ASTEROIDES CON PELIGRO POTENCIAL PARA LA TIERRA

Recientemente, la NASA llevó a cabo el primer intento para desviar un asteriode que pudiera suponer un peligro de colisión con la Tierra (https://www.nasa.gov/planetarydefense/dart/).

Utilizando datos procedentes de la NASA (https://ssd.jpl.nasa.gov/tools/sbdb_query.html) intentaremos evaluar mediante modelos de machine learning si es posible predecir si un asteroide es potencialmente peligroso para nuestro planeta.

1. ANÁLISIS EXPLORATORIO DE LOS DATOS

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelBinarizer, LabelEncoder, OrdinalEncoder, MinM
from sklearn.model_selection import StratifiedShuffleSplit, train_test_split, GridSe
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report,
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import precision_recall_fscore_support as score


import warnings
warnings.filterwarnings('ignore', module='sklearn')
warnings.filterwarnings('ignore', module='IPython')
```

In [2]:

```
filepath = 'C:/Users/NITROPC/Desktop/DATA SCIENCE/CERTIFICACION MACHINE LEARNING/06

data = pd.read_csv(filepath, sep = ',')
```

In [3]:

```
data.head()
```

Out[3]:

	id	spkid	full_name	pdes	name	prefix	neo	pha	H	diameter	...	sigma_i
0	a0000001	2000001	1 Ceres	1	Ceres	NaN	N	N	3.40	939.400	...	4.608900e-09
1	a0000002	2000002	2 Pallas	2	Pallas	NaN	N	N	4.20	545.000	...	3.469400e-06
2	a0000003	2000003	3 Juno	3	Juno	NaN	N	N	5.33	246.596	...	3.223100e-06
3	a0000004	2000004	4 Vesta	4	Vesta	NaN	N	N	3.00	525.400	...	2.170600e-07
4	a0000005	2000005	5 Astraea	5	Astraea	NaN	N	N	6.90	106.699	...	2.740800e-06

5 rows × 45 columns

A continuación se recoge la descripción básica de las columnas, de acuerdo a lo indicado en la web JPL:

- SPK-ID: Object primary SPK-ID
- Object ID: Object internal database ID
- Object fullname: Object full name/designation
- pdes: Object primary designation
- name: Object IAU name
- NEO: Near-Earth Object (NEO) flag
- PHA: Potentially Hazardous Asteroid (PHA) flag
- H: Absolute magnitude parameter
- Diameter: object diameter (from equivalent sphere) km Unit
- Albedo: Geometric albedo
- Diameter_sigma: 1-sigma uncertainty in object diameter km Unit
- Orbit_id: Orbit solution ID
- Epoch: Epoch of osculation in modified Julian day form
- Equinox: Equinox of reference frame
- e: Eccentricity
- a: Semi-major axis au Unit
- q: perihelion distance au Unit
- i: inclination; angle with respect to x-y ecliptic plane
- tp: Time of perihelion passage TDB Unit
- moid_ld: Earth Minimum Orbit Intersection Distance au Unit

Nuestra variable objetivo es la variable PHA, que indica si un asteriode es potencialmente peligroso o no.

In [4]:
```
data.shape
```

Out[4]:
```
(958524, 45)
```

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 958524 entries, 0 to 958523
Data columns (total 45 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   id              958524 non-null  object
 1   spkid           958524 non-null  int64
 2   full_name       958524 non-null  object
 3   pdes            958524 non-null  object
 4   name            22064 non-null   object
 5   prefix          18 non-null      object
 6   neo             958520 non-null  object
 7   pha             938603 non-null  object
 8   H               952261 non-null  float64
 9   diameter        136209 non-null  float64
 10  albedo          135103 non-null  float64
 11  diameter_sigma  136081 non-null  float64
 12  orbit_id        958524 non-null  object
 13  epoch           958524 non-null  float64
 14  epoch_mjd       958524 non-null  int64
 15  epoch_cal       958524 non-null  float64
 16  equinox         958524 non-null  object
```

```
 17  e              958524 non-null  float64
 18  a              958524 non-null  float64
 19  q              958524 non-null  float64
 20  i              958524 non-null  float64
 21  om             958524 non-null  float64
 22  w              958524 non-null  float64
 23  ma             958523 non-null  float64
 24  ad             958520 non-null  float64
 25  n              958524 non-null  float64
 26  tp             958524 non-null  float64
 27  tp_cal         958524 non-null  float64
 28  per            958520 non-null  float64
 29  per_y          958523 non-null  float64
 30  moid           938603 non-null  float64
 31  moid_ld        958397 non-null  float64
 32  sigma_e        938602 non-null  float64
 33  sigma_a        938602 non-null  float64
 34  sigma_q        938602 non-null  float64
 35  sigma_i        938602 non-null  float64
 36  sigma_om       938602 non-null  float64
 37  sigma_w        938602 non-null  float64
 38  sigma_ma       938602 non-null  float64
 39  sigma_ad       938598 non-null  float64
 40  sigma_n        938602 non-null  float64
 41  sigma_tp       938602 non-null  float64
 42  sigma_per      938598 non-null  float64
 43  class          958524 non-null  object
 44  rms            958522 non-null  float64
dtypes: float64(33), int64(2), object(10)
memory usage: 329.1+ MB
```

