EDA Exploratory Data Analysis

Análisis exploratorio de datos

El Análisis Exploratorio de Datos se refiere al proceso de realizar una serie de investigaciones inciales sobre los datos que tenemos para poder descubrir patrones, detectar anomalías, probar hipótesis y comprobar suposiciones con la ayuda de estadísticas y representaciones gráficas.

1. Entender las variables

- que variables temenos
.head(), .tail(), .describe(), .info(), .shape
- que tipos de datos
.dtypes(), .info()
- si temenos nulos o duplicados
.isnull().sum()
.duplicated().sum()
- que valores unicos temenos
.unique(), .value_counts()

librería sidetable:
stb.freq() devuelve el value_counts de variables categóricas, mas el porcentaje, cuenta cumulativa y porcentaje cumulativa
stb.missing() tabla de cuenta de nulos y el porcentaje del total

2. Limpiar el dataset

- quitar duplicados (filas o columnas)
- cambiar nombres de columnas
- cambiar tipo de datos de columnas
- ordenar columnas
- separar columna en dos con str.split()
- crear intervalos con pd.cut()
- crear porcentajes o ratios
- decidir como tratar outliers: mantenerlos, eliminarlos, o reemplazarlos con la media, mediana o moda; o aplicar una imputacion

- decidir como tratar nulos:
  - eliminar filas o columnas con nulos drop.na()
  - imputar valores perdidos:
    - reemplazarlos con la media, mediana o moda usando .fillna() o .replace()
    - imputer con metodos de machine learning usando la libreria sklearn: Simple-Imputer, Iterative-Imputer, o KNN Imputer

3. Analizar relaciones entre variables

Analizar relaciones entre las variables
- para encontrar patrones, relaciones o anomalias
Relaciones entre dos variables numéricas:
- scatterplot
- regplot – scatterplot con línea de regresion
- matriz de correlación y heatmap
- joinplot – permite emparejar dos gráficas – una histograma con scatter o reg plot por ejemplo
Relaciones entre dos variables categóricas:
- countplot
Relaciones entre variables numéricas y categóricas:
- swarmplot
- violinplot
- pointplot
- boxplot

ETL: Extract, Transform, Load

Extraccion
- obtener datos crudos y almacenarlos
  - Tablas de bases de datos SQL o NoSQL
  - Ficheros de texto plano
  - Emails
  - Información de páginas web
  - Hojas de cálculo
  - Ficheros obtenidos de API's

Transformación
- procesar los datos, unificarlos, limpiarlos, validarlos, filtrarlos, etc.
  - Formetear fechas
  - Reordenar filas o columnas
  - Unir o separar datos
  - Combinar las fuentes de datos
  - Limpiar y estandarizar los datos
  - Verificar y validar los datos
  - Eliminar duplicados o datos erroneos
  - Filtrado, realización de calculos o agrupaciones

Carga
- cargar los datos en su formato de destino, el tipo de lo cual dependerá de la naturaleza, el tamaño y la complejidad de los datos. Los sistemas más comunes suelen ser:
  - Ficheros csv
  - Ficheros json
  - Bases de datos
  - Almacenes de datos (Data Warehouse)
  - Lagos de datos (Data Lakes)

APIs

`import requests` libreria para realizar peticions HTTP a una URL, para hacer web scraping
`url = 'enlace'` el enlace de la que queremos extraer datos
`header = {}` opcional; contiene informacion sobre las peticiones realizadas (tipo de ficheros, credenciales)
`response = requests.get(url=url, header = header)` pedimos a la API que nos de los datos
`variables = {'parametro1':'valor1', 'parametro2':'valor2'}`
`response = request.get(url=url, params=variables)` pedimos a la API que nos de los datos con los parametros segun el diccionario de parametros que le pasamos
`response.status_code` devuelve el status de la peticion
`response.reason` devuelve el motive de codigo de estado
`response.text` devuelve los datos en formato string
`response.json()` devuelve los datos en formato json
`df = pd.json_normalize(response.json)` devuelve los datos en un dataframe

