Práctica 2

M2.851 – Tipología y ciclo de vida de los datos
Bloque 3 – Limpieza y análisis de los datos
By Alberto Caro

1.- **Descripción del dataset**

En esta práctica he utilizado dos *dataset* con las cuales he practicado y aprendido las operaciones de limpieza y análisis de los datos en el contexto de aprendizaje de máquina (ML). Utilicé el lenguaje de programación *python* en el ambiente de desarrollo *Anconda Navigator 3.*

- **DataSet** → *diabetes.csv* (https://www.kaggle.com/uciml/pima-indians-diabetes-database/diabetes.csv)

- **Descripción** → Este conjunto de datos es de autoría del Instituto Nacional de Diabetes y Enfermedades Digestivas y Renales de India. El objetivo de este dataset es predecir si un paciente padece de diabetes basándose en los atributos de estudio y algunas medidas de diagnóstico aplicadas a los datos. El dataset fue obtenido de pacientes mujeres de 21 años de edad o mayores de origen indio (Pima). Los conjuntos de datos se componen de varias variables predictoras y una variable de salida, *Outcome*. Las variables predictoras incluyen número de embarazos, su IMC, nivel de insulina, edad, etc.

- **Importancia** → El estudio de las causas de la diabetes es un tema de mucho interés que promueve mucha innovación e investigación a nivel mundial. El encontrar una cura a esta mortal enfermedad es un gran desafío que todavía mantiene a la comunidad médica trabajando para lograr encontrar una solución.

- **DataSet** → *diabetes.csv* (https://www.kaggle.com/uciml/iris). La utilicé solo con fines didácticos para explicar algunas operaciones de análisis de datos. No la detallo pues es un dataset clásico en ML.

Una primera mirada del dataset *diabetes.csv.*

```
In [12]: df = pd.DataFrame(df)

In [13]: df
```

Out[13]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

La estructura del dataset es la siguiente:

```
In [11]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Vamos a cambiar los nombres de las columnas para que sea más facil interpretar el dataset.

```
In [24]: ds = pd.read_csv('diabetes.csv')
```

```
In [25]: df = pd.DataFrame(ds)
```

```
In [27]: df.columns=['Partos','Glucosa','Presion','Piel','Insulina','IMC','Pedi','Edad','Clase']
```

```
In [28]: df
```

Out[28]:

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

768 rows × 9 columns

Descripción de los campos del dataset:

- **Partos** → Números de embarazos
- **Glucosa** → Concentración de glucosa en plasma a 2 horas en una prueba de tolerancia a la glucosa oral
- **Presión** → Presión arterial diastólica (mm Hg)
- **Piel** → Espesor del pliegue cutáneo del tríceps (mm)
- **Insulina** → Insulina sérica de 2 horas (mu U / ml)
- **IMC** → Indice de masa corporal (peso en kg / (altura en m) ^ 2)
- **Pedi** → Función del pedigrí de la diabetes
- **Edad** → Edad (años) del paciente
- **Clase** → Variable de clase (0: Sin diabetes o 1: Con diabetes)

En total este dataset contiene 768 registros de 9 campos cada uno → (768 , 9). El campo *Clase* es el *OutCome*.

Veremos una breve descripción estadistica del dataset.

```
In [30]: from pandas import set_option

In [32]: set_option('display.width', 100)
         set_option('precision', 2)

In [33]: df.describe()
```

Out[33]:

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
count	768.00	768.00	768.00	768.00	768.00	768.00	768.00	768.00	768.00
mean	3.85	120.89	69.11	20.54	79.80	31.99	0.47	33.24	0.35
std	3.37	31.97	19.36	15.95	115.24	7.88	0.33	11.76	0.48
min	0.00	0.00	0.00	0.00	0.00	0.00	0.08	21.00	0.00
25%	1.00	99.00	62.00	0.00	0.00	27.30	0.24	24.00	0.00
50%	3.00	117.00	72.00	23.00	30.50	32.00	0.37	29.00	0.00
75%	6.00	140.25	80.00	32.00	127.25	36.60	0.63	41.00	1.00
max	17.00	199.00	122.00	99.00	846.00	67.10	2.42	81.00	1.00

El cuadro anterior nos muestra los descriptores estadisticos que resumen la tendencia central, dispersion y forma de la distribución del dataset.

No considera los valores nulos (**NaN**). Trabaja solo con valores numéricos. No considera tampoco valores categóricos.

- **Count** → El conteo de ocurrencia de cada descriptor.
- **Mean** → La media de cada descriptor.
- **Std** → La desviación dstandar de cada descriptor.
- **Min** → Los valores mínimos de cada descriptor
- **25%** → Cuartil Q1 de cada descriptor.
- **50%** → Cuartil Q2 de cada descriptor.
- **75%** → Cuartil Q3 de cada descriptor.
- **Max** → Los valores máximos de cada descriptor.

Por ejemplo, los cuartiles son muy útiles también para detectar **Outliers.**

2.- Integración y selección de los datos del dataframe

En este dataset vamos a utilizar todos los descriptores pues ya han sido filtrado y no son muchas las filas con las que disponemos. Sin embargo vamos a verificar cómo se distribuyen las clases.

```
In [34]: Clases = df.groupby('Clase').size()

In [35]: Clases

Out[35]: Clase
         0    500
         1    268
         dtype: int64
```

La operación anterior nos indica que del total de filas (768), **500** de ellas son de la **Clase 0** → Sin diabetes y **268** de ellas son de la **Clase 1** → Con diabetes. Practicamente la **Clase 0** es el doble de la **Clase 1**. Vamos tambien a obtener el coeficiente de correlación el cual nos muestra cómo se relacionan los descriptores entre si. Se utilizará el coeficiente de correlación de **Pearson**. Estamos asumiendo que los datos son normales. Más adelante veremos si eso es verdad.

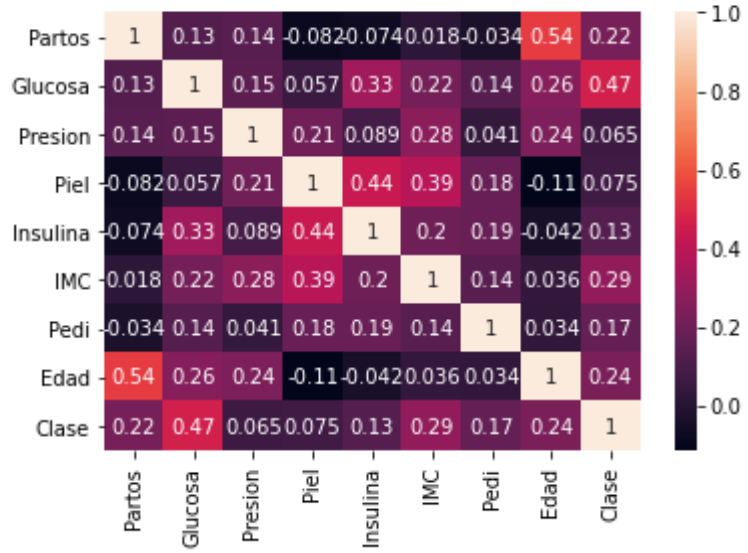
```
In [36]: Pearson = df.corr(method='pearson')

In [37]: Pearson
```

Out[37]:

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
Partos	1.00	0.13	0.14	-0.08	-0.07	0.02	-0.03	0.54	0.22
Glucosa	0.13	1.00	0.15	0.06	0.33	0.22	0.14	0.26	0.47
Presion	0.14	0.15	1.00	0.21	0.09	0.28	0.04	0.24	0.07
Piel	-0.08	0.06	0.21	1.00	0.44	0.39	0.18	-0.11	0.07
Insulina	-0.07	0.33	0.09	0.44	1.00	0.20	0.19	-0.04	0.13
IMC	0.02	0.22	0.28	0.39	0.20	1.00	0.14	0.04	0.29
Pedi	-0.03	0.14	0.04	0.18	0.19	0.14	1.00	0.03	0.17
Edad	0.54	0.26	0.24	-0.11	-0.04	0.04	0.03	1.00	0.24
Clase	0.22	0.47	0.07	0.07	0.13	0.29	0.17	0.24	1.00

Podemos mostrar la correlación de una manera más clara mediante un mapa de calor.