Veamos nuestra variable objetivo.

In [6]:

```
data['pha'].value_counts(normalize=True)
```

Out[6]:

```
N    0.997799
Y    0.002201
Name: pha, dtype: float64
```

Como podemos ver, los datos de nuestra variable objetivo están muy poco balanceados.

Veamos cómo se distribuyen los valores dentro de cada una de las variables.

In [7]:

```
pd.DataFrame([[i, len(data[i].unique())] for i in data.columns],
             columns=['Variable', 'Unique Values']).set_index('Variable')
```

Out[7]:

	Unique Values
Variable	
id	958524
spkid	958524
full_name	958524
pdes	958524
name	22065
prefix	2
neo	3
pha	3
H	9490

	Unique Values
Variable	
diameter	16592
albedo	1058
diameter_sigma	3055
orbit_id	4690
epoch	5246
epoch_mjd	5246
epoch_cal	5246
equinox	1
e	958444
a	958509
q	958509
i	958414
om	958518
w	958519
ma	958520
ad	958506
n	958514
tp	958519
tp_cal	958499
per	958511
per_y	958512
moid	314301
moid_ld	314302
sigma_e	254741
sigma_a	273298
sigma_q	248139
sigma_i	215742
sigma_om	223156
sigma_w	262720
sigma_ma	266817
sigma_ad	269242
sigma_n	251751
sigma_tp	291247
sigma_per	282688
class	13

	Unique Values
Variable	
rms	64387

2. LIMPIEZA DE DATOS

Eliminemos algunas columnas que no aportarán ningún tipo de información.

In [8]:
```
data1 = data.drop(['id','full_name', 'pdes', 'name', 'prefix', 'equinox'], axis='col
```

Veamos a continuación los datos perdidos dentro de nuestra dataframe.

In [9]:
```
num_missing = data.isnull().sum()
pctg_missing = data.isnull().sum().apply(lambda x: x/data.shape[0]*100)
```

In [10]:
```
missing_data = pd.DataFrame({'Number of Missing':  num_missing,
                             'Percentage of Missing': pctg_missing})

missing_data['Percentage of Missing'].sort_values(ascending = False)
```

Out[10]:
```
prefix            99.998122
name              97.698128
albedo            85.905100
diameter_sigma    85.803068
diameter          85.789714
sigma_ad           2.078821
sigma_per          2.078821
sigma_e            2.078404
sigma_a            2.078404
sigma_q            2.078404
sigma_i            2.078404
sigma_om           2.078404
sigma_w            2.078404
sigma_ma           2.078404
sigma_n            2.078404
sigma_tp           2.078404
pha                2.078300
moid               2.078300
H                  0.653400
moid_ld            0.013250
per                0.000417
ad                 0.000417
neo                0.000417
rms                0.000209
ma                 0.000104
per_y              0.000104
class              0.000000
id                 0.000000
tp_cal             0.000000
equinox            0.000000
full_name          0.000000
pdes               0.000000
orbit_id           0.000000
epoch              0.000000
epoch_mjd          0.000000
epoch_cal          0.000000
e                  0.000000
tp                 0.000000
a                  0.000000
q                  0.000000
i                  0.000000
```

```
om                  0.000000
spkid               0.000000
n                   0.000000
w                   0.000000
Name: Percentage of Missing, dtype: float64
```

Como podemos ver hay cinco variables con valores perdidos por encima del 80%. Estas variables no pueden ser calculadas ni sustituidas, por lo que pueden ser eliminadas.

In [11]:

```
asteroid_df = data1[data1['pha'].notna()]
asteroid_df = asteroid_df.drop(['diameter', 'albedo', 'diameter_sigma'], axis= 'colu
```

In [12]:

```
asteroid_df = asteroid_df[asteroid_df['H'].notna()]
asteroid_df = asteroid_df[asteroid_df['sigma_ad'].notna()]
asteroid_df = asteroid_df[asteroid_df['ma'].notna()]
```

In [13]:

```
asteroid_df.isnull().sum()
```

Out[13]:

```
spkid        0
neo          0
pha          0
H            0
orbit_id     0
epoch        0
epoch_mjd    0
epoch_cal    0
e            0
a            0
q            0
i            0
om           0
w            0
ma           0
ad           0
n            0
tp           0
tp_cal       0
per          0
per_y        0
moid         0
moid_ld      0
sigma_e      0
sigma_a      0
sigma_q      0
sigma_i      0
sigma_om     0
sigma_w      0
sigma_ma     0
sigma_ad     0
sigma_n      0
sigma_tp     0
sigma_per    0
class        0
rms          0
dtype: int64
```

Como podemos ver, ya no tenemos valores nulos en nuestro dataframe.

Algunas columnas presentan valores que no pueden ser procesados dentro de un modelo de machine learning. Estas variables tienen que ser convertidas a variables categóricas.

In [14]:

```
asteroid_df['neo'] = asteroid_df['neo'].astype('category')
asteroid_df['pha'] = asteroid_df['pha'].astype('category')
```

```
asteroid_df['class'] = asteroid_df['class'].astype('category')
```

In [15]:
```
orbits = asteroid_df['orbit_id'].value_counts().loc[lambda x: x<10].index.to_list()
```

In [16]:
```
asteroid_df.loc[asteroid_df['orbit_id'].isin(orbits), 'orbit_id'] = 'other'
```

Observemos la correlación entre las distintas variables.

In [61]:
```
plt.figure(figsize = (30, 15))
sns.heatmap(asteroid_df.corr(), annot = False, linewidths=.5, cmap = plt.cm.cool)
```

Out[61]: <AxesSubplot:>

Antes de hecer pasar nuestro dataframe por el modelo, es necesario escalar las variables numéricas. Para ello utilizaremos la función MinMaxScaler.

In [17]:
```
asteroid_df = asteroid_df.reset_index(drop=True)#Reseteamos el índice
```

In [18]:
```
#creamos un subset con las variables numericas
subset_df = asteroid_df[asteroid_df.columns[~asteroid_df.columns.isin(['spkid', 'ful
```

In [19]:
```
scaler = MinMaxScaler()
scaled_df = scaler.fit_transform(subset_df)
scaled_df = pd.DataFrame(scaled_df, columns = subset_df.columns)
asteroid_df = pd.concat([asteroid_df[['spkid', 'neo', 'pha', 'orbit_id', 'class']],s
scaled_df.head()
```

Out[19]:

	H	epoch	epoch_mjd	epoch_cal	e	a	q	i	om	
0	0.131195	0.988218	0.988218	0.989134	0.076017	0.000066	0.030975	0.060467	0.223071	0.204
1	0.154519	1.000000	1.000000	1.000000	0.230004	0.000066	0.025712	0.198916	0.480625	0.861
2	0.187464	1.000000	1.000000	1.000000	0.256972	0.000063	0.023805	0.074158	0.471810	0.689

	H	epoch	epoch_mjd	epoch_cal	e	a	q	i	om	
3	0.119534	0.988218	0.988218	0.989134	0.088732	0.000054	0.025911	0.040748	0.288363	0.41
4	0.233236	1.000000	1.000000	1.000000	0.190939	0.000060	0.025049	0.030614	0.393253	0.99

5 rows × 31 columns

Por último, necesitamos transformar las variables categóricas en variables codificadas. Para ello usaremos la función get_dummies

In [20]:

```
asteroid_df1 = pd.get_dummies(asteroid_df, columns = ['neo', 'class', 'orbit_id'])
asteroid_df1.head()
```

Out[20]:

	spkid	pha	H	epoch	epoch_mjd	epoch_cal	e	a	q	i
0	2000001	N	0.131195	0.988218	0.988218	0.989134	0.076017	0.000066	0.030975	0.060467
1	2000002	N	0.154519	1.000000	1.000000	1.000000	0.230004	0.000066	0.025712	0.198916
2	2000003	N	0.187464	1.000000	1.000000	1.000000	0.256972	0.000063	0.023805	0.074158
3	2000004	N	0.119534	0.988218	0.988218	0.989134	0.088732	0.000054	0.025911	0.040748
4	2000005	N	0.233236	1.000000	1.000000	1.000000	0.190939	0.000060	0.025049	0.030614

5 rows × 242 columns

In [21]:

```
lb = LabelBinarizer()

asteroid_df1['pha'] = lb.fit_transform(asteroid_df1['pha'])
```

In [22]:

```
asteroid_df1.head()
```

Out[22]:

	spkid	pha	H	epoch	epoch_mjd	epoch_cal	e	a	q	i
0	2000001	0	0.131195	0.988218	0.988218	0.989134	0.076017	0.000066	0.030975	0.060467
1	2000002	0	0.154519	1.000000	1.000000	1.000000	0.230004	0.000066	0.025712	0.198916
2	2000003	0	0.187464	1.000000	1.000000	1.000000	0.256972	0.000063	0.023805	0.074158
3	2000004	0	0.119534	0.988218	0.988218	0.989134	0.088732	0.000054	0.025911	0.040748
4	2000005	0	0.233236	1.000000	1.000000	1.000000	0.190939	0.000060	0.025049	0.030614

5 rows × 242 columns

In [23]:

```
outputfile = 'asteroid_processed.csv'
asteroid_df1.to_csv(outputfile, index=False)
```

3. DIVIDIR LOS DATOS

Antes de preparar nuestros modelos de clasificiación necesitamos dividir nuestros datos en los sets de entrenamiento y prueba. Como nuestro dataframe presenta datos muy sesgados para nuestra variable objetivo, vamos a utilizar la función StratifiedShuffleSplit para mantener la misma proporción de clases.

In [24]:
```
path = 'C:/Users/NITROPC/Desktop/DATA SCIENCE/CERTIFICACION MACHINE LEARNING/06 - PR

asteroid_processed = pd.read_csv(path, sep = ',')
```

In [25]:
```
feature_cols = list(asteroid_processed.columns)
feature_cols.remove('pha')
```

In [26]:
```
X_data = asteroid_processed.drop(['spkid', 'pha'], axis = 1)
y_data = asteroid_processed['pha']
```

In [27]:
```
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, test_size = 0.3,
```

In [28]:
```
X_test.shape
```

Out[28]:
```
(279701, 240)
```

In [29]:
```
y_test.shape
```

Out[29]:
```
(279701,)
```

In [30]:
```
X_train.shape
```

Out[30]:
```
(652634, 240)
```

In [31]:
```
y_train.shape
```

Out[31]:
```
(652634,)
```

Dado que nuestros datos presentan un sesgo muy fuerte hacia el No en la variable 'pha' necesitamos que nuestro modelo no sobremuestre la clase positiva. Utilizaremos para ello la libreria SMOTE.

In [32]:
```
from imblearn.over_sampling import SMOTE

smtn = SMOTE(random_state = 12)

X_train_res, y_train_res = smtn.fit_resample(X_train, y_train)
```

In [33]:
```
print("Before OverSampling, counts of label 'N': {}".format(sum(y_train == 0)))
print("Before OverSampling, counts of label 'Y': {} \n".format(sum(y_train == 1)))

print("After OverSampling, counts of label 'N': {}".format(sum(y_train_res == 0)))
print("After OverSampling, counts of label 'Y': {}".format(sum(y_train_res == 1)))
```

```
Before OverSampling, counts of label 'N': 651221
Before OverSampling, counts of label 'Y': 1413

After OverSampling, counts of label 'N': 651221
After OverSampling, counts of label 'Y': 651221
```

4. REGRESIÓN LOGISTICA

Vamos a comenzar nuestros modelos con el modelado por regresión logística.

In [34]:
```
#creamos el dataframe para nuestras métricas
metrics = pd.DataFrame()
```

In [35]:
```
#REgresión logística estandar
lr = LogisticRegression().fit(X_train_res, y_train_res)
y_pred_lr = lr.predict(X_test)
```

Para el modelo vamos a mostrar las métricas asociadas y la matriz de confusión.