Codigos de respuesta de HTTP

1XX informa de una respuesta correcta	4XX error durante peticion
2XX codigo de exito	401 peticion incorrecta
200 OK	402 sin autorizacion
201 creado	403 prohibido
202 aceptado	404 no encontrado
204 sin contenido	5XX error del servidor
3XX redireccion	501 error interno del servidor
	503 servicio no disponible

Machine Learning: Preparación

Hipotesis Nula y Errores Tipo I y II

Hipótesis nula (H0)
- en general es la afirmación contraria a la que queremos probar

Hipótesis alternativa (H1)
- en general la afirmación que queremos comprobar

p-valor
- medida de la probabilidad de que una hipótesis nula sea cierta
- valor entre 0 y 1
- si *p-valor* < 0.05 ✗ Rechazamos la hipótesis nula.
- si *p-valor* > 0.05 ✓ Aceptamos la hipótesis nula.

Error Tipo I:
- rechazar la hipótesis nula cuando es verdadera

Error Tipo II:
- aceptar la hipótesis nula cuando es falsa

Tests estadísticos

Normalidad

- la variable respuesta tiene que tener una distribución normal para poder crear un modelo de regresión lineal

Visualmente:
- histograma o distribución
- grafico de cuantiles teóricos (Q-Q)

más alineados están los puntos entorno a la recta, más normales serán nuestros datos

`import statsmodels.api as sm`
`sm.qqplot(datos, line ='45')`

Metodos analiticos:

Asimetría
- distribuciones asimétricas positivas: media > mediana y moda
- distribuciones asimétricas negativas: media < mediana y moda

`from scipy.stats import skew`
`skew(datos_normales)` método de scipy que calcula el sesgo
`df['columna'].skew()` método de pandas que calcula el sesgo

Curtosis
- **leptocurtosis**: valor de curtosis mayor que 0 (pico alto)
- **mesocurtosis**: valor de curtosis igual a 0 (pico medio)
- **platicurtosis**: valor de curtosis menor que 0 (plana)

`from scipy.stats import kurtosistest`
`kurtosistest(datos)` devuelve un p-valor
- p-valor del test > 0.05: datos normales ✓
- p-valor del test < 0.05: datos NO normales

Test de Shapiro-Wilk
- para muestras < 5000
- hipótesis nula: distribución normal

`from scipy import stats`
`stats.shapiro(df["datos"])`
- p-valor del test > 0.05: datos normales ✓
- p-valor del test <) 0.05: datos NO normales

Test de Kolmogorov-Smirnov
- para muestras > 5000
- hipótesis nula: distribución normal

`from scipy import kstest`
`kstest(df["datos"], 'norm')`
- p-valor del test > 0.05: datos normales ✓
- p-valor del test < p-valor (alfa) 0.05: datos NO normales

Tests estadísticos

Independencia entre variables predictoras

- las variables predictoras tienen que ser independientes para poder crear un modelo de regresión lineal

Variables numéricas: Correlaciones
- pairplot
`sns.pairplot(df)`
- covarianza
`df_numéricas.cov()`
- correlación de Pearson (relación lineal)
`df_numéricas.corr()`
- correlación de Spearman (relación no lineal)
`df_numéricas.corr(method = 'spearman')`
- correlación de Kendall (datos numéricos pero categóricos y ordinales)
`df_numéricas.corr(method = 'kendall')`

Variables categóricas: Chi-cuadrado
- V-Cramer: varía entre 0 y 1
  - más cerca a 1 más dependientes
  - resultado < 0,7 para hacer ML ✓

`import researchpy as rp`
`crosstab, test_results, expected = rp.crosstab (df["col1"], df["col2"], test= "chi-square", expected_freqs= True, prop= "cell")`
`test_results` devuelve los resultados del test en un dataframe