```
In [38]: import seaborn as sns

In [39]: sns.heatmap (Pearson,annot = True)

Out[39]: <AxesSubplot:>
```



La correlación nos aporta información muy útil pues nos indica como se relacionan los descriptores entre si. Si la correlación es positiva y cercana a 1.0 significa que los descriptores están fuertemente relacionados y el aumento de uno implica el aumento del otro y vice versa. Cuando la correlación es negativa, el aumento de un descriptor implica la disminución del otro. La diagonal siempre es 1.0 pues representa la correlación del descriptor consigo mismo.

3.- **Limpieza de los datos**

Se suele limpiar los datos cuando estos presentan errores o incosistencias dentro del dataset. Es común además, normalizar el dataset para evitar que los descriptores que poseen valores muy elevados "*arrastren*" a los que tienen valores más cercano a su media y los resultados obtenidos no sean representativos. Existen varias técnicas para trabajar con datos vacíos o nulos. Se puede reemplazar estos valores con la media, mediana, valores dentro de una ventana, eliminarlos, etc. En este dataset los siguientes descriptores poseen valores de cero (0).

```
In [33]: df.describe()

Out[33]:
```

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
count	768.00	768.00	768.00	768.00	768.00	768.00	768.00	768.00	768.00
mean	3.85	120.89	69.11	20.54	79.80	31.99	0.47	33.24	0.35
std	3.37	31.97	19.36	15.95	115.24	7.88	0.33	11.76	0.48
min	0.00	0.00	0.00	0.00	0.00	0.00	0.08	21.00	0.00

Los cuales no tendrían sentido, pues no existen pacientes que tengan estos niveles en los descriptores anteriores. Es muy probable que sean errores por omisión o por incompletitud. Vamos a reemplazarlos por **NaN**

que son más manejables y fáciles de reemplazar en un DataFrame de pandas.

```
In [40]: df_aux = df.copy(deep = True)
         df_aux[['Glucosa','Presion','Piel','Insulina','IMC']] =
         df_aux[['Glucosa','Presion','Piel','Insulina','IMC']].replace(0,np.NaN)
```

```
In [41]: df_aux
```

Out[41]:

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
0	6	148.0	72.0	35.0	NaN	33.6	0.63	50	1
1	1	85.0	66.0	29.0	NaN	26.6	0.35	31	0
2	8	183.0	64.0	NaN	NaN	23.3	0.67	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.17	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.29	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101.0	76.0	48.0	180.0	32.9	0.17	63	0
764	2	122.0	70.0	27.0	NaN	36.8	0.34	27	0
765	5	121.0	72.0	23.0	112.0	26.2	0.24	30	0
766	1	126.0	60.0	NaN	NaN	30.1	0.35	47	1
767	1	93.0	70.0	31.0	NaN	30.4	0.32	23	0

768 rows × 9 columns

Podemos obtener un estadístico de este reemplazo.

```
In [50]: Resumen_NaN = df_aux.isnull().sum()
```

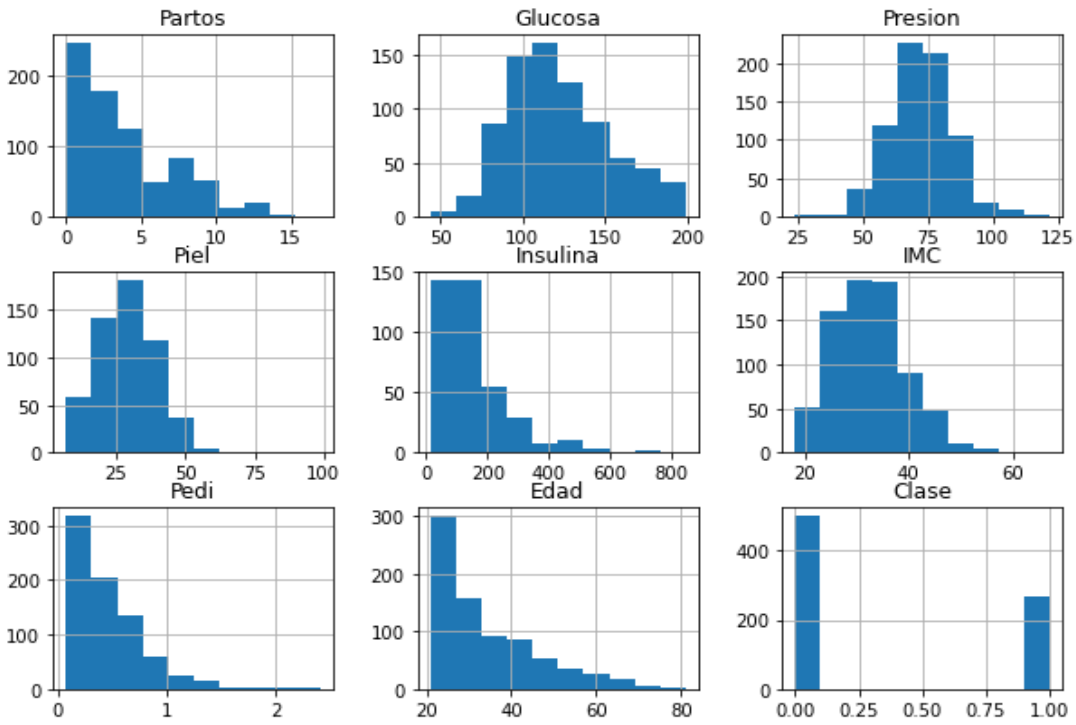
```
In [51]: Resumen_NaN
```

```
Out[51]: Partos        0
         Glucosa       5
         Presion      35
         Piel        227
         Insulina    374
         IMC          11
         Pedi          0
         Edad          0
         Clase         0
         dtype: int64
```

Este resumen nos servirá para poder analizar la distribución de los datos de los descriptores que poseen valores **NaN** para reemplazarlos más adelante por algún estadístico más adecuado.

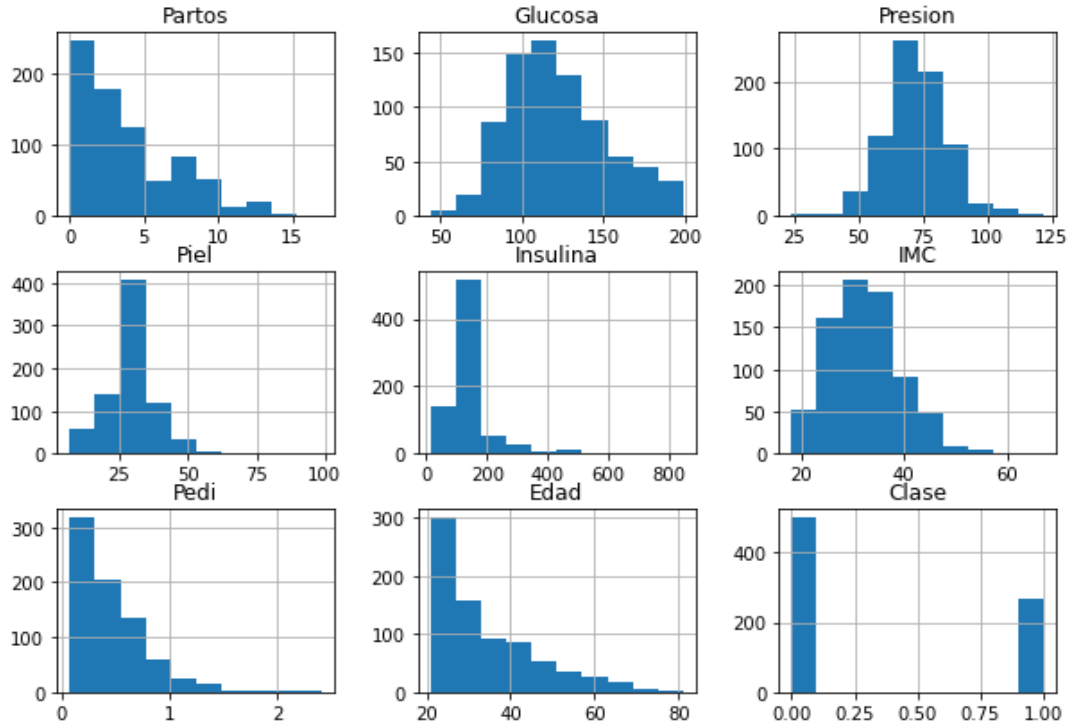
Ahora obtenemos el histograma de cada uno de los descriptores para ver sus distribuciónes sobre los datos.