In [36]:
```
precision_lr, recall_lr = (round(float(x),2) for x in list(score(y_test,
                                                                  y_pred_lr,
                                                                  average='weighte
# adding lr stats to metrics DataFrame
lr_stats = pd.Series({'precision':precision_lr,
                      'recall':recall_lr,
                      'accuracy':round(accuracy_score(y_test, y_pred_lr), 2),
                      'f1score':round(f1_score(y_test, y_pred_lr), 2),
                      'auc': round(roc_auc_score(y_test, y_pred_lr),2)},
                     name='Logistic Regression')
# Report outcomes
pd.DataFrame(classification_report(y_test, y_pred_lr, output_dict=True)).iloc[:3,:2]
```

Out[36]:

	0	1
precision	0.999931	0.208279
recall	0.991363	0.970904
f1-score	0.995629	0.342981

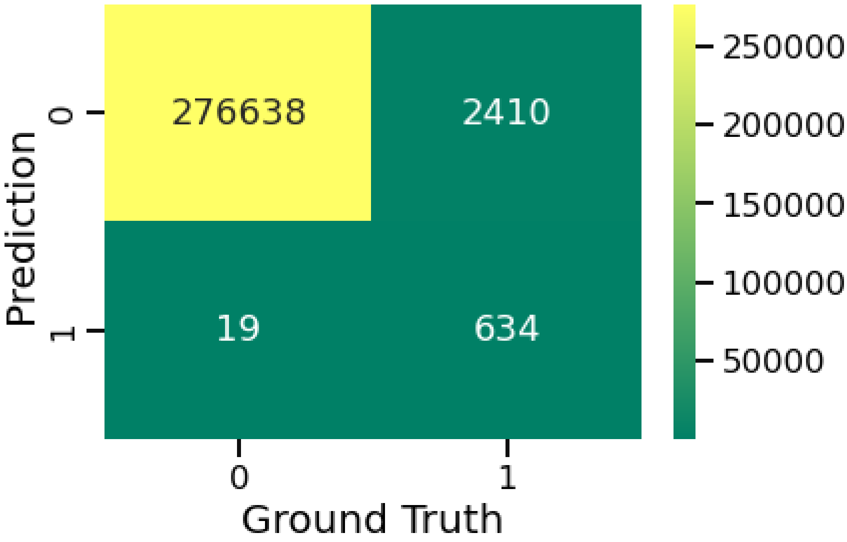
In [37]:
```
lr_stats
```

Out[37]:
```
precision    1.00
recall       0.99
accuracy     0.99
f1score      0.34
auc          0.98
Name: Logistic Regression, dtype: float64
```

In [38]:
```
sns.set_context('talk')
cm = confusion_matrix(y_test, y_pred_lr)
ax = sns.heatmap(cm, annot=True, fmt='d', cmap='summer')


ax.set_ylabel('Prediction', fontsize=20)
ax.set_xlabel('Ground Truth', fontsize=20)
```

Out[38]:
```
Text(0.5, 4.5, 'Ground Truth')
```



5. RANDOM FOREST

Nuestro siguiente modelo será Random Forest. Este modelo elimina parte de la posibilidad de sobreajuste.

Podríamos analizar el número de árboles iterando entre varios valores y posteriormente graficar el error para conocer con qué valor se estabiliza el error:

```
rf = RandomForestClassifier(oob_score = True, warm_start = True,
n_jobs = -1, random_state = 1551)

oob_list = list()

for n_trees in [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]:

    rf.set_params(n_estimators = n_trees)
    rf.fit(X_train_res, y_train_res)

    oob_error = 1 - rf.oob_score_
    oob_list.append(pd.Series({'n_trees' : n_trees, 'oob' :
oob_error}))

rf_oob_df = pd.concat(oob_list, axis = 1).T.set_index('n_trees')

sns.set_context('talk')
sns.set_style('white')

ax = rf_oob_df.plot(legend=False, marker='o', color="green", figsize=
(14, 7), linewidth=4)
ax.set(ylabel='out-of-bag error');
```

Sin embargo, para ahorrar tiempo de procesado, vamos a aplicar un valor de 150 a n_estimators.

