21 candidates, totalling 210 fits  
 Fitting 10 folds for each of 1152 candidates, totalling 11520 fits  
 Fitting 10 folds for each of 60 candidates, totalling 600 fits

```
In [ ]: best_grids = [(i, j.best_score_) for i, j in models_gridsearch.items()]

        best_grids = pd.DataFrame(best_grids, columns=["Grid", "Best
        score"]).sort_values(by="Best score", ascending=False)
        best_grids
```

```
Out[ ]:
```

	Grid	Best score
2	reg_log	0.958333
0	rand_forest	0.950000
1	svm	0.950000

```
In [ ]: # El mejor ha sido la regresion logistica. Ya esta entrenada con todo
        # train
        models_gridsearch['reg_log'].best_estimator_
```

```
Out[ ]: Pipeline(steps=[('imputer', SimpleImputer()), ('scaler', StandardScaler()),
                        ('reglog', LogisticRegression(C=7.742636826811269))])
```



```
In [ ]: # La probamos en test
models_gridsearch['reg_log'].best_estimator_.score(X_test, y_test)
```

```
Out[ ]: 1.0
```

[Volver al índice](#)

## 20. Resultados

Una vez elegidas las métricas y evaluados varios modelos, ya tendríamos los resultados del modelo. Sólo queda interpretarlos. Por desgracia este punto va a depender mucho del negocio.

1. **Evaluar los resultados en test:** tenemos que comprobar que los resultados no difieren mucho del entrenamiento. Si son inferiores es debido a que el modelo está sobreentrenado (ver abajo para solucionarlo). Si son superiores es raro, probablemente porque no tengamos muchos datos. Habría que intentar coseguir más observaciones.
2. Evaluar los resultados respecto a las **necesidades de negocio**
3. **Almacenar** todas las muestras utilizadas en el entrenamiento, así como los scripts y el propio modelo entrenado. La puesta en producción no es el objetivo a cubrir en este notebook.
4. Elegir y evaluar la/las métrica/s con las que **presentaremos los resultados del modelo**. No tenemos por qué elegir la misma métrica utilizada en la búsqueda del mejor modelo, ya que no siempre es la más entendible: MSE o AUC.

¿Qué hacer si tenemos overfitting?

Posibles opciones para reducir el overfitting

1. **Cross validation:** si hemos hecho bien el paso anterior, el cross validation es la mejor técnica para evitar el overfitting.
2. **Entrenar con más datos:** no quitar datos a test, sino intentar conseguir más datos.
3. **Eliminar features:** PCA o un algoritmo de feature selection podría ser una buena solución
4. **Max depth:** reducirlo si estamos con árboles
5. **Regularización:** aumentarla en los modelos que permitan regularización como regresiones lineales y SVM
6. **Ensembles:** sobretodo los algoritmos de bagging como el random forest no suelen sobreajustarse tanto.

```
In [ ]: # Guardar el modelo
import pickle

with open('finished_model.model', "wb") as archivo_salida:
    pickle.dump(models_gridsearch['reg_log'].best_estimator_,
archivo_salida)
```

```
In [ ]: # Para volver a leer el modelo
with open('finished_model.model', "rb") as archivo_entrada:
    pipeline_importada = pickle.load(archivo_entrada)

print(pipeline_importada)
```