```
In [57]: Distribuciones = df_aux.hist(figsize = (10,7))
```



Ahora obtenemos las nuevas distribuciones de los descriptores con **NaN** reemplazados con su media y mediana.

```
In [59]: Distribuciones_ajustadas = df_aux.hist(figsize = (10,7))
```



Estas nuevas distribuciones se obtuvieron de la siguiente manera.

```
In [58]: df_aux['Glucosa'].fillna(df_aux['Glucosa'].mean(), inplace = True)
         df_aux['Presion'].fillna(df_aux['Presion'].mean(), inplace = True)
         df_aux['Piel'].fillna(df_aux['Piel'].median(), inplace = True)
         df_aux['Insulina'].fillna(df_aux['Insulina'].median(), inplace = True)
         df_aux['IMC'].fillna(df_aux['IMC'].median(), inplace = True)
```

Ahora los drescriptores señalados que tenían valores **NaN** fueron reemplazados por su descriptor estadístico de tendencia central como la media y mediana los cuales son más idones y representativos para cada una de sus distribuciones. En el siguiente dataframe se resumen los nuevos valores.

```
In [60]: df_aux
```

Out[60]:

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
0	6	148.0	72.0	35.0	125.0	33.6	0.63	50	1
1	1	85.0	66.0	29.0	125.0	26.6	0.35	31	0
2	8	183.0	64.0	29.0	125.0	23.3	0.67	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.17	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.29	33	1
...	...	...	...	...	...	...	...	...	...
763	10	101.0	76.0	48.0	180.0	32.9	0.17	63	0
764	2	122.0	70.0	27.0	125.0	36.8	0.34	27	0
765	5	121.0	72.0	23.0	112.0	26.2	0.24	30	0
766	1	126.0	60.0	29.0	125.0	30.1	0.35	47	1
767	1	93.0	70.0	31.0	125.0	30.4	0.32	23	0

768 rows × 9 columns

Estas operaciones podrían ser consideradas como una forma de manejar los valores extremos u *Outliers*, pues valores de cero (0) sobre Glucosa, Presión, Piel, Insulina e IMC son extraños.

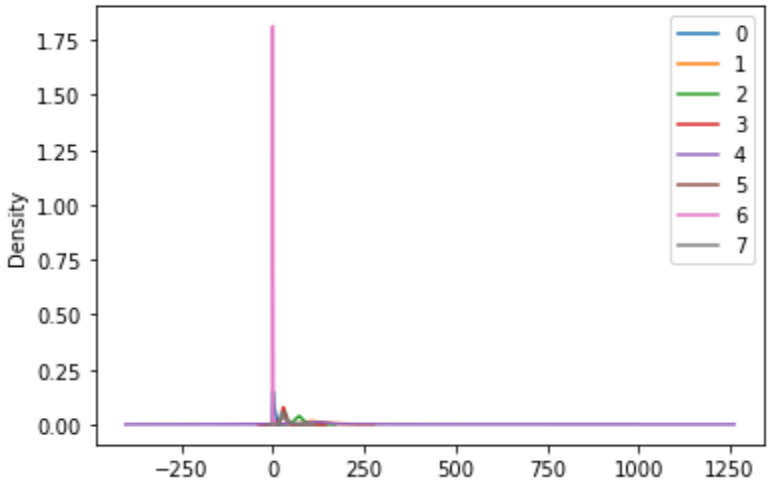
Podemos apreciar de mejor manera el dataset si hacemos un analisis de sus datos mediante la gráfica de su función de densidad. Una función de densidad de una variable aleatoria continua describe la probabilidad relativa de que esa variable tome un valor determinado. La función de densidad nos da una panorámica general y estadistica de como se distribuyen los datos a estudiar.

```
In [91]: # Grafiquemos la función de densidad para este DataSet
         # DataFrame -> Array Numpy, dejamos fuera columna de Clase
         aDF = df_aux.values[:,:-1]
```

```
In [92]: aDF

Out[92]: array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
                [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
                [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
                ...,
                [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
                [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
                [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Si calculamos la función de densidad sobre el dataset actual obtenemos el siguiente gráfico

```
In [94]: aDF = pd.DataFrame(aDF)
         aDF.plot(kind='density')

Out[94]: <AxesSubplot:ylabel='Density'>
```



Donde claramente no se aprecia muy bien, pues los datos estás sesgados por los valores más altos del descriptor. Por lo tanto, debemos normalizar los datos para que estén en la misma escala.

```
In [98]: # La funcion de densidad no se aprecia correctamente pues
         # el DataFrame no está normalizado.
         # Normalizemos los datos y veamos que sucede.

         from sklearn import preprocessing

         aDF = df_aux.values[:,:-1]
         normal = preprocessing.MinMaxScaler()
         aScaler = normal.fit_transform(aDF)
```

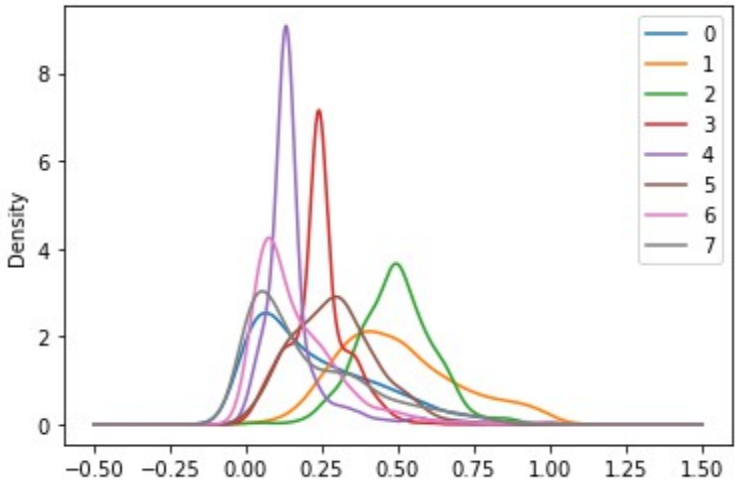
Ahora que el dataset está normalizado entonces obtenemos la función de densidad para cada descriptor.

```
In [99]: # Ahora el data frame está normalizado entre (0,1)
         aScaler
```

```
Out[99]: array([[0.35294118, 0.67096774, 0.48979592, ..., 0.31492843, 0.23441503,
                 0.48333333],
                [0.05882353, 0.26451613, 0.42857143, ..., 0.17177914, 0.11656704,
                 0.16666667],
                [0.47058824, 0.89677419, 0.40816327, ..., 0.10429448, 0.25362938,
                 0.18333333],
                ...,
                [0.29411765, 0.49677419, 0.48979592, ..., 0.16359918, 0.07130658,
                 0.15      ],
                [0.05882353, 0.52903226, 0.36734694, ..., 0.24335378, 0.11571307,
                 0.43333333],
                [0.05882353, 0.31612903, 0.46938776, ..., 0.24948875, 0.10119556,
                 0.03333333]])
```

```
In [101]: aDF = pd.DataFrame(aScaler)
          aDF.plot(kind='density')
```

```
Out[101]: <AxesSubplot:ylabel='Density'>
```

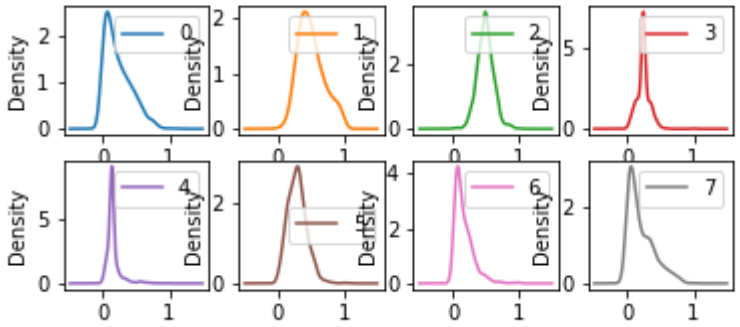


Como se aprecia el gráfico anterior, ahora cada descriptor muestra su propia función de densidad, la cual nos ayuda a ver si la muestra de cada uno de ellos proviene de una población normal.