Homocedasticidad (homogeneidad de varianzas)

- las variables predictoras tienen que tener homogeneidad de varianzas en comparación con la variable respuesta

Visualmente:
- violinplot
- boxplot
- regplot (columnas numéricas vs variable respuesta)

Metodos analiticos:
- test de Levene (más robusto ante falta de normalidad) o Bartlett

`from scipy import stats`
`from scipy.stats import levene`

Variables categóricas:
- hay que crear un dataframe para cada valor único de las columnas categóricas

```
df_valor1 = df[df['col1'] == 'valor1']['col_VR']
df_valor2 = df[df['col1'] == 'valor2']['col_VR']
levene_test = stats.levene(df_valor1, df_valor2, center='median')
bartlett_test = stats.bartlett(df_valor1, df_valor2, center='median')
```

Variables numéricas:
- hay que crear un dataframe de las columnas numéricas sin la variable respuesta

```
for col in df_numericas.columns:
    statistic, p_val = levene(df[col], df['col_VR'], center='median')
    resultados[col] = p_val
```

devuelve los p-valores en un diccionario
- p-valor del test > 0.05: varianzas iguales, homocedasticidad ✓
- p-valor del test < 0.05: varianzas diferentes, heterocedasticidad

Normalización

Método manual

- cogemos el valor que queremos normalizar y restamos la media de la columna, y dividimos el resultado por el maximo restado por el minimo de la columna

`df["col_norm"] = (df["col_VR"] - df["col_VR"].media()) / (df["col_VR"].max() - df["col_VR"].min())`

Método logarítmica

no se puede hacer si algún valor sea 0

`df["col_norm"] = df["col_VR"].apply(lambda x: np.log(x) if x > 0 else 0)`

Método raiz cuadrada

```
import math
df["col_norm"] = df["col_VR"].apply(lambda x: math.sqrt(x))
```

Método stats.boxcox()

aplica una transformación logarítmica para los valores positivos y exponencial para valores negativos de nuestra columna

```
from scipy import stats
df["col_norm"], lambda ajustada = stats.boxcox(df["col_VR"])
```

Método MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler
modelo = MinMaxScaler(feature_range=(0,1), copy=True)
modelo.fit(df["col_VR"])
datos_normalizados = modelo.transform(df["col_VR"])
df_datos_norm = pd.DataFrame(datos_normalizados, columns = ['col_norm'])
df['col_norm'] = df_datos_norm
```

ANOVA

```
import statsmodels.api as sm
from statsmodels.formula.api import ols
lm = ols('col_VR ~ col_VP1  + col_VP2 + col_VP3', data=df).fit()
```

devuelve un dataframe de los resultados:

df (degrees of freedom): para variables categóricas será el número de valores únicos menos 1; para variables numéricas será siempre 1

sum_sq: medida de variación/desviación de la media

mean_sq: es el resultado de dividir la suma de cuadrados entre el número de grados de libertad.

F: un test que se utiliza para evaluar la capacidad explicativa que tiene la variable predictora sobre la variación de la variable respuestae

- PR(>F): si el p-valor < 0.05 es una variable significativa; que puede afectar a la VR

`lm.summary()` devuelve una resumen de los resultados:

coef: representa los cambios medios en la VR para una unidad de cambio en la VP mientras se mantienen constantes el resto de las VP; los signos nos indican si esta relación es positiva o negativa

std err: cuanto menor sea el error estándar, más precisa será la estimación

t: es el resultado de dividir el coeficiente entre su error estándar

Machine Learning: Preprocesamiento

Estandarización

- cambiar los valores de nuestras columnas de manera que la desviación estándar de la distribución sea igual a 1 y la media igual a 0; para que las VP sean comparables

Método manual

