In [2]:
```
import numpy as np
import pandas as pd
from pandas import read_csv
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestRegressor
import seaborn as sns
import math

import statistics as stat
from sklearn.neural_network import MLPRegressor
```

Seguimos el hilo argumental del ejercicio 9.1 y ejercicio 9.2 para trabajarcon el ejercicio 10 . Primero cargamos el Data Set.

In [3]:
```
df = pd.read_csv("DelayedFlights.csv") # este es el conjunto de datos proporcionado en el
df.head(10)
```

Out[3]:

	Unnamed: 0	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier
0	0	2008	1	3	4	2003.0	1955	2211.0	2225	WN
1	1	2008	1	3	4	754.0	735	1002.0	1000	WN
2	2	2008	1	3	4	628.0	620	804.0	750	WN
3	4	2008	1	3	4	1829.0	1755	1959.0	1925	WN
4	5	2008	1	3	4	1940.0	1915	2121.0	2110	WN
5	6	2008	1	3	4	1937.0	1830	2037.0	1940	WN
6	10	2008	1	3	4	706.0	700	916.0	915	WN
7	11	2008	1	3	4	1644.0	1510	1845.0	1725	WN
8	15	2008	1	3	4	1029.0	1020	1021.0	1010	WN
9	16	2008	1	3	4	1452.0	1425	1640.0	1625	WN

10 rows × 30 columns

1. **Tratamiento de variables**.

En este apartado vamos a hacer tratamiento de variables para luego aplicar a programas de clasificación en función de la variable ArrDelay

Hemos dejado los enlaces del ejercicio 9.1 y ejercicio 9.2 en que se explica los razonamientos para sleccionar una variable un otra,

aunque volveremos a exolicar los motivos por los que se seleccionan variables, unas sí u otras no, no entraremos tan en detalle

De las notas de 9.1 y 9.2:

a) Variable Unnamed 0 y Year, básicamente son un índice y el año de vuelos del 2008. Year es una constante. Así que las eliminamos.

b) las variables "UniqueCarrier', 'FlightNum','TailNum, 'Month', 'DayofMonth', 'DayOfWeek', 'Cancelled', 'CancellationCode', "Diverted", "Origin", "Dest" tamopco tenían ningún peso de influencia en la variable ArrDelay

- UniqueCarrier es la compañía que hace el vuelo. No tiene piese alguno en la variable ArrDelay,no se puede afirmar que ninguna compañía vaya a tener más retrasos qur otra
- Flightnum(número de vuelo) y TailNum(Número de avión), que bien, sí podríam darnos información acerca de retrasos, nos generaría por Dummies centenares de variables nuevas. Ya que un avión que puede hacer X viajes en un dia o dos, nos podría indicar si va con retraso, si los vuelos anteriores van con retraso ya que no llegaría a la hora para el embarco, y suponiendo que el avión no fuera sustituído. Pero para poder hacer esta previsión se debería tener en el train la información del vuelo, de las horas anteriores o incluso del día anterior para vuelos de varia horas o que empezaron el día anterior. Se debería tener una secuencia de los vuelos anteriores, o al menos del anterior. Viendo que se necesita cierta información previa y que nos genera un exceso de variables independientes también las eliminamos.
- Los vuelos cancelados, el código de cancelación y desviados, no los ponemos en el Dataset ya que los vuelos no llegan a producirse. Se podría poner, a nivel conceptual arrdelay= ∞ , pero nos generaría outliers, así que tampoco contamos contamos con ello
- 'Month', 'DayofMonth', 'DayOfWeek. Son el Mes, día del mes o de la semana en la que se produce el vuelo. Como vimos, tampoco tiene ningún peso matemático con ArrDelay, así como el Origen o del Destino. No hay días del año o aeropuertos en que haya más probabilidad de tener un retraso. -También eliminamos 'ActualElapsedTime', 'CRSElapsedTime', 'AirTime' ya que tienen un 0.95 de correlación o más con Distance y aportan la misma información. ActualElapsed es el tiempo esperado total del vuelo(desembarco, salida, vuelo, más atterizaje), CRSElapsedTime es el mismo tiempo previsto, mientras que AirTime es el tiempo que el avión está en el aire y Distance la distancia recorrida en millas.

In [4]:
```
df[["ActualElapsedTime", 'CRSElapsedTime', 'AirTime' , "Distance" ]].corr()
```

Out[4]:

	ActualElapsedTime	CRSElapsedTime	AirTime	Distance
ActualElapsedTime	1.000000	0.971122	0.976660	0.952980
CRSElapsedTime	0.971122	1.000000	0.986086	0.981759
AirTime	0.976660	0.986086	1.000000	0.980294
Distance	0.952980	0.981759	0.980294	1.000000

In [5]:
```
df1= df.drop(["Unnamed: 0", "Year", "UniqueCarrier", 'FlightNum',"TailNum", "Month", "Dayo
              'DayOfWeek', 'Cancelled', 'CancellationCode', "Diverted", "Origin", "Dest","
df1.head(10)
```

Out[5]:

	DepTime	CRSDepTime	ArrTime	CRSArrTime	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	CarrierDelay	Wea
0	2003.0	1955	2211.0	2225	-14.0	8.0	810	4.0	8.0	NaN	
1	754.0	735	1002.0	1000	2.0	19.0	810	5.0	10.0	NaN	
2	628.0	620	804.0	750	14.0	8.0	515	3.0	17.0	NaN	
3	1829.0	1755	1959.0	1925	34.0	34.0	515	3.0	10.0	2.0	

	DepTime	CRSDepTime	ArrTime	CRSArrTime	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	CarrierDelay	Wea
4	1940.0	1915	2121.0	2110	11.0	25.0	688	4.0	10.0	NaN	
5	1937.0	1830	2037.0	1940	57.0	67.0	1591	3.0	7.0	10.0	
6	706.0	700	916.0	915	1.0	6.0	828	5.0	19.0	NaN	
7	1644.0	1510	1845.0	1725	80.0	94.0	828	6.0	8.0	8.0	
8	1029.0	1020	1021.0	1010	11.0	9.0	162	6.0	9.0	NaN	
9	1452.0	1425	1640.0	1625	15.0	27.0	1489	7.0	8.0	3.0	

Como vamos a empezar a transformar variables, vamos a hacer previamente un muestreo.

In [6]:
```
df2= df1.sample (390000,random_state=55)
```

Cómo vimos en los ejercicios 9.1 y 9.2, los valores NaN de Arrdelay, así como de otras variables coincidían con aquellos que el vuelo había desviado o cancelado, valores imposibles de deducir (los NaN) por el mismo concepto de cancelación o desvió. por lo que eliminamos los valores NaN de ArrDelay

In [7]:
```
df3=df2.dropna( subset=["ArrDelay"]).reset_index(drop=True)
df3.isna().sum()
```

Out[7]:
```
DepTime                0
CRSDepTime             0
ArrTime                0
CRSArrTime             0
ArrDelay               0
DepDelay               0
Distance               0
TaxiIn                 0
TaxiOut                0
CarrierDelay      137592
WeatherDelay      137592
NASDelay          137592
SecurityDelay     137592
LateAircraftDelay 137592
dtype: int64
```

In [8]:
```
df3.shape
```

Out[8]:
```
(388357, 14)
```

0.1. **Transformación de variables horarias**

En esta parte vamos a convertir las variables DepTime, CRSDepTime, ArrTime, CRSArrTime en la función cíclica.

Estas cuatro variables vienen en formato horario hh:mm. Lo que haremos será contar todos los minutos transcurridos durante

el día, siendo 0 minutos a las 00:00 y 1440 los minutos transcurridos durante el día a las 23:59.

Más adelante, en el apartado 0.4, transformaremos estas variables en cíclicas.