Donde los valores siguientes representan: **0** → Parto, **1** → Glucosa, **2** → Presion, **3** → Piel, **4** → Insulina, **5** → IMC, **6** → PEDI y **7** → Edad.

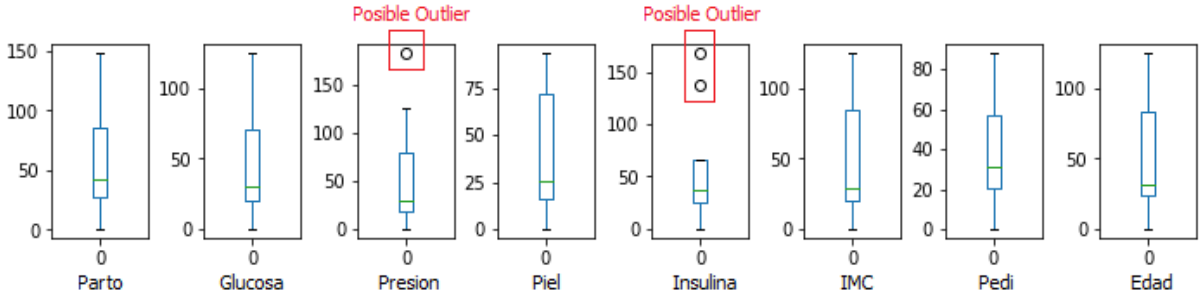
En la siguiente gráfica se muestran las funciones de densidad para cada descriptor de manera por separado.

```
In [104]: aDF = pd.DataFrame(aScaler)
          aDF.plot(kind='density',subplots=True, layout=(3,4), sharex=False)

Out[104]: array([[<AxesSubplot:ylabel='Density'>, <AxesSubplot:ylabel='Density'>,
                  <AxesSubplot:ylabel='Density'>, <AxesSubplot:ylabel='Density'>],
                 [<AxesSubplot:ylabel='Density'>, <AxesSubplot:ylabel='Density'>,
                  <AxesSubplot:ylabel='Density'>, <AxesSubplot:ylabel='Density'>],
                 [<AxesSubplot:ylabel='Density'>, <AxesSubplot:ylabel='Density'>,
                  <AxesSubplot:ylabel='Density'>, <AxesSubplot:ylabel='Density'>]],
                dtype=object)
```



Otra forma de visualizar rapidamente los valores extremos es mediante la utilización de **diagramas de bloque**.

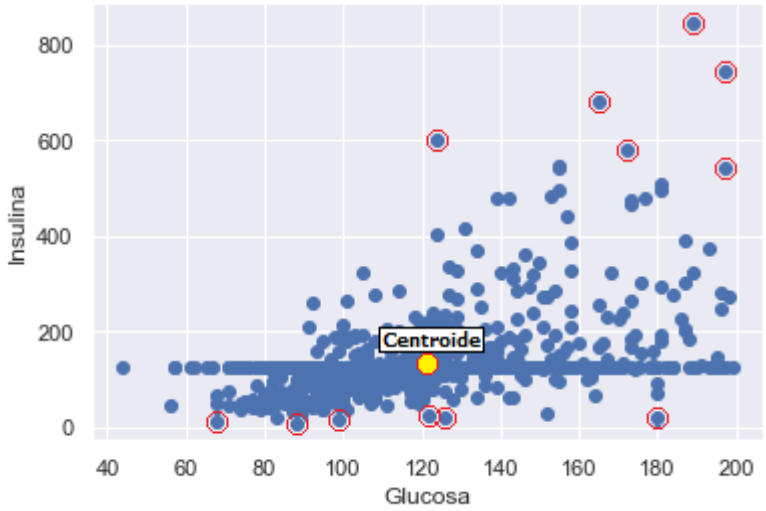
```
In [133]: # Verificar si existen Outliers en dataset original. Dejo fuera la Columna Clase (0/1)
          for nDataSet in range(0,8):
           aDF = df_aux.values[nDataSet,:-1]
           aDF = pd.DataFrame(aDF)
           aDF.plot(kind= 'box',figsize=(1,2))
```



Aunque los *Outliers* del descriptor **Presión** e **Insulina** se reflejan en el diagrama de bloques, existen otros que se asumen valores correctos (los valores de cero) los cuales obviamente son valores fuera de rango, pues nadie tiene presión sanguinea cero (estaría muerto).

Existe otra forma también de analizar graficamente posibles valores extremos (*Outlier*). Es mediante la gráfica del tipo *scatter* en pandas. En esta gráfica se relacional espacialmente dos descriptores los cuales presentan alguna distribución interesante. En caso de *scatte*r sobre los descriptores de (**Glucosa,Insulina**) por ejemplo, algunos de los nodos resaltados en rojos son candidatos a ser considerados *Outliers*, por encontrase bastante lejos del centroide de la distribución analizada. Esto se puede apreciar en la siguiente gráfica.

```
In [141]: import seaborn
          seaborn.set()
          plt.scatter(df_aux['Glucosa'],df_aux['Insulina'])
          plt.xlabel('Glucosa')
          plt.ylabel('Insulina')

Out[141]: Text(0, 0.5, 'Insulina')
```



Claramente, los nodos rojos cerca de valores de cero (0) son valores extremos, pues nadie tiene gluocosa e insulina en esos niveles, a no ser que sea una persona muy enferma. Esta técnica tambien se puede lograr aplicando el algoritmo no supervisado de Kmeans, pues se pueden apreciar como se distribuyen los datos espacialmente según los centroides (parámetro de k).

4.- **Análisis de los datos**

Respecto de la selección de los grupos de datos a analizar se trabajará sobre el dataset normalizado para evitar sesgos en los descriptores. Para la comprobación de la normalidad se probarán tres técnicas mediante las pruebas de graficación de cuantiles teóricos (Q-Q Plot), Shapiro-Wilk y D' Agostino K-Squared.

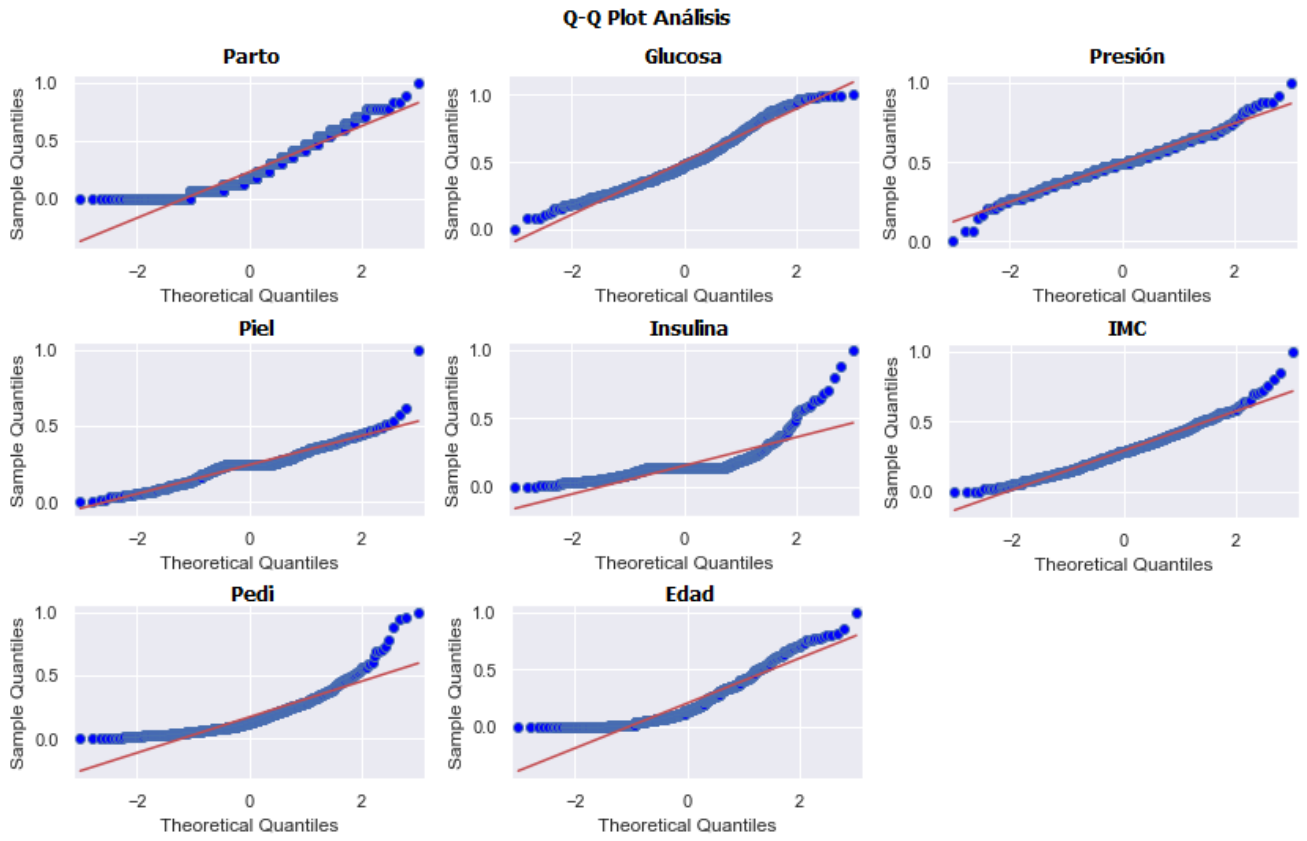
- **Cuantiles teóricos (Q-Q Plot)** → Aqui las muestras se dividen en grupos, llamados cuantiles. Cada punto de datos de la muestra se empareja con un miembro similar de la distribución con la que comparamos en la misma distribución de acumulación. Los puntos resultantes se trazan como un diagrama de dispersión.