```
df["col_esta"] = (df ["col_VR"] - df ["col_VR"].media()) / (df ["col_VR"].std()
```

Sklearn StandardScaler

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df_num_sin_VR)
datos_estandarizados = scaler.transform (df_num_sin_VR)
df_datos_esta = pd.DataFrame(datos_estandarizados, columns = df_num_sin_VR.columns)
```

Sklearn RobustScaler

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
scaler.fit(df_num_sin_VR)
datos_estandarizados = scaler.transform (df_num_sin_VR)
df_datos_esta = pd.DataFrame(datos_estandarizados, columns = df_num_sin_VR.columns)
```

Encoding

Variables categóricas

Ordinaria: no requiere números pero sí consta de un orden o un puesto; diferencias de medianas entre categorías

Nominal: variable que no es representada por números, no tiene algún tipo de orden, y por lo tanto es matemáticamente menos precisa; no habrá grandes diferencias de medianas entre categorías

Binaria: dos posibilidades; puede tener orden o no

Variables sin orden: creamos una columna nueva por valor único, asignando unos y zeros

One-Hot Encoding

```
from sklearn.preprocessing import OneHotEncoder
oh = OneHotEncoder()
df_transformados = oh.fit_transform(df[['columna']])
oh_df = pd.DataFrame(df_transformados.toarray())
oh_df.columns = oh.get_feature_names_out()
df_final = pd.concat([df, oh_df], axis=1)
```

get_dummies

```
df_dum = pd.get_dummies(df['col'], prefix='prefijo', dtype=int)
df[df_dum.columns] = df.dum
df.drop('col', axis=1, inplace=True)
```

Variables que tienen orden:

Label Encoding asigna un número a cada valor único de una variable

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['col_VR_le'] = le.fit_transform(df[col_VR'])
```

map() asigna el valor que queramos según el mapa que creamos

```
df['col_VR_map'] = df[col_VR'].map(diccionario)
```

Ordinal-Encoding asignamos etiquetas basadas en un orden o jerarquía

```
from sklearn.preprocessing import OrdinalEncoder
```

Regresión Lineal: Métricas

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

R2: representa la proporción de la varianza que puede ser explicada por las VP del modelo; mayor R2=mejor modelo

```
r2_score(y_train,y_predict_train)
r2_score(y_test,y_predict_test)
```

MAE (Mean absolute error): medida de la diferencia entre los valores predichos vs los reales; menor MAE=mejor modelo

```
mean_absolute_error(y_train,y_predict_train)
mean_absolute_error(y_test,y_predict_test)
```

MSE (Mean Squared Error): mide el promedio(media) de los errores al cuadrado; menor MSE=mejor modelo

```
mean_squared_error(y_train,y_predict_train)
mean_squared_error(y_test,y_predict_test)
```

RMSE (Root Mean Squared Error): distancia promedio entre los valores predichos y los reales; menor RMSE=mejor modelo

```
np.sqrt(mean_squared_error(y_train,y_predict_train))
np.sqrt(mean_squared_error(y_test,y_predict_test))
```

Linear Regression: Modelo

1. separar los datos de las variables predictoras (x) de la variable respuesta (y)

```
X = df.drop('col_VR', axis=1)
y = df['col_VR']
```

2. dividimos los datos en datos de entrenamiento y datos de test con train_test_split()

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
```

3. Ajustamos el modelo

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression(n_jobs=-1)
lr.fit(x_train, y_train)
```

4. Hacemos las predicciones

```
y_predict_train = lr.predict(x_train)
y_predict_test = lr.predict(x_test)
```

5. Guardamos los resultados en dataframes y los concatenamos

```
train_df = pd.DataFrame({'Real': y_train, 'Predicted': y_predict_train, 'Set': ['Train']*len(y_train)})
test_df  = pd.DataFrame({'Real': y_test,  'Predicted': y_predict_test,  'Set': ['Test']*len(y_test)})
resultados = pd.concat([train_df,test_df], axis = 0)
```