In [9]:
```
# Primero de todo convierto las variables horarias en formato hora y para eso tienen que l
#izquierda con ceros
# Primero tengo que convertir en entero las variables DepTime y ArrTime en enteros para ev
```

```
df3['DepTime'] = df3['DepTime'].astype(int)
```

In [10]:
```
df3['ArrTime'] = df3['ArrTime'].astype(int)
```

In [11]:
```
#relleno por la izquierda con ceros
df3['DepTime'] = df3['DepTime'].astype(str).str.zfill(4)
df3['CRSDepTime'] = df3['CRSDepTime'].astype(str).str.zfill(4)
df3['ArrTime'] = df3['ArrTime'].astype(str).str .zfill(4)
df3['CRSArrTime'] = df3['CRSArrTime'].astype(str).str.zfill(4)
df3.head()
```

Out[11]:

	DepTime	CRSDepTime	ArrTime	CRSArrTime	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	CarrierDelay	Wea
0	0918	0905	1044	1035	9.0	13.0	414	3.0	12.0	NaN	
1	1421	1400	1500	1450	10.0	21.0	192	1.0	7.0	NaN	
2	2250	2153	0013	2330	43.0	57.0	432	5.0	17.0	0.0	
3	0049	0040	0521	0530	-9.0	9.0	1235	4.0	17.0	NaN	
4	1854	1845	2017	2010	7.0	9.0	488	4.0	11.0	NaN	

In [12]:
```
# las convierto en formato horario( Nota: en un principio lo pasé a formato horario por si
# al final opté por otro tipo de conversión)
df3['DepTime'] = df3['DepTime'].astype(str).str[:2]  + ':' + df3['DepTime'].astype(str).st
df3['CRSDepTime'] = df3['CRSDepTime'].astype(str).str[:2] + ':' + df3['CRSDepTime'].astype
df3['ArrTime'] = df3['ArrTime'].astype(str).str[:2] + ':' + df3['ArrTime'].astype(str).str
df3['CRSArrTime'] = df3['CRSArrTime'].astype(str).str[:2] + ':' + df3['CRSArrTime'].astype


df3
```

Out[12]:

	DepTime	CRSDepTime	ArrTime	CRSArrTime	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	CarrierDelay
0	09:18:00	09:05:00	10:44:00	10:35:00	9.0	13.0	414	3.0	12.0	NaN
1	14:21:00	14:00:00	15:00:00	14:50:00	10.0	21.0	192	1.0	7.0	NaN
2	22:50:00	21:53:00	00:13:00	23:30:00	43.0	57.0	432	5.0	17.0	0.0
3	00:49:00	00:40:00	05:21:00	05:30:00	-9.0	9.0	1235	4.0	17.0	NaN
4	18:54:00	18:45:00	20:17:00	20:10:00	7.0	9.0	488	4.0	11.0	NaN
...	...	...	...	...	...	...	...	...	...	...
388352	08:59:00	08:45:00	11:01:00	10:20:00	41.0	14.0	1333	5.0	15.0	14.0
388353	14:59:00	14:50:00	16:52:00	16:50:00	2.0	9.0	594	2.0	14.0	NaN
388354	21:32:00	20:30:00	22:43:00	21:30:00	73.0	62.0	255	5.0	14.0	11.0
388355	19:42:00	17:50:00	21:18:00	19:35:00	103.0	112.0	583	3.0	14.0	83.0
388356	08:15:00	07:45:00	09:32:00	09:05:00	27.0	30.0	427	5.0	13.0	0.0

388357 rows × 14 columns

In [13]:
```
# creamos la función minutos, que divide la hora hh:mm con un Split, en una lista ("hh","m
# para luego pasarlos a minutos, y con la reconverión ya comentada aplica la función minut
```

```
def minutos(x):
    x=x.split( sep=":")
    seg= 60*(int(x[0]))+(int(x[1]))

    return seg



dfhoras= df3[["DepTime", "CRSDepTime", "ArrTime", "CRSArrTime"]]
```

In [14]:
```
dfhoras_DT= dfhoras["DepTime"].apply(minutos)
dfhoras_CRSD=dfhoras["CRSDepTime"].apply(minutos)
dfhoras_AT=dfhoras["ArrTime"].apply(minutos)
dfhoras_CRSA=dfhoras["CRSArrTime"].apply(minutos)
```

In [15]:
```
df4= df3.drop([ 'DepTime',
       'CRSDepTime', 'ArrTime', 'CRSArrTime'], axis=1)
```

In [16]:
```
# ahora añadimos las cuatro columnas nuevas

df5= pd.concat([df4, dfhoras_DT,dfhoras_CRSD , dfhoras_AT, dfhoras_CRSA], axis=1)
df5.columns
```

Out[16]:
```
Index(['ArrDelay', 'DepDelay', 'Distance', 'TaxiIn', 'TaxiOut', 'CarrierDelay',
       'WeatherDelay', 'NASDelay', 'SecurityDelay', 'LateAircraftDelay',
       'DepTime', 'CRSDepTime', 'ArrTime', 'CRSArrTime'],
      dtype='object')
```

In [17]:
```
df5[["DepTime", "CRSDepTime", "ArrTime", "CRSArrTime"]].describe()# miramos como quedan pa
# los máximos y mínimos
```

Out[17]:

	DepTime	CRSDepTime	ArrTime	CRSArrTime
count	388357.000000	388357.00000	388357.000000	388357.000000
mean	922.620877	891.35868	978.678528	992.171942
std	269.858220	255.01740	327.865454	278.288941
min	1.000000	0.00000	1.000000	0.000000
25%	723.000000	695.00000	797.000000	805.000000
50%	944.000000	910.00000	1035.000000	1025.000000
75%	1140.000000	1095.00000	1230.000000	1213.000000
max	1440.000000	1439.00000	1440.000000	1439.000000

In [18]:
```
df5.isna().sum()
```

Out[18]:
```
ArrDelay                0
DepDelay                0
Distance                0
TaxiIn                  0
TaxiOut                 0
CarrierDelay       137592
WeatherDelay       137592
NASDelay           137592
SecurityDelay      137592
```

```
LateAircraftDelay    137592
DepTime                   0
CRSDepTime                0
ArrTime                   0
CRSArrTime                0
dtype: int64
```

0.2 **Valores NaN**

En esta sección vamos a completar las variables del motivo del retraso que tiene varios NaN. Como ya pudimos observar en el ejercicio 9.2, los vuelos con retrasos de menos de 14 minutos, tienen valores Nan

Carrier Delay es el retraso de la compañía

WeatherDelay es el retraso por las condiciones climatológicas SecurityDelay es el retraso por cuestiones de seguridad

LateAircraftDelay es el retraso de la misma aeronave.

Nas delay son los retraso causado por el Sistema Nacional del Espacio Aéreo (NAS)

por lo que vamos a asigarn 0 a los valores Nan

In [19]:
```
df_delay= df5[['ArrDelay','DepDelay','CarrierDelay', 'WeatherDelay', 'NASDelay', 'Security
df5_0= df5.drop(  ["CarrierDelay", 'WeatherDelay', 'NASDelay', 'SecurityDelay','LateAircra
delay_not_NAN= df_delay[['CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay','Late
```

In [20]:
```
df5= pd.concat([df5_0, delay_not_NAN], axis =1)
df5.head(10)
```

Out[20]:

	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	DepTime	CRSDepTime	ArrTime	CRSArrTime	CarrierDelay	Wea
0	9.0	13.0	414	3.0	12.0	558	545	644	635	0.0	
1	10.0	21.0	192	1.0	7.0	861	840	900	890	0.0	
2	43.0	57.0	432	5.0	17.0	1370	1313	13	1410	0.0	
3	-9.0	9.0	1235	4.0	17.0	49	40	321	330	0.0	
4	7.0	9.0	488	4.0	11.0	1134	1125	1217	1210	0.0	
5	134.0	105.0	316	15.0	39.0	1150	1045	1219	1085	0.0	
6	-15.0	8.0	1235	5.0	16.0	768	760	997	1012	0.0	
7	32.0	40.0	304	5.0	13.0	440	400	502	470	32.0	
8	24.0	17.0	1589	5.0	29.0	527	510	654	630	17.0	
9	49.0	36.0	787	7.0	16.0	1226	1190	1369	1320	6.0	

In [21]:
```
df5.isna().sum()
```

Out[21]:
```
ArrDelay            0
DepDelay            0
Distance            0
TaxiIn              0
TaxiOut             0
DepTime             0
CRSDepTime          0
ArrTime             0
```

```
CRSArrTime          0
CarrierDelay        0
WeatherDelay        0
NASDelay            0
SecurityDelay       0
LateAircraftDelay   0
dtype: int64
```

0.3 **dos nuevas variables.**

Se puede ver a simple vista que ArrDelay y DepDelay se obtienen de la resta entre (DepTime-CRSDepTime)y (ArrTime-CRSArrTime), así que calculamos dos nuevas variables, para reducir la dimensionalidad.