- **Shapiro-Wilk** → Este es es un ejemplo de test de hipótesis para saber si una distribución sigue la forma de otra distribución. *Shapiro-Wilk* es una prueba bastante buena para comprobar la normalidad de una variable. Se suele utilizar con muestras de datos pequeñas tal como el dataset que estamos utilizando.

- **D' Agostino K-Squared** → En esta prueba se calcula la curtosis y asimetría a partir de los datos para determinar si la distribución de datos se aparta de la distribución normal.

Al plicar **Q-Q Plot** sobre el dataset se obtiene lo siguiente.

In [193]:
```
# Como probar si una distribución de datos es normal
from numpy.random import seed, randn
from statsmodels.graphics.gofplots import qqplot
# Configuro la semilla aleatoria
seed(1993)
for nDe in range(8): # son 8 los descriptores que hay que analizar
    aDF = pd.DataFrame(aScaler) # DataSet Normalizado
    aData = aDF.values[0::,nDe]
    with plt.rc_context():
        plt.rc("figure", figsize=(4,2))
        qqplot(aData,line='s')
plt.show()
```



En este análisis se puede observar que los descriptores de *Glucosa, Presión Piel* e *IMC* siguen una distribución normal pues la gran mayoría de sus puntos se distribuyen sobre la línea roja. Sin embargo, los descriptores de *Parto, Insulina, Pedi* y *Edad* podrían no provenir de una muestra normal.

Sin embargo, debido a que muchos puntos al inicio consideran el valor cero o muy cercano a el, los descriptores de *Pedi* y *Edad* podrian ser considerados normales, descontando obviamente los valores extremos que son candidatos a ser *Outlier.*

Al aplicar las otras pruebas de **Shapiro-Wilk** y **D' Agostino K-Squared** resultaron que el dataset no es normal.

In [210]:
```
# Prueba de Shapiro-Wilk
from numpy.random import seed, randn
from scipy.stats import shapiro

seed(1993)
for nDe in range(8): # son 8 los descriptores que hay que analizar
    aDF = pd.DataFrame(aScaler)
    aData = aDF.values[0::,nDe]
    # Prueba de Shapiro-Wilk
    stat, p = shapiro(aData)
    print('Estadisticos = %.3f, p = %.3f' % (stat,p))
    # Interpretación
    alpha = 0.05
    if p > alpha:
       print('La muestra parece Gaussiana (no se rechaza la hipótesis nula H0')
    else:
       print('La muestra no parece Gaussiana(se rechaza la hipótesis nula H0')
```

In [207]:
```
# Prueba de D' Agostino K-Squared
from numpy.random import seed, randn
from scipy.stats import normaltest

for nDe in range(8): # son 8 los descriptores que hay que analizar
    aData = df_aux.values[0::,nDe]
    # Prueba de D' Agostino K-Squared
    stat, p = normaltest(aData)
    print('Estadisticos = %.3f, p = %.3f' % (stat,p))
    # Interpretación
    alpha = 0.05
    if p > alpha:
       print('La muestra parece Gaussiana (no se rechaza la hipótesis nula H0')
    else:
       print('La muestra no parece Gaussiana (se rechaza la hipótesis nula H0')
```

Respecto de la homogeneidad (homocedasticidad) se considera que la varianza es constante en los diferentes niveles de un factor, es decir, entre diferentes grupos. En los modelos de regresión lineal, esta condición de homocedasticidad suele hacer referencia a los errores (residuos) del modelo, es decir, que la varianza de los errores es constante en todas las predicciones.

En este caso, no se debe aplicar el analisis de varianza (homogeneidad) debido a que no hay garantía que la muestra provenga de una población con distribución normal. De hecho de los tres test anteriores, dos de ellos concluyeron que el dataset no sigue una distribución normal.

Sin embargo para aplicar lo aprendido en el **Bloque 3** voy a utilizar un dataset muy utilizado para fines didacticos que sigue una distribución normal y que permitirá hacer el análisis de homogeneida de la varianza para aplicar lo aprendido. El dataset se puede obtener de https://www.kaggle.com/uciml/iris. Se trata de un estudio de tres tipos diferentes de plantas del tipo iris, las cuales se pueden clasificar como setosa, versicolor y virginica. Es un dataset de 150 regristros donde cada una de las plantas aparece cicnuenta veces. Veamos su descripción con un pandas.

```
In [231]: iris = pd.read_csv('iris.csv')

In [232]: iris = pd.DataFrame(iris)

In [235]: iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             150 non-null    int64
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    int64
dtypes: float64(4), int64(2)
memory usage: 7.2 KB
```

Eliminaremos el descriptor '*Id*' para mayor comodidad en el procesamiento y ahora el dataset queda asi.

```
In [238]: iris.drop('Id',axis=1)
```

Out[238]:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

Donde el descriptor 'Species' corresponde al tipo de planta iris: 0 → setosa, 1 → versicolor y 2 → virginica. Los otros descriptores son los atributos de cada una de estas plantas los cuales permitiran clasificar su tipo.

Para hacer la prueba de la homegeneidad de la varianza crearemos dos DataFrame. También realizaremos

algunas operaciones de filtrado.

```
In [259]: setosa = iris[iris.Species == 0]     # DataFrame planta iris->sotosa
          versicolor = iris[iris.Species == 1] # DataFrame planta iris->versicolor
```

```
In [263]: setosa.head()
```

Out[263]:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Out[265]:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
50	7.0	3.2	4.7	1.4	1
51	6.4	3.2	4.5	1.5	1
52	6.9	3.1	4.9	1.5	1
53	5.5	2.3	4.0	1.3	1
54	6.5	2.8	4.6	1.5	1

Queremos determinar si existen diferencias significativas en el ancho del sépalo (SepalWidthCm) entre las plantas setosa y versicolor. Para ello, vamos a agrupar por especies (Clases = Species = 0,1,2).

```
In [267]: iris.groupby('Species')['SepalWidthCm'].describe()
```

Out[267]:

Species	count	mean	std	min	25%	50%	75%	max
0	50.0	3.42	0.38	2.3	3.12	3.4	3.68	4.4
1	50.0	2.77	0.31	2.0	2.52	2.8	3.00	3.4
2	50.0	2.97	0.32	2.2	2.80	3.0	3.18	3.8

Ahora haremos la prueba de homogeneidad de las varianzas. Se utilizará la prueba de **Levene**. El resultado se muestra a continuación.

```
In [270]: from scipy import stats

In [271]: stats.levene(setosa['SepalWidthCm'],versicolor['SepalWidthCm'])
Out[271]: LeveneResult(statistic=0.6635459332943233, pvalue=0.41728596812962016)
```

El cual nos muestra que la prueba no es significativa, es decir, no hay diferencias significativas en las varianzas de ambas muestras de setosa y versicolor. Es decir, la hipótesis nula no se rechaza (H_0).

Podemos además hacer otras pruebas de normalidad mediante el test de **Shapiro**.

```
In [273]: stats.shapiro(setosa['SepalWidthCm'])
Out[273]: ShapiroResult(statistic=0.968691885471344, pvalue=0.20465604960918427)

In [274]: stats.shapiro(versicolor['SepalWidthCm'])
Out[274]: ShapiroResult(statistic=0.9741330742835999, pvalue=0.33798879384994507)
```

El cual nos indica que ninguna de las variables anteriores viola el principicio de normalidad.

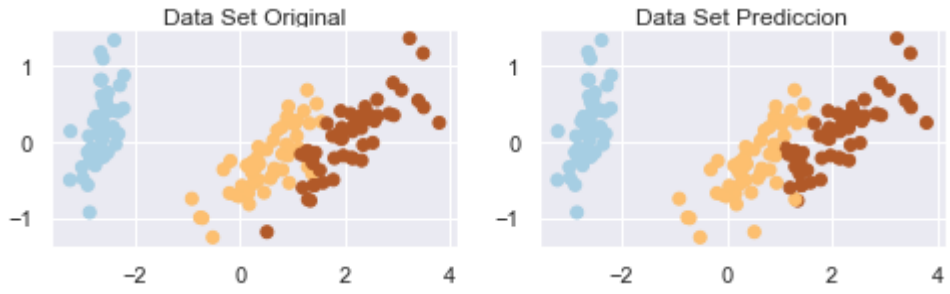
Ahora extraeremos información util de los dos dataset presentados en esta práctica. Comenzaremos con el datset de iris por ser un clásico y que me ha permitido practicar lo aprendido en esta práctica. Una de las operaciones más utilizadas en machine learning es la clasificación y regresión. Vamos a utilizar el método de **KNN** para clasificar y predecir la pertenencia de una muestra a sus clases de setosa, versicolor o virginica.

```
In [286]: X = iris.values[:,:-1] # Separamos los descriptores

In [288]: y = iris.values[:,4] # Separamos las Clases

In [290]: len(y) # Verificamos tamaño dataset
Out[290]: 150

In [296]: # Vamos a probar que KNN clasifica 100% bien con todos el set de prueba
          # despues haremos una prediccion con KNN (k=5)
          iris_knn = KNeighborsClassifier(n_neighbors=1).fit(X,y)

In [297]: y_pred = iris_knn.predict(X)

In [298]: print(np.all(y_pred==y))
          True

In [299]: iris_knn.score(X,y)
Out[299]: 1.0
```

Utilizamos **KNN (k=1 vecino)**. Si bien es cierto esto es una mala idea pues estamos considerando todos los primeros vecinos más cercanos, seguro produce overfitting (sobre entrenamiento). Sin embargo lo hacemos para

comprobar que las predicciones de todo el dataset (X) tiene un 100% de clasificación. Ahora probaremos con KNN (K=5 vecinos).

```
In [301]: # Ahora probamos cin KNN (k=5) vecinos
          iris_knn = KNeighborsClassifier(n_neighbors = 5).fit(X,y)

In [302]: iris_knn
Out[302]: KNeighborsClassifier()

In [303]: iris_knn.score(X,y)
Out[303]: 0.9666666666666667

In [304]: y_pred=iris_knn.predict(X)

In [305]: y_pred
Out[305]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
                 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
                 1., 1., 2., 1., 2., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 1.,
                 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2.,
                 2., 2., 2., 2., 1., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
                 1., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
                 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])

In [322]: # Vamos a utilizar PCA para hacer una reducción a 2D de los descriptores
          from sklearn.decomposition import PCA

In [323]: X_2D = PCA(2).fit_transform(X)

In [327]: fig,ax = plt.subplots(1,2,figsize=(8,2))
          ax[0].scatter(X_2D[:,0], X_2D[:,1], c=y, cmap=plt.cm.Paired)
          ax[0].set_title('Data Set Original')
          ax[1].scatter(X_2D[:,0], X_2D[:,1], c=y_pred, cmap=plt.cm.Paired)
          ax[1].set_title('Data Set Prediccion')
          plt.show()
```



Aquí se puede apreciar que **KNN** hizo una predicción al 100%, pues etiquetó correctamente las clases de los descriptores. Pero hizimos trampa pues hay overfitting. Es decir, el clasificador sobre aprendió las caracteristicas de este dataset y no puede generalizar correctamente con nuevos dataset de test. Lo que vamos haver ahora, es

hacer lo sugerido en las lecturas. Vamos a separar el dataset en entrenamiento y de test para luego aplicar el clasificador con los datos de prueba.

```
In [332]: from sklearn.model_selection import train_test_split

In [333]: X_train, X_test, y_train, y_test = train_test_split(X,y)

In [336]: X_train[0:5], X_test[0:5], y_train[0:5], y_test[0:5]

Out[336]: (array([[5.2, 3.4, 1.4, 0.2],
                  [5.7, 2.9, 4.2, 1.3],
                  [5.5, 2.6, 4.4, 1.2],
                  [6.7, 3. , 5.2, 2.3],
                  [6.9, 3.1, 5.4, 2.1]]),
           array([[5.7, 2.5, 5. , 2. ],
                  [6.7, 2.5, 5.8, 1.8],
                  [6.4, 3.1, 5.5, 1.8],
                  [5.4, 3.7, 1.5, 0.2],
                  [5. , 3.6, 1.4, 0.2]]),
           array([0., 1., 1., 2., 2.]),
           array([2., 2., 2., 0., 0.]))
```

Ahora tenemos separados los dataset para entrenamiento y pruebas.

```
In [337]: X_train.shape, X_test.shape

Out[337]: ((112, 4), (38, 4))
```

Ahora si probamos KNN(k=5) veamos como clasifica los dataset de prueba.

```
In [338]: iris_knn = KNeighborsClassifier(n_neighbors=5).fit(X_train,y_train)
          y_pred = iris_knn.predict(X_test)
          y_train_pred = iris_knn.predict(X_train)

In [340]: # n_neighbors=5, Training cross-validation score
          iris_knn.score(X_tetrain,y_train)

Out[340]: 0.9553571428571429

In [341]: # n_neighbors=5 Test cross-validation score
          iris_knn.score(X_test,y_test)

Out[341]: 1.0
```

Ahora tenemos los scores de croos-validation de data Train y Test. Los resultados son muy robustos, 95% y 100% cada uno de ellos.

Ahora, habiendo realizado algunas operaciones de análisis de datos sobre el dataset de iris, trabajaremos sobre el dataset de diabetes.