6. creamos una columna de los residuos: la diferencia entre los valores observados y los de la predicción

```
resultados['residuos'] = resultados['Real'] - resultados['Predicted']
```

Cross-validation

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_validate
cv_scores = cross_val_score(estimator = LinearRegression(), X = X, y = y, scoring = 'neg_root_mean_squared_error', cv = 10)
cv_scores.mean()
```

calcula la media de los resultados de CV de una métrica

```
cv_scores = cross_validate(estimator = LinearRegression(), X = X, y = y, scoring ='r2', 'neg_root_mean_squared_error', cv = 10)
cv_scores["test_r2"].mean()
cv_scores["test_neg_root_mean_squared_error"].mean()
```

calcula las medias de los resultados de validación de múltiples métricas

Regresión Logística: Métricas

Matriz de confusión

Matriz de confusión		Predicción	
		Positivo	Negativo
Realidad	Positivo	Verdadero positivo	Falso negativo
	Negativo	Falso positivo	Verdadero negativo

para crear un heatmap de una matriz de confusión:

```
from sklearn.metrics import confusion_matrix
mat_lr = confusion_matrix(y_test, y_pred_test)
plt.figure(figsize = (n,m))
sns.heatmap(mat_lr, square=True, annot=True=
plt.xlabel('valor predicho')
plt.ylabel('valor real')
plt.show()
```

Métricas

```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score , cohen_kappa_score, roc_curve,roc_auc_score
```

Accuracy (exactitud): porcentaje de los valores predichos están bien predichos

```
accuracy_score(y_train,y_predict_train)
accuracy_score(y_test,y_predict_test)
```

Recall: porcentaje de casos positivos capturados

si preferimos FP, queremos recall alta

```
recall_score(y_train,y_predict_train)
recall_score(y_test,y_predict_test)
```

Precisión (sensibilidad): porcentaje de predicciones positivas correctas

si preferimos FN, queremos precisión alta

```
precision_score(y_train,y_predict_train)
presicion_score(y_test,y_predict_test)
```

Especificidad: porcentaje de los casos negativos capturados

F1: la media de la precisión y el recall

```
f1_score(y_train,y_predict_train)
f1_score(y_test,y_predict_test)
```

kappa: una medida de concordancia que se basa en comparar la concordancia observada en un conjunto de datos, respecto a la que podría ocurrir por mero azar

- <0 No acuerdo
- 0.0-0.2 Insignificante
- 0.2-0.4 Bajo
- 0.4-0.6 Moderado
- 0.6-0.8 Bueno
- 0.8-1.0 Muy bueno

```
cohen_kappa_score(y_train,y_predict_train)
cohen_kappa_score(y_test,y_predict_test)
```

curva ROC: forma gráfica de ver la kappa; la sensibilidad vs. la especificidad

AUC (área under curve): la área bajo la curva ROC; cuanto más cerca a 1, mejor será nuestro modelo clasificando los VP

Balanceo para Regresión Logística

Downsampling

ajustar la cantidad de datos de la categoría mayoritaria a la minoritaria

Método manual

```
df_minoritaria = df[df['col'] == valor_min]
df_muestra = df[df['col'] == valor_max].sample (num_minoritarios, random_state = 42)
df_balanceado = pd.concat([df_minoritaria, df_muestra],axis = 0)
```

Método RandomUnderSample

```
import imblearn
X = df.drop('col_VR', axis=1)
y = df['col_VR']
down_sampler = RandomUnderSampler()
X_down, y_down = down_sampler.fit_resample(X,y)
df_balanceado = pd.concat([X_down, y_down], axis = 1)
```

Método Tomek

```
x = df.drop('col_VR', axis=1)
y = df['col_VR']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 42)
tomek_sampler = SMOTETomek()
X_train_res, y_train_res = tomek_sampler.fit_resample(X_train, y_train)
```

Upsampling

ajustar la cantidad de datos de la categoría minoritaria a la mayoritaria

Método manual