Primero miramos el valor más bajo de ArrDelay, para poder hacer los cálculo correctamente

In [22]:
```
minimo=df[df["ArrDelay"]<0].sort_values("ArrDelay")# miramos los valores más bajos de Arr
zmin=minimo["ArrDelay"].min()
zmin
```

Out[22]:
```
-109.0
```

In [23]:
```
def rest(z):
    x=z[0]
    y=z[1]
    if (x < y) & ((x-y)<zmin):
        t= (1440+x)-y
        return t
    else:
        t= x-y
        return t

x11= df5[["DepTime","CRSDepTime" ]].apply(rest, axis=1)
x10= df5[["ArrTime","CRSArrTime" ]].apply(rest,axis=1)

x10=x10.rename("X10")
x11=x11.rename("X11")
x10.describe()
```

Out[23]:
```
count    388357.000000
mean         42.414732
std          60.245467
min         -96.000000
25%           9.000000
50%          24.000000
75%          56.000000
max        1439.000000
Name: X10, dtype: float64
```

In [24]:
```
x11.describe()
```

Out[24]:
```
count    388357.000000
mean         42.990375
std          52.804525
min         -82.000000
25%          12.000000
50%          24.000000
75%          53.000000
max        1280.000000
Name: X11, dtype: float64
```

Sean **x10** y **x11** definidas por

x10= df6["ArrTime"]-df6["CRSArrTime"]

x11= df6["DepTime"]-df6["CRSDepTime"] y tienen una relación lineal con ArrDelay y DepDeplay respectivamente

In [25]:
```
x1= pd.concat([x10,x11,df5["ArrDelay"],df5["DepDelay"] ],axis=1)
x1.corr()
```

Out[25]:

	X10	X11	ArrDelay	DepDelay
X10	1.000000	0.885914	0.925995	0.884185
X11	0.885914	1.000000	0.951128	0.998096
ArrDelay	0.925995	0.951128	1.000000	0.952835
DepDelay	0.884185	0.998096	0.952835	1.000000

In [26]:
```
# remodelamos el data set con las dos nuevas variables
df5b=df5.drop(['DepTime', 'CRSDepTime', 'ArrTime', 'CRSArrTime'], axis=1)
df6=pd.concat([df5b, x10,x11], axis =1)
df6.head(10)
```

Out[26]:

	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	CarrierDelay	WeatherDelay	NASDelay	SecurityDelay	LateAircraf
0	9.0	13.0	414	3.0	12.0	0.0	0.0	0.0	0.0	
1	10.0	21.0	192	1.0	7.0	0.0	0.0	0.0	0.0	
2	43.0	57.0	432	5.0	17.0	0.0	43.0	0.0	0.0	
3	-9.0	9.0	1235	4.0	17.0	0.0	0.0	0.0	0.0	
4	7.0	9.0	488	4.0	11.0	0.0	0.0	0.0	0.0	
5	134.0	105.0	316	15.0	39.0	0.0	0.0	87.0	0.0	
6	-15.0	8.0	1235	5.0	16.0	0.0	0.0	0.0	0.0	
7	32.0	40.0	304	5.0	13.0	32.0	0.0	0.0	0.0	
8	24.0	17.0	1589	5.0	29.0	17.0	0.0	7.0	0.0	
9	49.0	36.0	787	7.0	16.0	6.0	0.0	13.0	0.0	

In [27]:
```
df6.shape
```

Out[27]:
```
(388357, 12)
```

En el siguiente paso vamos a analizar la multicolinealidad. Para ver si alguna variable tiene un multicolinealidad muy elevada VIFi = 1/ (1 -Ri^2) donde Ri es el coeficiente de determinación de la regresión lineal

In [28]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor

def vif(X):
    vifDF = pd.DataFrame()
    vifDF["variables"] = X.columns
    vifDF["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vifDF
Xvif=df6.drop(["ArrDelay"], axis=1)
round(vif(Xvif),2)
```

Out[28]:

	variables	VIF
0	DepDelay	472.98
1	Distance	2.64
2	TaxiIn	2.21
3	TaxiOut	3.10
4	CarrierDelay	10.92
5	WeatherDelay	2.97
6	NASDelay	5.71
7	SecurityDelay	1.02
8	LateAircraftDelay	11.72
9	X10	10.52
10	X11	448.95

Podemos observar en las variables independientes que DepDelay tiene un "factor de agrandamiento de la varianza "(VIF) muy elevado Así que procedemos a eliminar DepDelay y ver como queda el VIF

In [29]:

```
Xvif2=df6.drop(["ArrDelay", "DepDelay"], axis=1)

round(vif(Xvif2),2)
```

Out[29]:

	variables	VIF
0	Distance	2.62
1	TaxiIn	2.21
2	TaxiOut	3.06
3	CarrierDelay	10.07
4	WeatherDelay	2.84
5	NASDelay	5.42
6	SecurityDelay	1.02
7	LateAircraftDelay	11.01
8	X10	10.37
9	X11	23.68

Podemos observar que el vif ha mejorado notablemente que pero deberíamos tener un Factor de agrandamiento de la varianza por debajo de 5, así que eliminamos la siguiente variable con más VIF que es X11

In [30]:

```
Xvif3=df6.drop(["ArrDelay", "DepDelay", "X11"], axis=1)

round(vif(Xvif3),2)
```

Out[30]:

	variables	VIF
0	Distance	2.19
1	TaxiIn	2.20

	variables	VIF
2	TaxiOut	2.61
3	CarrierDelay	3.71
4	WeatherDelay	1.59
5	NASDelay	3.01
6	SecurityDelay	1.01
7	LateAircraftDelay	3.99
8	X10	10.26

Estas podrían ser una buena selección de variables pero vamos a ver que nos dice el método Anova para selección de variables en clasificación

In [31]:
```
ypreseleccion= df6["ArrDelay"]
def categorizar ( x):
    if x<0 :
        t= 1
        return t
    else:
        t=0
        return t
Y0= ypreseleccion.apply( categorizar)
Y0.value_counts()
```

Out[31]:
```
0    352231
1     36126
Name: ArrDelay, dtype: int64
```

In [32]:
```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
Xpreseleccion= df6.drop(["ArrDelay"], axis=1)

for i in range(1,12): # vamos a mirar como f_selection va escogiendo las variables en func
    fs = SelectKBest(score_func=f_classif, k=i)
    XS=fs.fit(Xpreseleccion,Y0)


    filter=fs.get_support()
    variables=np.array(Xpreseleccion.columns)
    print(variables[filter])
    print(XS.scores_[filter])
    print ("\n")
```

```
['X10']
[24201.4517981]


['X10' 'X11']
[24201.4517981  15895.98672409]


['DepDelay' 'X10' 'X11']
[15802.90975804 24201.4517981  15895.98672409]


['DepDelay' 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  8464.97417996 24201.4517981  15895.98672409]
```

```
['DepDelay' 'Distance' 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  7577.92853205  8464.97417996 24201.4517981
 15895.98672409]


['DepDelay' 'Distance' 'TaxiOut' 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  7577.92853205  5503.79740071  8464.97417996
 24201.4517981  15895.98672409]


['DepDelay' 'Distance' 'TaxiOut' 'NASDelay' 'LateAircraftDelay' 'X10'
 'X11']
[15802.90975804  7577.92853205  5503.79740071  4840.90984533
  8464.97417996 24201.4517981  15895.98672409]