In [349]:
```
#DataFrame diabetes original sin normalizar
aData = df_aux
```

In [356]:
```
aData.head()
```

Out[356]:

	Partos	Glucosa	Presion	Piel	Insulina	IMC	Pedi	Edad	Clase
0	6	148.0	72.0	35.0	125.0	33.6	0.63	50	1
1	1	85.0	66.0	29.0	125.0	26.6	0.35	31	0
2	8	183.0	64.0	29.0	125.0	23.3	0.67	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.17	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.29	33	1

In [351]:
```
X = df_aux.values[:,:-1] # Separamos los descriptores
```

In [352]:
```
y = df_aux.values[:,8] # Separamos las Clases
```

In [359]:
```
X
```

Out[359]:
```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

In [355]:
```
y[0::10] # mostramos solo las 10 primeras clases...
```

Out[355]:
```
array([1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 1., 1., 0., 0., 0.,
       1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0.,
       0., 0., 1., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0.,
       1., 0., 0., 1., 0., 1., 0., 1., 1., 0., 0., 0., 1., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 1., 1., 1., 0.])
```

Ahora separamos el data set de diabetes en dataset de entrenamiento y de test.

In [360]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y)
```

In [361]:
```
X_train.shape, X_test.shape
```

Out[361]:
```
((576, 8), (192, 8))
```

Al aplicar KNN(k=3) obtenemos.

```
In [362]: diabetes_knn = KNeighborsClassifier(n_neighbors=3).fit(X_train, y_train)
          y_pred = diabetes_knn.predict(X_test)
          y_train_pred = diabetes_knn.predict(X_train)
```

```
In [365]: y_pred
```

```
Out[365]: array([0., 0., 1., 0., 0., 1., 1., 0., 1., 1., 0., 1., 0., 0., 1., 0., 1.,
                 1., 0., 1., 1., 1., 0., 0., 0., 1., 1., 0., 1., 1., 0., 0., 0., 1.,
                 1., 0., 0., 0., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0.,
                 1., 0., 0., 1., 1., 0., 1., 1., 1., 0., 0., 0., 0., 1., 0., 0., 0.,
                 0., 1., 1., 0., 1., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0.,
                 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
                 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0.,
                 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 1., 1.,
                 1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0.,
                 0., 0., 1., 0., 1., 1., 0., 1., 1., 1., 0., 0., 0., 0., 1., 0., 1.,
                 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0.,
                 0., 1., 1., 0., 0.])
```

Obtenemos el siguiente rendimiento.

```
In [366]: diabetes_knn.score(X_train, y_train)
Out[366]: 0.8385416666666666

In [367]: diabetes_knn.score(X_test, y_test)
Out[367]: 0.7291666666666666
```


```
In [366]: diabetes_knn.score(X_train, y_train)
Out[366]: 0.8385416666666666

In [367]: diabetes_knn.score(X_test, y_test)
Out[367]: 0.7291666666666666
```


```
In [370]: print(metrics.classification_report(y_test, y_pred, target_names=['No Diabetes', 'Diabetes']))
```

	precision	recall	f1-score	support
No Diabetes	0.78	0.81	0.79	123
Diabetes	0.63	0.58	0.61	69
accuracy			0.73	192
macro avg	0.71	0.70	0.70	192
weighted avg	0.72	0.73	0.73	192

En este resumen se aprecian las métricas de precision, recall y F1-score para las pruebas anteriores

Este resumen nos muestra que el dataset (dabietes.csv) analizado posee dos clases → Diabetes y No Diabetes. El dataset de **test** está compuesto por 192 filas (123 → No Diabetes y 69 → Diabetes). La *precision* nos indica cuantos filas están correctamente clasificadas dentro de su clase → 78% → (0.78 x 123) → 96 filas para No Diabetes y 63% → (0.63 x 69) → 44 filas para Diabetes. El *recall* nos indica cuantos elementos de esta clase se encuentran en el numero total de elemenstos de su clase. F1-score es la media armónica entre *precision y recall*. Y support es el número de ocurrencias de la clase en el dataset. Asi en esta prueba de clasificación, hemos obtenido un score de un 83% de precisión en el entrenamiento y un 73% de precisión en el test.

Si aplicamos Kmean, obtenemos lo siguiente.

```
In [374]: X_train, X_test, y_train, y_test = train_test_split(X,y)

In [375]: kmeans = KMeans(n_clusters=2, random_state=0).fit(X_train)

In [378]: x_pred = kmeans.predict(X_test)
          #y_pred = kmeans.predict(X_test)

In [379]: x_pred

Out[379]: array([0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [380]: print(metrics.classification_report(x_pred, y_pred, target_names=['No Diabetes', 'Diabetes']))

                precision    recall  f1-score   support

    No Diabetes       0.95      0.67      0.78       182
       Diabetes       0.05      0.30      0.08        10

       accuracy                           0.65       192
      macro avg       0.50      0.49      0.43       192
   weighted avg       0.90      0.65      0.75       192
```

Y para terminar, se calcula la regresión lineal y sus métricas de *mse* y *r2score*.

```
In [381]: from sklearn.linear_model import LinearRegression

In [397]: model = LinearRegression()
          model.fit(X_train,y_train)

Out[397]: LinearRegression()

In [403]: prediccion = model.predict(X_test)

In [399]: prediccion[0:5] # los 5 primeros...

Out[399]: array([0.86988647, 0.2725784 , 0.35930825, 0.62239398, 0.34165801])

In [386]: from sklearn.metrics import mean_squared_error as mse
          from sklearn.metrics import r2_score

In [407]: # Calculo de estadisticos
          print(mse(y_test,prediccion)) # Error Medio Cuadratico
          r2_score(y_test,prediccion)   # R2 score

          0.1348961071320479

Out[407]: 0.40201898829776184
```

El error medio cuadrático es bajo pero el coeficiente de determinación es casi un 40% lo cual nos dice que las predicciones para este modelo no son tan robustas.

Conclusiones

En este trabajo se ha puesto en práctica lo planteado en el bloque 3 (limpieza y análisis de los datos) donde se han utilizado las dataset de prueba diabetes.csv e iris.csv. Se han programado y aplicado las fases de limpieza, filtrado y selección de datos para trabajar con ellos de manera más adecuada. En el datset de diabetes se intentó encontrar alguna relación de depencia y su correlación entre sus descriptores para hecer algunas predicciones y obtener sus clases utilizando la librería de sklearn de python. Además, se resaltó el hecho que la normalidad de los datos es crucial para hacer un robusto análisis junto con un correcto manejo de los valores extremos. En el caso del dataset de iris.csv su utilización fue más bien pedagógica y didactica para poder aplicar algunas operaciones de analisis de varianza y normalidad. El resultado de la aplicación de las operaciones de regresión y clasificación nos demuuestran que es posible hacer algunas generalizaciones con estos dataset pero que el proceso de normalización, manejo de valores extremos y limpieza son cruciales y afectan los resultados fuertemente. Además de utilizar correctamente el tipo de los datos para los análisis pues los resultados pueden ser diferentes si los datos son del tipo continuos o categóricos. La normalidad de los datos tambien es importante pues por ejemplo la técnica de KNN es muy sensible a las magnitudes de los descriptores en cambio en otros métodos basados en arboles de decisión no es relevante. En resumen, creo que falta mucho por aprender y probar sobre este apasionante tema de data science.

Referencias

- Material de lectura bloque 3: Limpieza y análisis de los datos
- Machine Learning: SkLearn Python
- Internet