```
df_mayoritaria = df[df['col'] == valor_may]
df_muestra = df[df['col'] == valor_min].sample (num_mayoritarias, random_state = 42)
df_balanceado = pd.concat([df_mayoritaria, df_muestra],axis = 0)
```

Método RandomOverSample

```
import imblearn
X = df.drop('col_VR', axis=1)
y = df['col_VR']
down_sampler = RandomUnderSampler()
X_down, y_down = down_sampler.fit_resample(X,y)
df_balanceado = pd.concat([X_down, y_down], axis = 1)
```

Logistic Regression: Modelo

seguir los mismos pasos como para la Regresión Lineal pero con `LogisticRegression()`

```
from sklearn.linear_model import LogisticRegression
```

Decision Tree: Modelo

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import DecisionTreeRegressor
from sklearn import tree
```

seguir los mismos pasos como para la Regresión Lineal pero con `DecisionTreeRegressor()`

```
arbol = DecisionTreeRegressor(random_state=42)
```

Para dibujar el árbol:

```
fig = plt.figure(figsize = (10,6))
tree.plot_tree(arbol, feature_names = x_train.columns, filled = True)
plt.show()
```

GridSearch y best_estimator_

Despues de hacer las predicciones de un modelo Decision Tree, examinamos lás métricas de los resultados:

- si temenos overfitting hay que reducir la profundidad del modelo

- si temenos underfitting hay que aumentar la profundidad del modelo

```
max_features = np.sqrt(len(x_train.columns))
```

podemos calcular el valor de max_features siendo la raíz cuadrada del número de variables predictoras

`arbol.tree_.max_depth` nos muestra el max depth usado por defecto, para poder ajustarlo; deberíamos usar la mitado como mucho

- GridSearch ejecuta todas las posibles combinaciones de hiperparámetros que le damos con el parámetro 'param' y best_estimator_ devuelve la major combinacion encontrado

1. Definimos un diccionario de los hiperparametros

```
param = {"max_depth": [n,m,l], "max_features": [a,b,c,d], "min_samples_split": [x,y,z], "min_samples_leaf": [r,s,t]} from sklearn.model_selection import GridSearchCV
```

2. Iniciamos el modelo con GridSearch

```
gs = GridSearchCV(estimator = DecisionTreeRegressor(), param_grid = param, cv=10, verbose=-1, return_train_score = True, scoring = "neg_mean_squared_error")
```

3. Ajustamos el modelo en el GridSearch

```
gs.fit(x_train, y_train)
```

4. Aplicamos el método de best_estimator_

```
mejor_modelo = gs.best_estimator_
```

devuelve la mejor combinación de hiperparámetros

5. Volvemos a sacar las predicciones

```
y_pred_test_dt2 = mejor_modelo.predict(x_test)
y_pred_train_dt2 = mejor_modelo.predict(x_train)
```

Importancia de los predictores

```
importancia_predictores = pd.DataFrame(
{'predictor': x_train.columns, 'importancia': mejor_modelo.feature_importances_})
importancia_predictores.sort_values(by=["importancia"], ascending=False, inplace = True)
```

crea un dataframe con la relativa importancia de cada VP

- para los variables categóricas nominales a los cuales se ha aplicado encoding, hay que sumar los resultados de las columnas divididas:

```
df_sum = importancia_predictores_esta.iloc[[n, m]]
importancia_predictores_esta.drop(df_sum.index, inplace = True)
importancia_predictores_esta.loc[n] = ["nombre_col", df_sum["importancia"].sum()]
```

Random Forest: Modelo

seguir los mismos pasos como para el Decision Tree pero con `RandomForestRegressor()`

```
from sklearn.ensemble import RandomForestRegressor
```

- se puede usar los mismos hiperparámetros del best_estimator_ o volver a ejecutar el GridSearch