['DepDelay' 'Distance' 'TaxiOut' 'CarrierDelay' 'NASDelay'
 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  7577.92853205  5503.79740071  4750.48690446
  4840.90984533  8464.97417996 24201.4517981  15895.98672409]


['DepDelay' 'Distance' 'TaxiIn' 'TaxiOut' 'CarrierDelay' 'NASDelay'
 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  7577.92853205  1552.33280781  5503.79740071
  4750.48690446  4840.90984533  8464.97417996 24201.4517981
 15895.98672409]


['DepDelay' 'Distance' 'TaxiIn' 'TaxiOut' 'CarrierDelay' 'WeatherDelay'
 'NASDelay' 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  7577.92853205  1552.33280781  5503.79740071
  4750.48690446   760.91617888  4840.90984533  8464.97417996
 24201.4517981  15895.98672409]


['DepDelay' 'Distance' 'TaxiIn' 'TaxiOut' 'CarrierDelay' 'WeatherDelay'
 'NASDelay' 'SecurityDelay' 'LateAircraftDelay' 'X10' 'X11']
[15802.90975804  7577.92853205  1552.33280781  5503.79740071
  4750.48690446   760.91617888  4840.90984533    51.89537543
  8464.97417996 24201.4517981  15895.98672409]
```

Podemos ver que el método Anova puntúa para la clasificación, de mejor manera las variables DepDelay y X11, obviamente con más correlación lineal que el resto de variables, y para evitar la multicolinealidad y el ruido, evitaremos usarlas. Probamos a ver, si eliminando alguna variable con poco peso para el test de Anova, conseguimos reducir el VIF, tras varia

In [33]:
```
Xvif4=df6.drop(["ArrDelay", "DepDelay", "X11",  "CarrierDelay"], axis=1)

round(vif(Xvif4),2)
```

Out[33]:

	variables	VIF
0	Distance	2.19
1	TaxiIn	2.20
2	TaxiOut	2.60
3	WeatherDelay	1.17

	variables	VIF
4	NASDelay	1.77
5	SecurityDelay	1.00
6	LateAircraftDelay	1.92
7	X10	3.10

Podemos observar que elimiando Carrier Delay, que es la octava mejor variable, de once que hay, para el problema de clasificación, con el método de Anova, nos queda una VIF bueno para ajuste.

In [34]:
```
Y0= Y0.rename("ArrDelay")# renombramos Y0 como ArrDelay
scaler = StandardScaler()

df70= pd.DataFrame(scaler.fit_transform(Xvif4), columns=Xvif4.columns )# estandarizamos la

df7= pd.concat( [df70,Y0 ], axis=1)
df7.head(10)
```

Out[34]:

	Distance	TaxiIn	TaxiOut	WeatherDelay	NASDelay	SecurityDelay	LateAircraftDelay	X10	ArrDelay
0	-0.612104	-0.725974	-0.434897	-0.138081	-0.346468	-0.036093	-0.456058	-0.554644	0
1	-0.999535	-1.108255	-0.785606	-0.138081	-0.346468	-0.036093	-0.456058	-0.538045	0
2	-0.580690	-0.343693	-0.084187	2.358603	-0.346468	-0.036093	-0.456058	0.009715	0
3	0.820694	-0.534833	-0.084187	-0.138081	-0.346468	-0.036093	-0.456058	-0.853422	1
4	-0.482960	-0.534833	-0.505039	-0.138081	-0.346468	-0.036093	-0.456058	-0.587841	0
5	-0.783132	1.567714	1.458933	-0.138081	2.755455	-0.036093	0.860013	1.520204	0
6	0.820694	-0.343693	-0.154329	-0.138081	-0.346468	-0.036093	-0.456058	-0.953015	1
7	-0.804074	-0.343693	-0.364755	-0.138081	-0.346468	-0.036093	-0.456058	-0.172872	0
8	1.438490	-0.343693	0.757515	-0.138081	-0.096888	-0.036093	-0.456058	-0.305662	0
9	0.038851	0.038589	-0.154329	-0.138081	0.117038	-0.036093	0.383987	0.109307	0

Vamos a ver si la clase ArrDelay está desvalanceada

In [35]:
```
print(Y0.value_counts()[0]/(Y0.value_counts()[0]+Y0.value_counts()[1]))
print(Y0.value_counts()[1]/(Y0.value_counts()[0]+Y0.value_counts()[1]))
```

```
0.9069773430117134
0.09302265698828655
```

In [36]:
```
df7.describe().round(2)
```

Out[36]:

	Distance	TaxiIn	TaxiOut	WeatherDelay	NASDelay	SecurityDelay	LateAircraftDelay	X10	ArrD
count	388357.00	388357.00	388357.00	388357.00	388357.00	388357.00	388357.00	388357.00	3883
mean	-0.00	-0.00	-0.00	0.00	0.00	-0.00	-0.00	0.00	
std	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
min	-1.28	-1.30	-1.28	-0.14	-0.35	-0.04	-0.46	-2.30	
25%	-0.74	-0.53	-0.58	-0.14	-0.35	-0.04	-0.46	-0.55	

	Distance	TaxiIn	TaxiOut	WeatherDelay	NASDelay	SecurityDelay	LateAircraftDelay	X10	ArrD
50%	-0.28	-0.15	-0.29	-0.14	-0.35	-0.04	-0.46	-0.31	
75%	0.41	0.23	0.20	-0.14	-0.13	-0.04	0.05	0.23	
max	7.33	36.93	24.26	66.52	35.63	149.21	27.60	23.18	

Aunque la clase está desbalanceada, no es por debajo del 5%, así que no vamos a manipular el desbalanceo. Por último hacemos la separación en Train y Test

In [37]:
```
X_= df7.drop(["ArrDelay"], axis=1)

y_= df7["ArrDelay"]

Xtrain, Xtest, ytrain, ytest = train_test_split( X_, y_, test_size=0.30, random_state=42,
```

- **Ejercicio 1**.

Crea al menos tres modelos de clasificación, Haremos una regresión logística, y dos predicciones por Random Forest y red neuronal

In [38]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
log= LogisticRegression()
rft= RandomForestClassifier()
mlp= MLPClassifier()
```

Empezamos por la regresión logística

In [39]:
```
log.fit(Xtrain,ytrain)
fxlog=log.predict(Xtest)
```

In [40]:
```
log.score(Xtest, ytest)# devuelve Accuracy
```

Out[40]:
```
0.9602087410306588
```

Seguimos por el Bosque

In [41]:
```
rft.fit(Xtrain, ytrain)
fxrft= rft.predict(Xtest)
```

In [42]:
```
rft.score(Xtest, ytest)
```

Out[42]:
```
0.9999914168984104
```

Acabamos con el la red neuronal

In [43]:
```
mlp.fit(Xtrain, ytrain)
fxmlp= mlp.predict(Xtest)
```

In [44]:
```
mlp.score(Xtest, ytest)
```

Out[44]:
```
0.9999914168984104
```

Podemos observar que tanto la Red Neuronal como el Bosque aleatorio tienen mejor precisión (accuracy) que la regresión logística

Vamos a intentar analizar mejor el error con otros estimadores

1. Cálculo de métricas.

calcula la accuracy, la matriz de confianza , y otras métricas más avanzadas

In [45]:
```
from sklearn.metrics import accuracy_score
```

Calculamos la accuracy para los tres modelos de clasificación

In [46]:
```
print ( " precisión para la regresión logísticas es : ", accuracy_score( ytest, fxlog))
print ( " precisión para el Bosque aleatorio es : ", accuracy_score( ytest, fxrft))
print ( " precisión para la red neuronal es : ", accuracy_score( ytest, fxmlp))
```