ANEXO

Mi trabajo (códigos)

```
# By Alberto Caro
# Practica 2
#------------------------------------------------------------
import numpy as np
import pandas as pd

ds = pd.read_csv('diabetes.csv')
df = pd.DataFrame(ds)
df.columns=['Partos','Glucosa','Presion','Piel','Insulina','IMC','Pedi','Edad','Clase']

df.describe()

from pandas import set_option

set_option('display.width', 100)
set_option('precision', 2)

df.describe()

Clases = df.groupby('Clase').size()

Pearson = df.corr(method='pearson')

import seaborn as sns

sns.heatmap (Pearson,annot = True)

df_aux = df.copy(deep = True)
df_aux[['Glucosa','Presion','Piel','Insulina','IMC']] =
df_aux[['Glucosa','Presion','Piel','Insulina','IMC']].replace(0,np.NaN)

df_aux

Resumen_NaN = df_aux.isnull().sum()

Distribuciones = df_aux.hist(figsize = (10,7))

df_aux['Glucosa'].fillna(df_aux['Glucosa'].mean(), inplace = True)
df_aux['Presion'].fillna(df_aux['Presion'].mean(), inplace = True)
df_aux['Piel'].fillna(df_aux['Piel'].median(), inplace = True)
df_aux['Insulina'].fillna(df_aux['Insulina'].median(), inplace = True)
df_aux['IMC'].fillna(df_aux['IMC'].median(), inplace = True)


Distribuciones_ajustadas = df_aux.hist(figsize = (10,7))

# Grafiquemos la función de densidad para este DataSet
# DataFrame -> Array Numpy, dejamos fuera columna de Clase

aDF = df_aux.values[:,:-1]
aDF = pd.DataFrame(aDF)
```

```
aDF.plot(kind='density')

# La funcion de densidad no se aprecia correctamente pues
# el DataFrame no está normalizado.
# Normalizemos los datos y veamos que sucede.

from sklearn import preprocessing

aDF = df_aux.values[:,:-1]
normal = preprocessing.MinMaxScaler()
aScaler = normal.fit_transform(aDF)

aDF = pd.DataFrame(aScaler)
aDF.plot(kind='density',subplots=True, layout=(3,4), sharex=False)

# Verificar si existen Outliers en dataset original. Dejo fuera la Columna Clase (0/1)
for nDataSet in range(0,8):
 aDF = df_aux.values[nDataSet::,:-1]
 aDF = pd.DataFrame(aDF)
 aDF.plot(kind= 'box',figsize=(20,10))

import seaborn
seaborn.set()
plt.scatter(df_aux['Glucosa'],df_aux['Insulina'])
plt.xlabel('Glucosa')
plt.ylabel('Insulina')

import seaborn
seaborn.set()
plt.scatter(df_aux['Glucosa'],df_aux['Presion'])
plt.xlabel('Glucosa')
plt.ylabel('Presion')


# Como probar si una distribución de datos es normal
from numpy.random import seed, randn
from statsmodels.graphics.gofplots import qqplot
# Configuro la semilla aleatoria
seed(1993)
# Genero 100 muestras
data = randn(100)
# Represento el Q-Q plot
qqplot(data,line='s')
plt.show()

#aData = df_aux.values[0::,:-1]
aData = df_aux.values[0::,0] # DataSet Original sin normalización - Descriptor 0 -> Parto
aData

# Como probar si una distribución de datos es normal
from numpy.random import seed, randn
from statsmodels.graphics.gofplots import qqplot
# Configuro la semilla aleatoria
```

```
seed(1993)
for nDe in range(8): # son 8 los descriptores que hay que analizar
    aDF = pd.DataFrame(aScaler) # DataSet Normalizado
    aData = aDF.values[0::,nDe]
    with plt.rc_context():
        plt.rc("figure", figsize=(4,2))
        qqplot(aData,line='s')
plt.show()

# Prueba de Shapiro-Wilk
from numpy.random import seed, randn
from scipy.stats import shapiro

seed(1993)
for nDe in range(8): # son 8 los descriptores que hay que analizar
    aDF = pd.DataFrame(aScaler)
    aData = aDF.values[0::,nDe]
    # Prueba de Shapiro-Wilk
    stat, p = shapiro(aData)
    print('Estadisticos = %.3f, p = %.3f' % (stat,p))
    # Interpretación
    alpha = 0.05
    if p > alpha:
      print('La muestra parece Gaussiana (no se rechaza la hipótesis nula H0')
    else:
      print('La muestra no parece Gaussiana(se rechaza la hipótesis nula H0')


# Prueba de D' Agostino K-Squared
from numpy.random import seed, randn
from scipy.stats import normaltest

for nDe in range(8): # son 8 los descriptores que hay que analizar
    aData = df_aux.values[0::,nDe]
    # Prueba de D' Agostino K-Squared
    stat, p = normaltest(aData)
    print('Estadisticos = %.3f, p = %.3f' % (stat,p))
    # Interpretación
    alpha = 0.05
    if p > alpha:
      print('La muestra parece Gaussiana (no se rechaza la hipótesis nula H0')
    else:
      print('La muestra no parece Gaussiana (se rechaza la hipótesis nula H0')

iris = pd.read_csv('iris.csv')

iris = pd.DataFrame(iris)

iris.info()

iris.drop('Id',axis=1)

# Como probar si una distribución de datos es normal
```

```
from numpy.random import seed, randn
from statsmodels.graphics.gofplots import qqplot
for nDe in range(4): # son 4 los descriptores que hay que analizar
    aData = iris.values[0::,nDe]
    with plt.rc_context():
        plt.rc("figure", figsize=(4,2))
        qqplot(aData,line='s')
plt.show()

setosa = iris[iris.Species == 0]     # DataFrame planta iris->sotosa
versicolor = iris[iris.Species == 1] # DataFrame planta iris->versicolor

setosa.head()

# Como probar si una distribución de datos es normal
from numpy.random import seed, randn
from statsmodels.graphics.gofplots import qqplot
aData = setosa.values[0::,1]
with plt.rc_context():
     plt.rc("figure", figsize=(4,2))
     qqplot(aData,line='s')
plt.show()

iris = iris.drop('Id',axis=1)

setosa = iris[iris.Species == 0]     # DataFrame planta iris->sotosa
versicolor = iris[iris.Species == 1] # DataFrame planta iris->versicolor

versicolor.head()

iris.groupby('Species')['SepalWidthCm'].describe()

from scipy import stats

stats.levene(setosa['SepalWidthCm'],versicolor['SepalWidthCm'])

stats.shapiro(versicolor['SepalWidthCm'])

df_aux.info()

X = iris.values[:,:-1] # Separamos los descriptores
y = iris.values[:,4] # Separamos las Clases

len(y) # Verificamos tamaño dataset

from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

# Vamos a probar que KNN clasifica 100% bien con todos el set de prueba
# despues haremos una prediccion con KNN (k=5)
iris_knn = KNeighborsClassifier(n_neighbors=1).fit(X,y)

y_pred = iris_knn.predict(X)
```

```
print(np.all(y_pred==y))

iris_knn.score(X,y)

# Ahora probamos cin KNN (k=5) vecinos
iris_knn = KNeighborsClassifier(n_neighbors = 5).fit(X,y)

iris_knn.score(X,y)
y_pred=iris_knn.predict(X)

# Ahora probamos cin KNN (k=3) vecinos
iris_knn = KNeighborsClassifier(n_neighbors = 3).fit(X,y)

iris_knn.score(X,y)

# Vamos a utilizar PCA para hacer una reducción a 2D de los descriptores
from sklearn.decomposition import PCA

X_2D = PCA(2).fit_transform(X)

fig,ax = plt.subplots(1,2,figsize=(8,2))
ax[0].scatter(X_2D[:,0], X_2D[:,1], c=y, cmap=plt.cm.Paired)
ax[0].set_title('Data Set Original')
ax[1].scatter(X_2D[:,0], X_2D[:,1], c=y_pred, cmap=plt.cm.Paired)
ax[1].set_title('Data Set Prediccion')
plt.show()

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y)
X_train[0:5], X_test[0:5], y_train[0:5], y_test[0:5]
X_train.shape, X_test.shape

iris_knn = KNeighborsClassifier(n_neighbors=5).fit(X_train,y_train)
y_pred = iris_knn.predict(X_test)
y_train_pred = iris_knn.predict(X_train)

# n_neighbors=5, Training cross-validation score
iris_knn.score(X_train,y_train)

# n_neighbors=5 Test cross-validation score
iris_knn.score(X_test,y_test)

#DataFrame diabetes original sin normalizar
aData = df_aux

aData.head()

X = df_aux.values[:,:-1] # Separamos los descriptores
y = df_aux.values[:,8] # Separamos las Clases

y[0::10] # mostramos solo las 10 primeras clases...
```

```
X_train, X_test, y_train, y_test = train_test_split(X,y)

X_train.shape, X_test.shape

diabetes_knn = KNeighborsClassifier(n_neighbors=3).fit(X_train, y_train)
y_pred = diabetes_knn.predict(X_test)
y_train_pred = diabetes_knn.predict(X_train)

diabetes_knn.score(X_train, y_train)
diabetes_knn.score(X_test, y_test)

print(metrics.classification_report(y_test, y_pred, target_names=['No Diabetes', 'Diabetes']))

X_train, X_test, y_train, y_test = train_test_split(X,y)

kmeans = KMeans(n_clusters=2, random_state=0).fit(X_train)

x_pred = kmeans.predict(X_test)

print(metrics.classification_report(x_pred, y_pred, target_names=['No Diabetes', 'Diabetes']))

from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train,y_train)
prediccion = model.predict(X_test)

prediccion[0:5] # los 5 primeros...

from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import r2_score
# Calculo de estadisticos
print(mse(y_test,prediccion)) # Error Medio Cuadratico
r2_score(y_test,prediccion)   # R2 score
```