In [39]:
```
rf = RandomForestClassifier(n_estimators = 150, oob_score = True, warm_start = True,
```

In [40]:
```
rf.fit(X_train_res, y_train_res)
```

```
RandomForestClassifier(n_estimators=150, n_jobs=-1, oob_score=True,
```

Out[40]:
```
                    random_state=1551, warm_start=True)
```

In [41]:
```
y_pred_rf = rf.predict(X_test)
```

In [42]:
```
precision_rf, recall_rf = (round(float(x),2) for x in list(score(y_test,
                                                                  y_pred_rf,
                                                                  average='weighte

rf_stats = pd.Series({'precision':precision_rf,
                      'recall':recall_rf,
                      'accuracy':round(accuracy_score(y_test, y_pred_rf), 2),
                      'f1score':round(f1_score(y_test, y_pred_rf), 2),
                      'auc': round(roc_auc_score(y_test, y_pred_rf),2)}, name='Rando

pd.DataFrame(classification_report(y_test, y_pred_rf, output_dict=True)).iloc[:3,:2]
```

Out[42]:

	0	1
precision	0.999968	0.947059
recall	0.999871	0.986217
f1-score	0.999919	0.966242

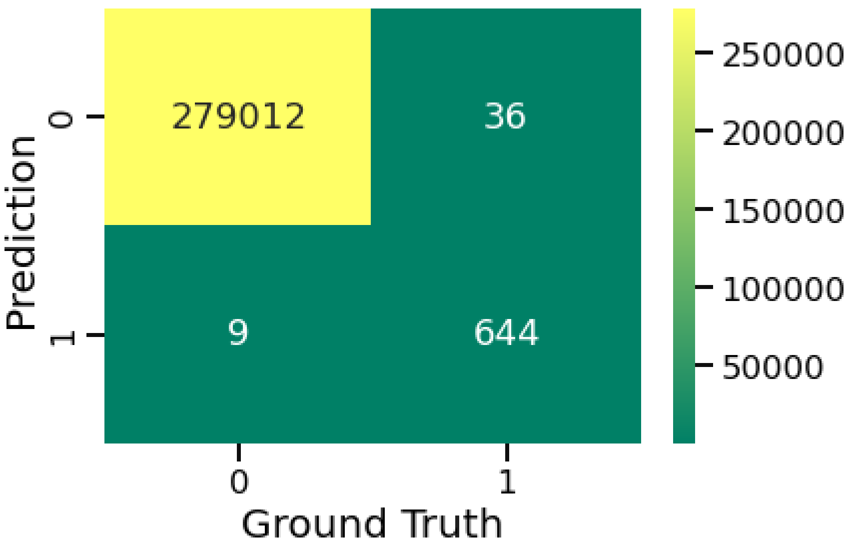
In [43]:
```
rf_stats
```

Out[43]:
```
precision    1.00
recall       1.00
accuracy     1.00
f1score      0.97
auc          0.99
Name: Random Forest, dtype: float64
```

In [44]:
```
sns.set_context('talk')
cm_rf = confusion_matrix(y_test, y_pred_rf)
ax = sns.heatmap(cm_rf, annot=True, fmt='d', cmap='summer')


ax.set_ylabel('Prediction', fontsize=20)
ax.set_xlabel('Ground Truth', fontsize=20)
```

Out[44]:
```
Text(0.5, 4.5, 'Ground Truth')
```



6.POTENCIADOR DE GRADIENTE

Por último vamos a aplicar un modelo de potenciador de gradiente. Esta técnica combina los principios de la potenciación de gradiente con la aletoriedad de los árboles de decisión.

In [48]:
```
from sklearn.ensemble import GradientBoostingClassifier
```

Al igual que en el modelo Random Forest podríamos analizar el número de árboles iterando entre varios valores y posteriormente graficar el error para conocer con qué valor se estabiliza:

```
error_list = list()

tree_list = [15, 25, 50, 100, 200, 400]
for n_trees in tree_list:

    GBC = GradientBoostingClassifier(n_estimators=n_trees,
random_state=42)

    print(f'Fitting model with {n_trees} trees')
    GBC.fit(X_train_res.values, y_train_res.values)
    y_pred = GBC.predict(X_test)

    error = 1.0 - accuracy_score(y_test, y_pred)
    Store it
    error_list.append(pd.Series({'n_trees': n_trees, 'error': error}))

error_df = pd.concat(error_list, axis=1).T.set_index('n_trees')

error_df
```

Para ahorrar tiempo de procesado, vamos a aplicar un valor de 150 a n_estimators.

In [49]:
```
gbc = GradientBoostingClassifier(n_estimators = 400, learning_rate = 0.1, subsample
```

In [50]:
```
gbc.fit(X_train_res, y_train_res)
```

Out[50]: GradientBoostingClassifier(max_features=4, n_estimators=400, subsample=0.5)

In [51]:
```
y_pred_gbc = gbc.predict(X_test)
```

In [57]:
```
precision_gbc, recall_gbc = (round(float(x),2) for x in list(score(y_test,
                                                                   y_pred_rf,
                                                                   average='weighte

gbc_stats = pd.Series({'precision':precision_gbc,
                      'recall':recall_rf,
                      'accuracy':round(accuracy_score(y_test, y_pred_gbc), 2),
                      'f1score':round(f1_score(y_test, y_pred_gbc), 2),
                      'auc': round(roc_auc_score(y_test, y_pred_gbc),2)}, name='Rand

pd.DataFrame(classification_report(y_test, y_pred_gbc, output_dict=True)).iloc[:3,:2
```

Out[57]:

	0	1
precision	0.999989	0.705755
recall	0.999029	0.995406
f1-score	0.999509	0.825921

In [58]:

```
gbc_stats
```

Out[58]:

```
precision     1.00
recall        1.00
accuracy      1.00
f1score       0.83
auc           1.00
Name: Random Forest, dtype: float64
```

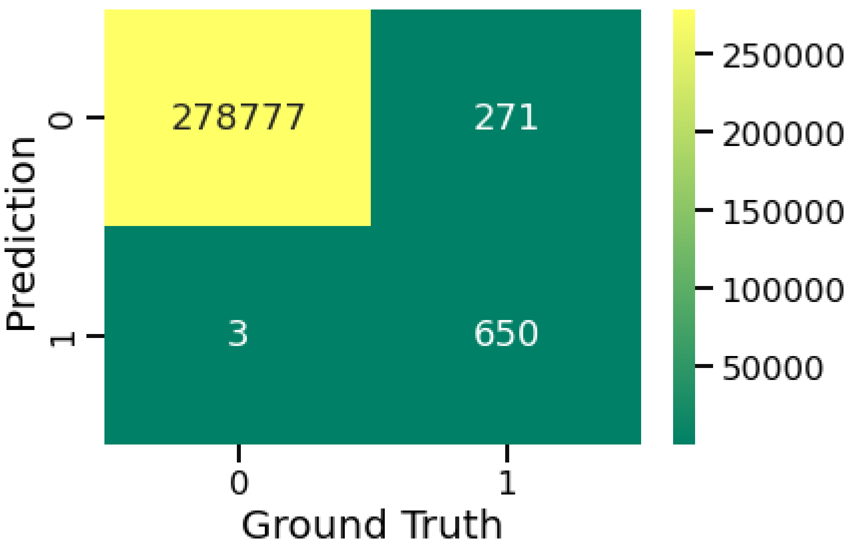
In [55]:

```
sns.set_context('talk')
cm_rf = confusion_matrix(y_test, y_pred_gbc)
ax = sns.heatmap(cm_rf, annot=True, fmt='d', cmap='summer')


ax.set_ylabel('Prediction', fontsize=20)
ax.set_xlabel('Ground Truth', fontsize=20)
```

Out[55]:

```
Text(0.5, 4.5, 'Ground Truth')
```



7. CONCLUSIONES

A continuación se resumen las métricas para cada uno de los modelos aplicados:

In [59]:

```
metrics.append([lr_stats, rf_stats, gbc_stats])
```

Out[59]:

	precision	recall	accuracy	f1score	auc
Logistic Regression	1.0	0.99	0.99	0.34	0.98
Random Forest	1.0	1.00	1.00	0.83	1.00
Random Forest	1.0	1.00	1.00	0.83	1.00

Todos los modelos presentan unos valores similares. Sin embargo, dada la naturaleza sesgada de los datos, se hace necesario el uso de algoritmos de potenciamiento y validadción cruzada, que suponen un mayor tiempo de procesado.