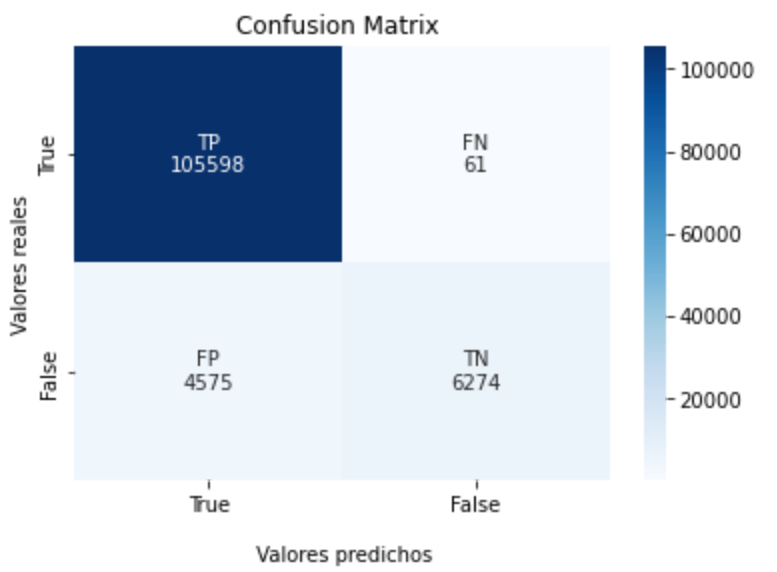
```
precisión para la regresión logísticas es :  0.9602087410306588
precisión para el Bosque aleatorio es :  0.9999914168984104
precisión para la red neuronal es :  0.9999914168984104
```

Que son los resultados obtenidos previamente con score(x,y)

In [47]:
```
from sklearn.metrics import confusion_matrix
```

-Empezamos por la regresión logística

In [48]:
```
cmlog= confusion_matrix(ytest, fxlog)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmlog.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmlog, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



Observamos que detecta mucho mejor los vuelos retrasados(clase mayoritaria), que los que llegan puntuales, clase minoritaria.

Calculamos la sensibilidad y la especificidad.

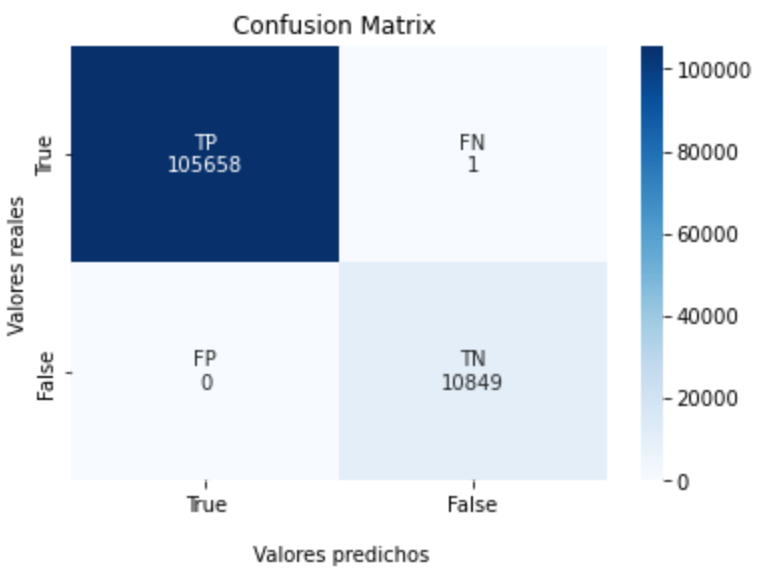
In [49]:
```
print(" sensibilidad =  ", cmlog[0][0]/(cmlog[0][0]+cmlog[0][1]))
print(" especificidad =  ", cmlog[1][1]/(cmlog[1][1]+cmlog[1][0]))
```

```
 sensibilidad =   0.9994226710455333
 especificidad =   0.5783021476633791
```

Podemos ver que a la regresión logística le cuesta acertar los verdaderos negativos, la clase minoritaria, no llega ni a acertar el 60% de los vuelos que llegarán puntuales. Si para la compañía, es más importante conocer los vuelos que van a llegar tarde, para preveer los costes generados por los retrasos, es una buena clasificación.

- Bosque Aleatorio

In [50]:
```
cmrft= confusion_matrix(ytest, fxrft)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmrft.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmrft, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');



ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```

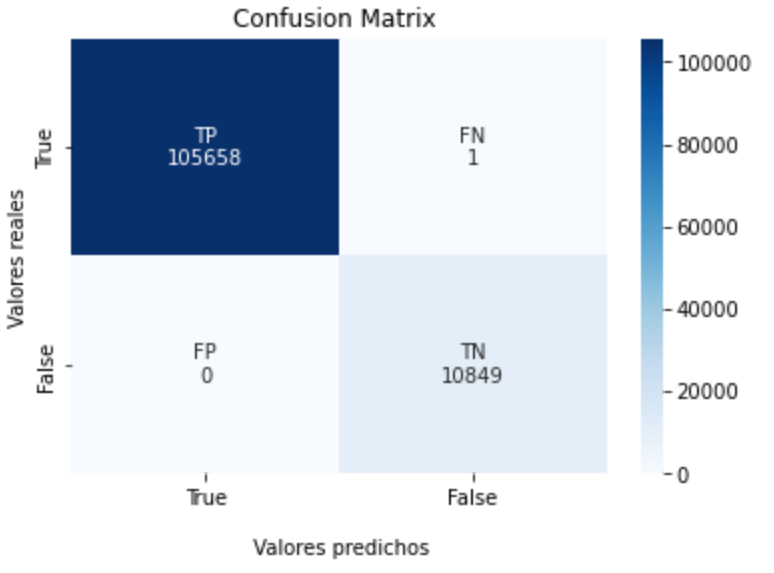


Observamos que el índice de acierto es casi perfecto, con sensibilidad y especificad igual a uno(aprox)

- Red Neuronal

In [51]:

```
cmmlp= confusion_matrix(ytest, fxmlp)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmmlp.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmmlp, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');



ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



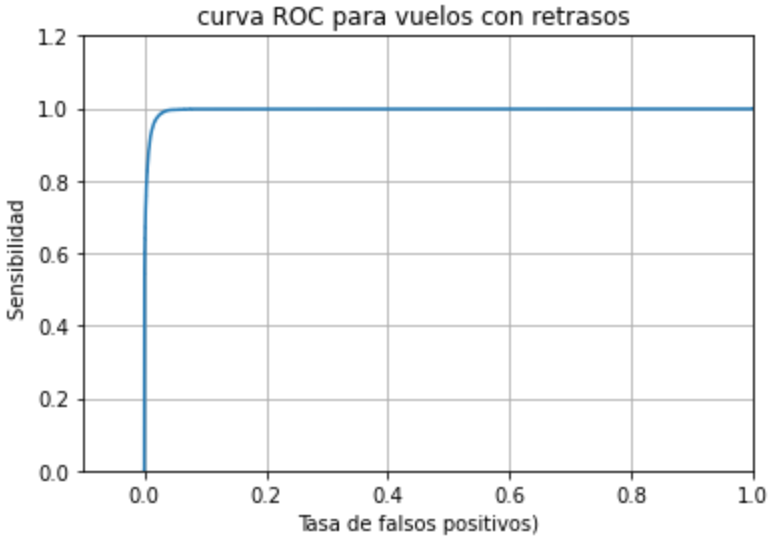
Cómo en el bosque aleatorio la sensibilidad casi igual a 1 y la especificidad igual a 1

Curva Roc y Area bajo al curva

Vamos a comrpobar la precisión de la curva roc y el área bajo la curva

- Regresión logística

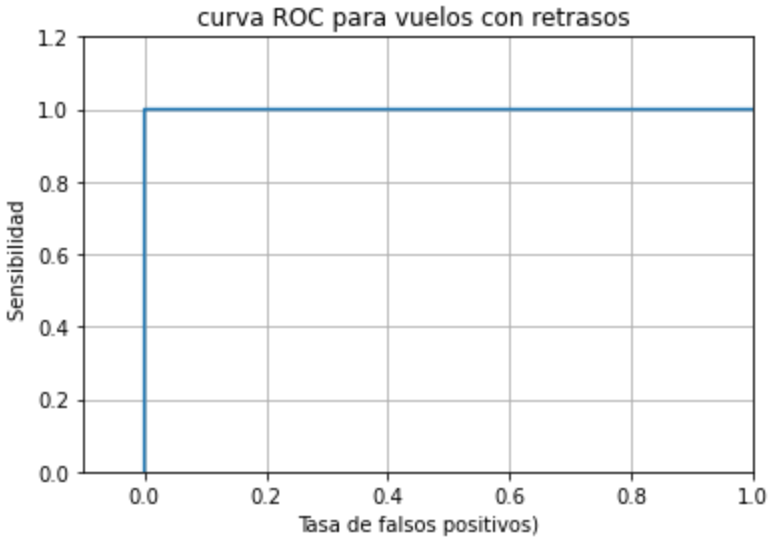
In [56]:
```
probalog= log.predict_proba(Xtest)
from sklearn import metrics
FP, TP, thresholds = metrics.roc_curve(ytest, probalog[::,1])

plt.plot(FP, TP)
plt.xlim([-0.1, 1.0])
plt.ylim([0.0, 1.2])

plt.title('curva ROC para vuelos con retrasos ')
plt.xlabel('Tasa de falsos positivos)')
plt.ylabel('Sensibilidad ')
plt.grid(True)
```



In [53]:
```
# calculamos el área bajo la curva
area= metrics.roc_auc_score(ytest,probalog[:,1])
print ("el área bajo la curva es igual a : " , area)
```

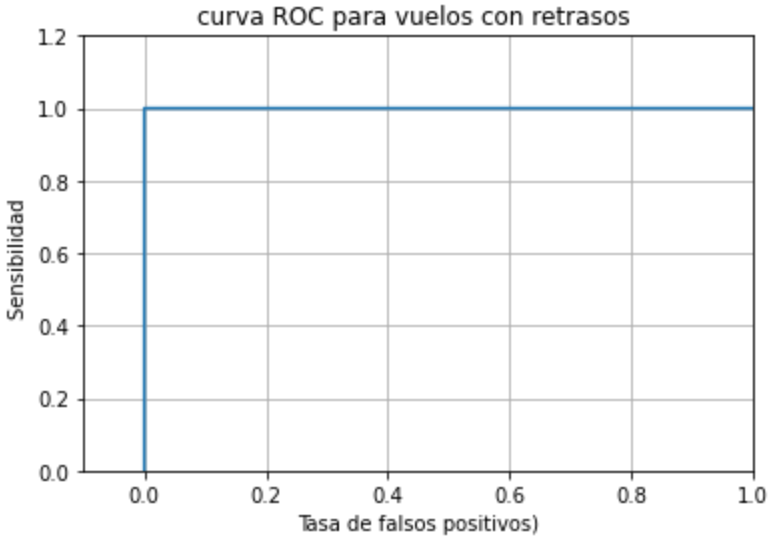
```
el área bajo la curva es igual a :  0.9952594747312626
```

In [54]:
```
probarft= rft.predict_proba(Xtest)
from sklearn import metrics
FP, TP, thresholds = metrics.roc_curve(ytest, probarft[:,1])

plt.plot(FP, TP)
plt.xlim([-0.1, 1.0])
plt.ylim([0.0, 1.2])

plt.title('curva ROC para vuelos con retrasos ')
plt.xlabel('Tasa de falsos positivos)')
plt.ylabel('Sensibilidad ')
plt.grid(True)
```



In [55]:

```
probamlp= mlp.predict_proba(Xtest)
from sklearn import metrics
FP, TP, thresholds = metrics.roc_curve(ytest, probamlp[:,1])

plt.plot(FP, TP)
plt.xlim([-0.1, 1.0])
plt.ylim([0.0, 1.2])

plt.title('curva ROC para vuelos con retrasos ')
plt.xlabel('Tasa de falsos positivos)')
plt.ylabel('Sensibilidad ')
plt.grid(True)
```



En los casos de del Bosque Aleatorio y de la Red Neuronal, el Área bajo la curva es 1 ya que los falsos Positivos son cero

- Ejercicio 3.

Entrena los modelos usando distintos parámetros.

- **Regresión logística.**

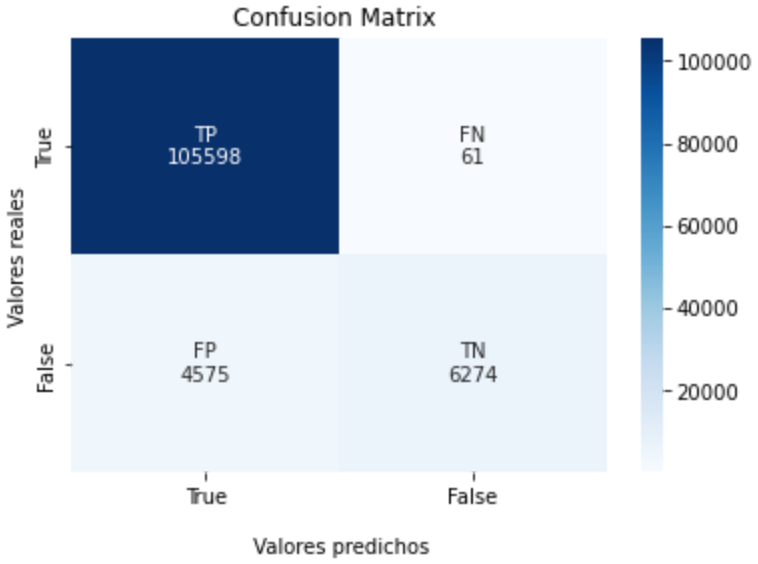
En este caso voy a intentar mejorar el rendimiento, usando el cálculo completo de la matriz Hessiana mediante el método de Newton, ya que los datos no están normalizados, y por bibliografía encontrada, parece que el método de del cálculo del gradiente funciona de manera más optima.

Pondría una penalización L1, ya que sabemos que no hay multicolinealidad, pero no es admisible con Newton.

In [58]:
```
log1= LogisticRegression(solver= "newton-cg")
```

In [59]:
```
log1.fit(Xtrain,ytrain)
fxlog1=log1.predict(Xtest)
```

In [60]:
```
# miramos la matriz de confusión
cmlog1= confusion_matrix(ytest, fxlog1)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmlog1.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmlog1, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



In [64]:
```
print(" sensibilidad =  ", cmlog1[0][0]/(cmlog1[0][0]+cmlog1[0][1]))
print(" especificidad =  ", cmlog1[1][1]/(cmlog1[1][1]+cmlog1[1][0]))
```

```
 sensibilidad =   0.9994226710455333
 especificidad =   0.5783021476633791
```

In [76]:
```
print(Xtrain.columns)
print (log1.coef_)
```

```
Index(['Distance', 'TaxiIn', 'TaxiOut', 'WeatherDelay', 'NASDelay',
       'SecurityDelay', 'LateAircraftDelay', 'X10'],
```

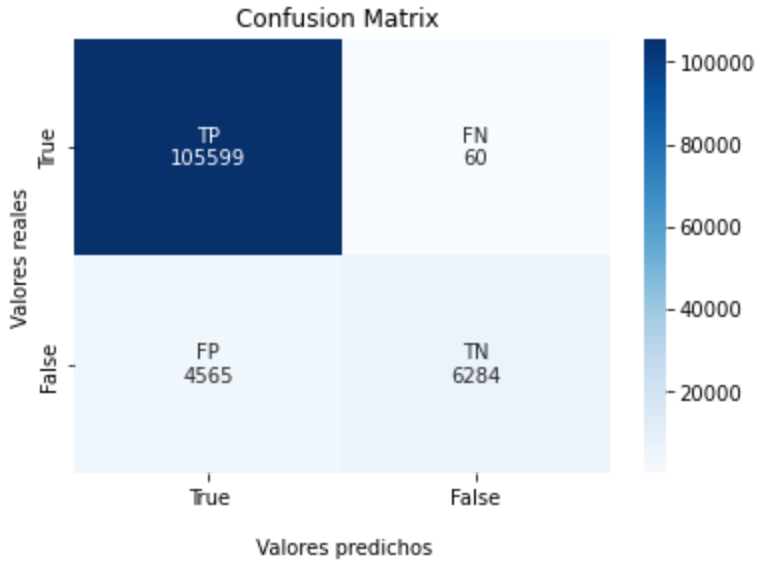
```
      dtype='object')
[[ 0.28278018 -0.11213857 -0.40381137 -2.26513331 -7.44574972 -0.48298071
  -7.60047795 -9.55181298]]
```

Sacamos los mismo resultado. Así que vamos a intentar mejorar el resultado cambiando a una regularización L1,

In [96]:
```
log2= LogisticRegression( solver = "saga",penalty="elasticnet", l1_ratio =0.1, random_stat
```

In [97]:
```
log2.fit(Xtrain,ytrain)
fxlog2=log2.predict(Xtest)
```

In [123…
```
# miramos la matriz de confusión
cmlog2= confusion_matrix(ytest, fxlog2)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmlog2.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmlog2, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



In [99]:
```
print(" sensibilidad =  ", cmlog2[0][0]/(cmlog2[0][0]+cmlog2[0][1]))
print(" especificidad =  ", cmlog2[1][1]/(cmlog2[1][1]+cmlog2[1][0]))
```

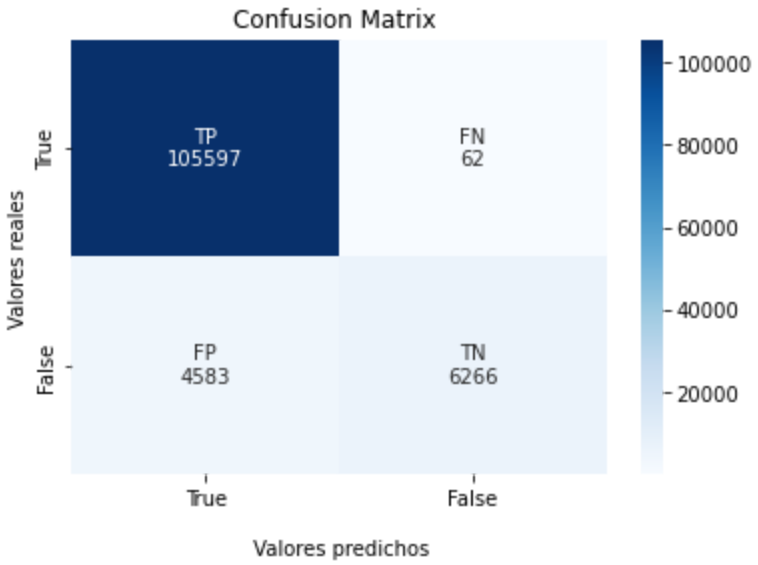
```
 sensibilidad =   0.9994321354546228
 especificidad =   0.5792238916029128
```

Para intentar otra solución, vamos a intentar penalizar el modelo, aumentando el sobreajuste de los datos

In [162…
```
log4= LogisticRegression(  C= 100000)
```

In [164…
```
log4.fit(Xtrain,ytrain)
fxlog4=log4.predict(Xtest)
```

In [165…
```
# miramos la matriz de confusión
cmlog4= confusion_matrix(ytest, fxlog4)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmlog4.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmlog4, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



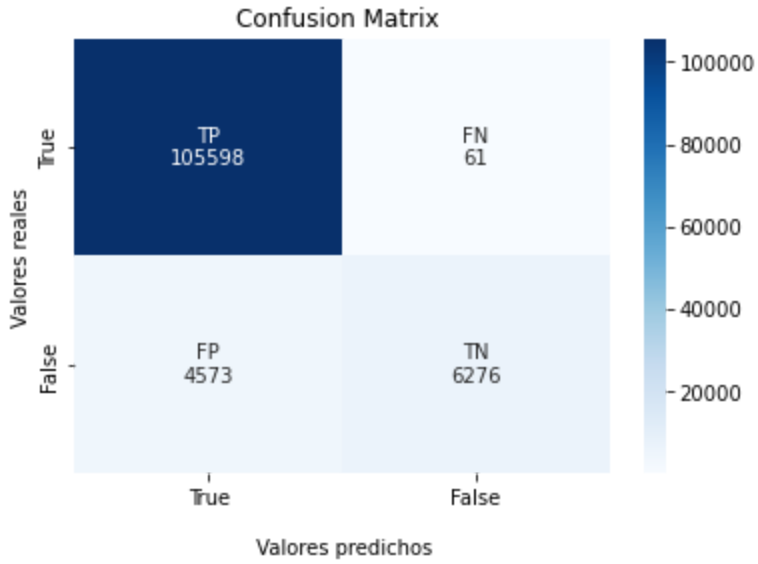
Podemos ver que tampoco hemos mejorado mucho la precisión de los falsos positivos incluso con sobreajuste

In [121…
```
log5= LogisticRegression(solver= "newton-cg", multi_class= "multinomial")
```

In [122…
```
log5.fit(Xtrain,ytrain)
fxlog5=log5.predict(Xtest)
```

In [155…
```
# miramos la matriz de confusión
cmlog5= confusion_matrix(ytest, fxlog5)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
```

```
                  cmlog5.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmlog5, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



Sospecho que una regularización del tipo L1 es más importante que el método de solución(Newton, Saga,etc...), penalizando la variables poco importantes con L1

In [168…

```
log6= LogisticRegression( solver = "saga",penalty="l1", random_state=21, max_iter=3000, C=
```

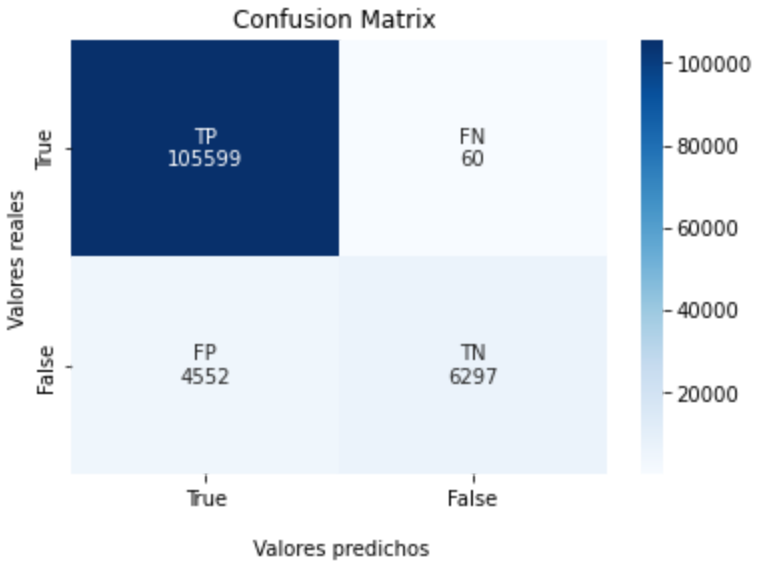
In [169…

```
log6.fit(Xtrain,ytrain)
fxlog6=log6.predict(Xtest)
```

In [170…

```
# miramos la matriz de confusión
cmlog6= confusion_matrix(ytest, fxlog6)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmlog6.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmlog6, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');
```

```
ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



In [171…
```
print(" sensibilidad =  ", cmlog6[0][0]/(cmlog6[0][0]+cmlog6[0][1]))
print(" especificidad =  ", cmlog6[1][1]/(cmlog6[1][1]+cmlog6[1][0]))
```

```
sensibilidad =   0.9994321354546228
especificidad =   0.5804221587243064
```

Se mejora ligeramente la especificidad, pero nada destacable.

- Bosque Aleatorio

Al igual que en la red neuronal, el bosque aleatorio funciona muy bien. Mi idea principal, sería modificar un parámetro para cada uno, ya que mejorar la sensibilidad y la especificidad, hasta encontrar el modelo que detecte el único falso negativo(contemos que ambos casos tiene sólo un falso negativo), lo veo innecesario. Al final, tal cómo he dejado el Data Set, con que el modelo detecte si X10 es mayor o menor que cero, es suficiente para que funcione y clasificamente correctamene. Pero bien podría ser que el modelo funcionosae bien para esos datos de test, así que vamos a adelanter el ejercicio 4 y vamos a hacer una validación cruzada.

- Ejercicio 4.

Comprobamos la validación cruzada en el train. con cuatro métricas

In [183…
```
from sklearn.model_selection import cross_validate
```

In [176…
```
metric = ["accuracy", "precision", "recall", "roc_auc"] # las 4 métricas
```

In [177…
```
CVlog = cross_validate( log , Xtrain, ytrain, cv=5, scoring=metric, return_train_score=Tru
```

In [178…
```
sorted(CVlog.keys())
```

Out[178…
```
['fit_time',
```

```
'score_time',
'test_accuracy',
'test_precision',
'test_recall',
'test_roc_auc',
'train_accuracy',
'train_precision',
'train_recall',
'train_roc_auc']
```

In [181…
```
test = ["test_accuracy","test_precision","test_recall" , "test_roc_auc"]
for k in test:
    print ( k , ":", CVlog[k])
```

```
test_accuracy : [0.96119183 0.96222181 0.95379805 0.9612654  0.96336147]
test_precision : [0.99066978 0.99245159 0.99187935 0.99035904 0.99291921]
test_recall : [0.58813056 0.59821958 0.50731804 0.58920095 0.61028684]
test_roc_auc : [0.9944198  0.9939508  0.99498612 0.9940373  0.99419331]
```

Observamos unos valores coherentes con lo predicho anteriormente, y ninguna parte del train falla

- Bosque Aleatorio

In [182…
```
CVrft = cross_validate( rft , Xtrain, ytrain, cv=5, scoring=metric, return_train_score=Tru
```

In [184…
```
for k in test:
    print ( k , ":", CVrft[k])
```

```
test_accuracy : [1. 1. 1. 1. 1.]
test_precision : [1. 1. 1. 1. 1.]
test_recall : [1. 1. 1. 1. 1.]
test_roc_auc : [1. 1. 1. 1. 1.]
```

Podemos observar lo esperado, así como en la **red neuronal** que viene tras esto.

In [185…
```
CVmlp = cross_validate( mlp , Xtrain, ytrain, cv=5, scoring=metric, return_train_score=Tru
```

In [186…
```
for k in test:
    print ( k , ":", CVmlp[k])
```

```
test_accuracy : [0.99998161 1.         0.99998161 0.99998161 1.        ]
test_precision : [0.99980222 1.         0.99980225 0.99980225 1.        ]
test_recall : [1. 1. 1. 1. 1.]
test_roc_auc : [0.9999973  1.         0.99998453 1.         1.        ]
```

Seguimos con el ejercicio 3. Haremos un par de probaturas en la red neuronal y el bosque Aleatorio. Vamos a intentar empeorar el rendimiento de ambos, ya que mejorarlo epodría llevar mucho trabajo.

En el caso del Bosque aleatorio optaremos por menos estimadores, el método de convergencia por entropía y límite de profundidad. En el caso de la Red Neuronal, usaremos relu para activación, y la convergencia lbfgs, apta para datasets pequeños.

In [188…
```
rft2= RandomForestClassifier(n_estimators= 40, criterion="entropy", max_depth= 3)
mlp2= MLPClassifier(hidden_layer_sizes= (50), activation="relu", solver="lbfgs")
```

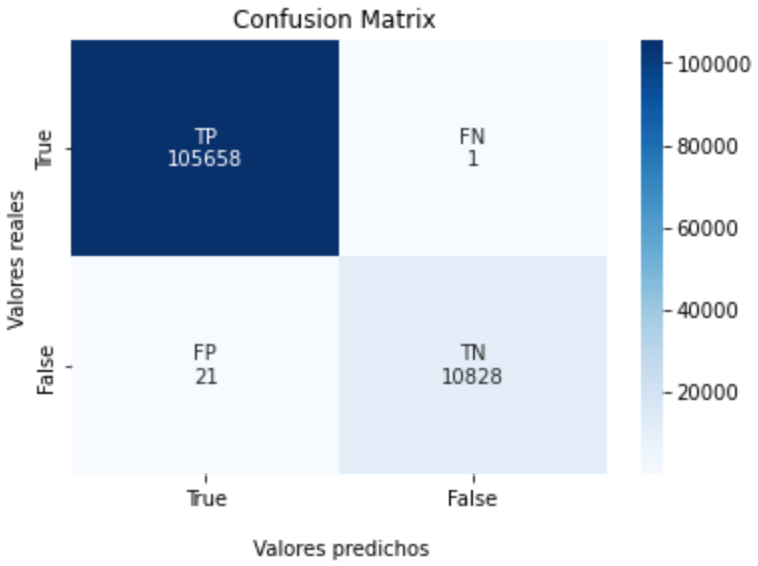
In [189…
```
rft2.fit(Xtrain, ytrain)
fxrft2= rft2.predict(Xtest)
```

In [190…

```
mlp2.fit(Xtrain, ytrain)
fxmlp2= mlp2.predict(Xtest)
```

In [191…

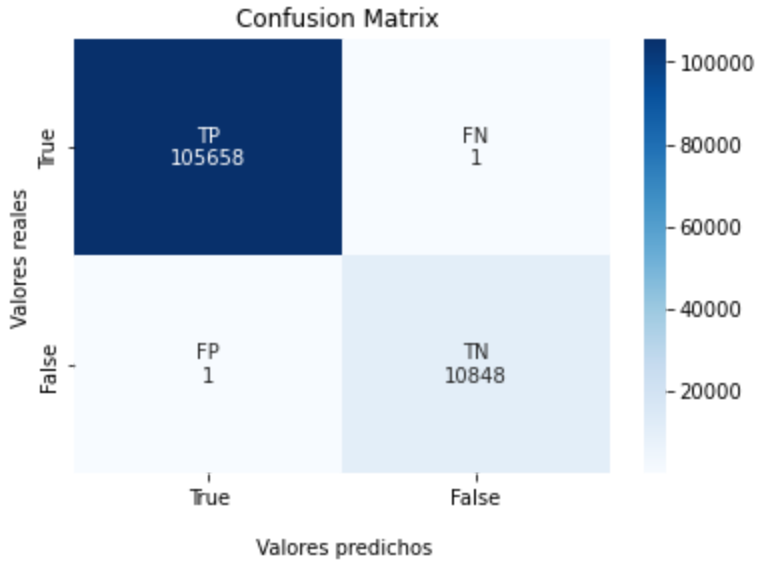
```
cmrft2= confusion_matrix(ytest, fxrft2)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmrft2.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmrft2, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



In [192…

```
cmmlp2= confusion_matrix(ytest, fxmlp2)

group_names = ["TP","FN","FP","TN"]
group_counts = ["{0:0.0f}".format(value) for value in
                cmmlp2.flatten()]
labels = [f"{v1}\n{v2}" for v1, v2 in
          zip(group_names,group_counts)]
labels = np.asarray(labels).reshape(2,2)
ax = sns.heatmap(cmmlp2, annot=labels, fmt="" ,cmap='Blues')

ax.set_title('Confusion Matrix ');
ax.set_xlabel('\n Valores predichos')
ax.set_ylabel('Valores reales');




ax.xaxis.set_ticklabels(['True','False'])
```

```
ax.yaxis.set_ticklabels(['True','False'])

plt.show()
```



In [193…
```
Xtest.shape
```

Out[193…
```
(116508, 8)
```

Conclusiones.

Primero: El dataset, una vez transformado, y observado que ArrDelay es igua a la resta de ArrTime y CRSArrtime, X10, el problema de clasificación se reduce a clasificar en función de signo de X10.

Segundo: La Red Neuronal y el Bosque aleatorio han clasificado todas bien, menos un falso negativo. Incluso empeorando ambos modelos a través de los parámetros, el fallo ha estado en 22 contra 116508.

Tercero. La regresión logística no ha clasificado tan bien un problema de clasificación tan simple al clasificar muchos falsos positivos. Incluso discriminando variables poco importantes para la clasificación, usando la regularización L1, ha seguido clasificando muchos falsos positivos. Seguramente un data Set más limpio de variables habría obtenido mejores resultados.

In [ ]:

In [ ]: