Analisis de Carteras de Ethereum

Juli Sahun
Victor Pla

7 de enero de 2023

Índice

1. Introduction

En este trabajo, intentaremos predecir si una cartera de Ethereum es verídica o si, por lo contrario, contiene algún dato que la hace fraudulenta. Para lograr esto, usaremos diversos modelos estudiados en clase y mediremos su precisión.

Con la ayuda de la librería sklearn implementaremos 3 modelos lineales y 3 modelos no lineales.

2. Los Datos

Hemos extraído los datos de, aproximadamente, 12000 cartera de Ethereum de la página web kaggel. En este apartado, describiremos de qué datos disponemos y sus tipos. También veremos algún ejemplo del *dataset*, como están distribuidos y el preanálisis que les hemos aplicado.

2.1. Descripcion de los datos

A continuación describiremos con qué datos contamos de cada una de las carteras:

- address: contiene la dirección de la cartera → *string*
- flag: representa si bien la cartera es verídica (0) o si, por lo contrario, es fraudulenta (0) → *boolean*
- min/max/avgTimeBetweeSentTnx: estos tres campos contienen el tiempo mínimo, máximo y medio entre transacciones enviadas de la cartera→ *float, float, float*
- min/max/avgTimeBetweeRecTnx: estos tres campos contienen el tiempo mínimo, máximo y medio entre transacciones recibidas de la cartera → *float, float, float*
- lifetime: desfcribe el tiempo que esta cartera ha estado activa → *float*
- sentTransactions: contiene el número total de transacciones enviadas de la cartera→ *Integer*
- receivedTransactions: contiene el número total de transacciones recibidas de la cartera → *Integer*
- createdContracts: representa el número de contratos creados para la certera → *Integer*
- numUniqSentAddress: representa el número de transacciones enviadas a direcciones únicas. → *Integer*
- numUniqRecAddress: representa el número de transacciones recibidas de direcciones únicas. → *Integer*
- min/max/avgValSent: estos tres campos contienen el valor mínimo, máximo y medio de las transacciones enviadas → *float*
- min/max/avgValSent: estos tres campos contienen el valor mínimo, máximo y medio de las transacciones recibidas → *float*
- totalTransactions: representa el número total de transacciones realizadas desde la cartera en cuestión → *float*
- totalEtherSent: representa el número total de Ethereum enviado→ *float*
- totalEtherReceived: representa el número total de Ethereum recibido → *float*
- totalEtherSentContracts: contiene el número total de Ethereum enviado a través de contratos → *float*
- totalEtherBalance: contiene el número total de Ethereum de la cartera→ *float*
- activityDays: registra el total de días de actividad de la cartera→ *float*
- dailyMax: registra el número total de transacciones realizadas en un día→ *float*

- ratioRecSent: contiene la proporción de Ethereum recibido respecto al enviado→ *float*
- ratioSentTotal: contiene la proporción de Ethereum enviado respecto al total→ *float*
- ratioRecTotal: contiene la proporción de Ethereum recibido respecto al total→ *float*
- giniSent: representa el índice de gini para las transacciones enviadas → *float*
- giniRec: representa el índice de gini para las transacciones recibidas → *float*
- txFreq: representa la frecuencia con la que se realizan transacciones para esa cartera → *float*
- stdBalanceEth: contiene valor estandarizado de Ethereum de la cartera → *float*

2.2. Visualización de los datos

Una vez hemos visto de qué datos disponemos, podemos ver algún ejemplo del data set:

```
data.head()
```

	address	flag	minTimeBetweenSentTnx	maxTimeBetweenSentTnx	avgTimeBetweenSentTnx
0	0xd0cc2b24980cbcca47ef755da88b220a82291407	1	0.0	2.387e+06	5.808e+04
1	0x4cdc1cba0aeb5539f2e0ba158281e67e0e54a9b1	1	0.0	0.000e+00	0.000e+00
2	0x00e01a648ff41346cdeb873182383333d2184dd1	1	37.0	2.511e+07	1.710e+06
3	0x858457daa7e087ad74cdeeceab8419079bc2ca03	1	0.0	6.425e+05	1.576e+04
4	0x240e125c20a4cc84bd6e7f8d1fd07aff4c06d43d	1	0.0	0.000e+00	0.000e+00

Figura 1: Contenido de los 5 primeros ejemplos del *dataset*

Podemos ahora ver alguna información más general de los datos, como el número total de valores que tenemos, la media, el valor máximo, entre otras cosas.

```
data.describe(include='all').T
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
address	12146	12146	0xf031f853acac153c015ae52d0bd3591a84826dca	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
flag	1.21e+04	NaN	NaN	NaN	0.424	0.494	0	0	0	1	1
minTimeBetweenSentTnx	1.21e+04	NaN	NaN	NaN	2.65e+04	7.17e+05	0	0	0	80	6.2e+07
maxTimeBetweenSentTnx	1.21e+04	NaN	NaN	NaN	4.27e+07	2.51e+08	0	0	2.04e+05	1.85e+06	1.66e+09
avgTimeBetweenSentTnx	1.21e+04	NaN	NaN	NaN	3.65e+06	4.12e+07	0	0	1.49e+04	1.39e+05	8.29e+08
minTimeBetweenRecTnx	1.21e+04	NaN	NaN	NaN	2.5e+05	3.4e+06	0	0	48	715	1.51e+08
maxTimeBetweenRecTnx	1.21e+04	NaN	NaN	NaN	1.16e+09	7.16e+08	0	2.79e+06	1.58e+09	1.64e+09	1.66e+09
avgTimeBetweenRecTnx	1.21e+04	NaN	NaN	NaN	1.6e+08	2.33e+08	0	8.56e+04	4.78e+07	2.19e+08	8.31e+08
lifetime	1.21e+04	NaN	NaN	NaN	192	300	1	9	68	251	2.29e+03
sentTransactions	1.21e+04	NaN	NaN	NaN	517	1.88e+03	0	1	6	70	1e+04
receivedTransactions	1.21e+04	NaN	NaN	NaN	678	2.25e+03	0	3	11	55	1e+04
createdContracts	1.21e+04	NaN	NaN	NaN	1.76	106	0	0	0	0	9.95e+03
numUniqSentAddress	1.21e+04	NaN	NaN	NaN	99.6	623	0	1	3	17	9.95e+03
numUniqRecAddress	1.21e+04	NaN	NaN	NaN	278	1.12e+03	0	2	5	16	1e+04
minValSent	1.21e+04	NaN	NaN	NaN	6.25	248	0	0	0	0.0523	2.55e+04
maxValSent	1.21e+04	NaN	NaN	NaN	167	6.05e+03	0	0.0214	1.01	7.03	6.11e+05
avgValSent	1.21e+04	NaN	NaN	NaN	12.1	296	0	0.00404	0.138	1.3	2.55e+04
minValReceived	1.21e+04	NaN	NaN	NaN	13.7	495	0	0	0.0064	0.0575	2.55e+04
maxValReceived	1.21e+04	NaN	NaN	NaN	265	8.2e+03	0	0.12	1	5	8e+05
avgValReceived	1.21e+04	NaN	NaN	NaN	52.1	2.67e+03	0	0.042	0.25	1.04	2.84e+05
totalTransactions	1.21e+04	NaN	NaN	NaN	1.2e+03	2.94e+03	1	7	34	282	1.99e+04
totalEtherSent	1.21e+04	NaN	NaN	NaN	3.06e+03	9.27e+04	0	0.0335	3	23.8	6.81e+06
totalEtherReceived	1.21e+04	NaN	NaN	NaN	2.92e+03	5.82e+04	0	0.305	3.8	23.3	2.96e+06
totalEtherSentContracts	1.21e+04	NaN	NaN	NaN	0.00369	0.354	0	0	0	0	39
totalEtherBalance	1.21e+04	NaN	NaN	NaN	-137	6.58e+04	-5.35e+06	-0.353	0.00156	0.0406	1.85e+06
activityDays	1.21e+04	NaN	NaN	NaN	85.5	173	1	3	15	85	1.91e+03
dailyMax	1.21e+04	NaN	NaN	NaN	209	975	1	2	5	24	1e+04
ratioRecSent	1.21e+04	NaN	NaN	NaN	581	2.21e+03	0	0.23	0.62	2.74	1e+04
ratioSentTotal	1.21e+04	NaN	NaN	NaN	0.459	0.332	0	0.143	0.455	0.774	1
ratioRecTotal	1.21e+04	NaN	NaN	NaN	0.452	0.318	0	0.187	0.375	0.714	1
giniSent	1.21e+04	NaN	NaN	NaN	0.434	0.375	-6.74e-17	0	0.494	0.801	1
giniRec	1.21e+04	NaN	NaN	NaN	0.41	0.301	-9.32e-17	0.0971	0.446	0.635	1
txFreq	1.21e+04	NaN	NaN	NaN	64.6	544	0.00114	0.278	1.08	4	1e+04
stdBalanceEth	1.21e+04	NaN	NaN	NaN	697	2.43e+04	0	0.0836	0.934	6.83	2.04e+06

Figura 2: descripción de los datos del *dataset*

Aquí podemos ver que no tenemos ningún valor perdido, de todos los campos tenemos *count=12146*, nos podemos dar cuenta que muchas de las variables contienen valores extremos, podemos ver que de la mayoría de variables tienen muchos de los valores en 0. También podemos ver que la columna *address* es totalmente irrelevante, ya que no solo contiene información incomparable, sino que cada cartera tiene una diferente.

A continuación, separaremos en dos conjuntos, los datos de *train* y los datos de *test*. Los datos de *train* serán los usados para entrenar cada uno de los modelos, mientras que los datos de *test* serán usados para evaluar la calidad de los modelos.

```
X = data.iloc[:,1:]
y = data.loc[:,'flag'].copy()
target_names=['Normal', 'Fraudulent']
```

separamos los datos en variables de información (X) y la variable a predecir (y), en nuestro caso, *flag*

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.33, random_state=42, stratify=y)
```

Para poder visualizar la distribución de los datos, hemos generado una gráfica por cada variable, donde podemos ver la frecuencia de cada valor de esa variable.

```
fig, axes = plt.subplots(8,4,figsize=(20,50))
X_train_nr = X_train_n.round(3)
X_train_frame = pd.DataFrame(X_train_nr)
X_train_frame.columns=X.columns
for i, c in enumerate(X.columns):
    ax = axes.reshape(-1)[i]
    a = sns.histplot(x=c,data=X_train_frame,ax=ax)
plt.tight_layout()
```

Podemos ver que para esta prueba, se está usando la variable *XTrainN.round(3)* esto corresponde a los datos de *XTrain* normalizados y redondeados, esto se ha hecho por temas de eficiencia, lo podremos ver en más detalle en el apartado de analisi de los datos

Podemos ver esta figura en los anexos

Otra forma de visualizar los datos es hacerlo mediante las relaciones entre ellos, para ello usaremos un *heatmap* para ver cuanto de correlacionadas están cada una de las variables con las demás del conjunto

```
corr = X_train.corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
plt.subplots(figsize=(10, 8))
sns.heatmap(
    corr,
    mask=mask,
    cmap='seismic',
    center=0,
    square=True,
    linewidths=.5,
    cbar_kws={"shrink": .5})
```

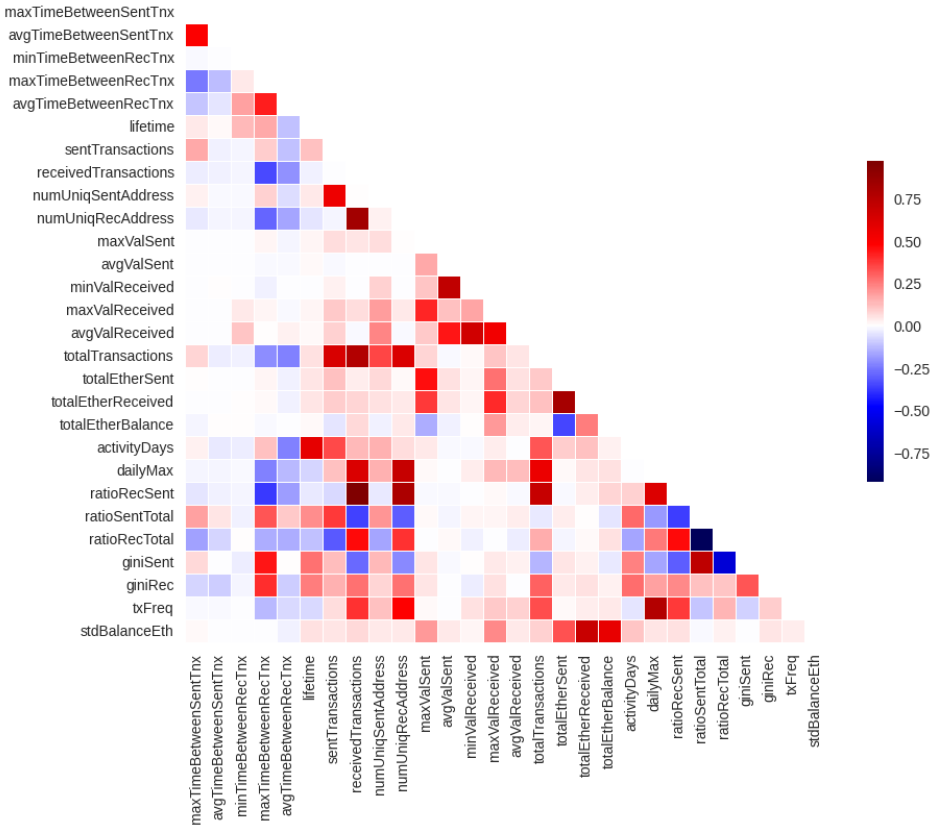

Figura 3: Mapa de calor de la relacion entre las variables

Podemos ver que existen relaciones entre las variables, pero no parece nada exagerado, así que no quitaremos ninguna.

Por último, podemos ver cuan relacionadas están cada una de las variables con la variable Objetivo *flag*

```
plt.figure(figsize=(10,8))
visualizer =
    feature_correlation(
        X_train,
        y_train,
        labels=list(X_train.columns),
        method='mutual_info-classification'
        )
```

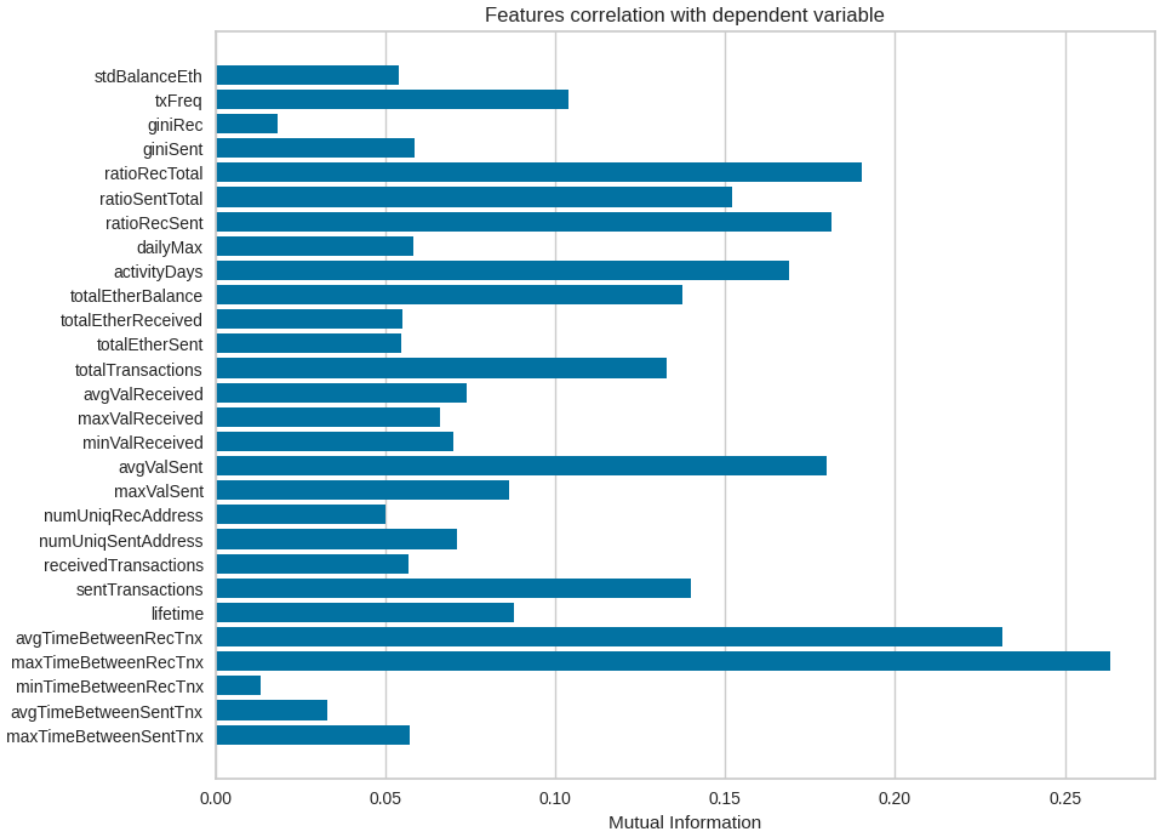

Figura 4: Relación de cada una de las variables con la variable objetivo

Podemos ver que la variable más relacionada con *flag* es el máximo tiempo entre recibo de transacciones. No obstante, la diferencia no es muy relevante con las demás variables, veremos si los modelos coinciden con estas conclusiones.

2.3. Analisi de los datos

En este apartado, veremos las transformaciones que les hemos realizado a los datos para poder trabajar con ellos con mayor comodida y eficiencia.
Para empezar, dado que el rango de valores entre variables es muy dispar, hemos normalizado todas las variables para que tomen valores entre 0 y 1.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

X_train_n = scaler.fit_transform(X_train)
X_test_n = scaler.transform(X_test)
```

Una vez con los datos normalizados, hemos realizado un análisis de las variables usando el método PCA.

```
pca = PCA().fit(X_train_n)
fig = plt.figure(figsize=(8,6))
plt.plot(
    range(1,len(pca.explained_variance_ratio_)+1),
    pca.explained_variance_ratio_,
    alpha=0.8,
    marker='.',
    label="Variancia Explicada")
```

```
plt.plot(
    range(1,len(pca.explained_variance_ratio_)+1),
    np.cumsum(pca.explained_variance_ratio_),
    c='red',
    marker='.',
    label="Variancia explicada acumulativa")

plt.legend();
```

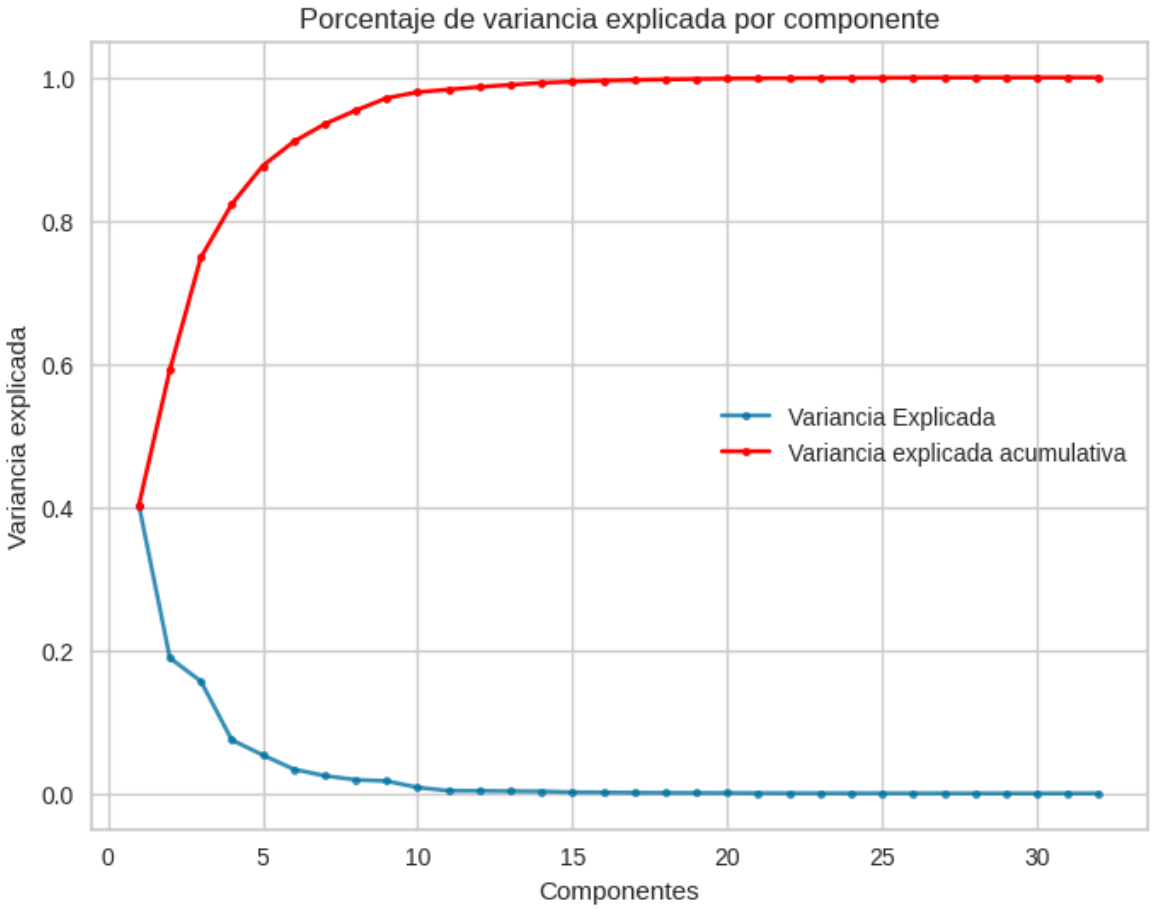

Figura 5: Variancia explicada en función de las dimensiones del PCA

Vemos que el resultado no es del todo malo, con 5 dimensiones podríamos explicar casi el 90 % de la varianza de los datos.

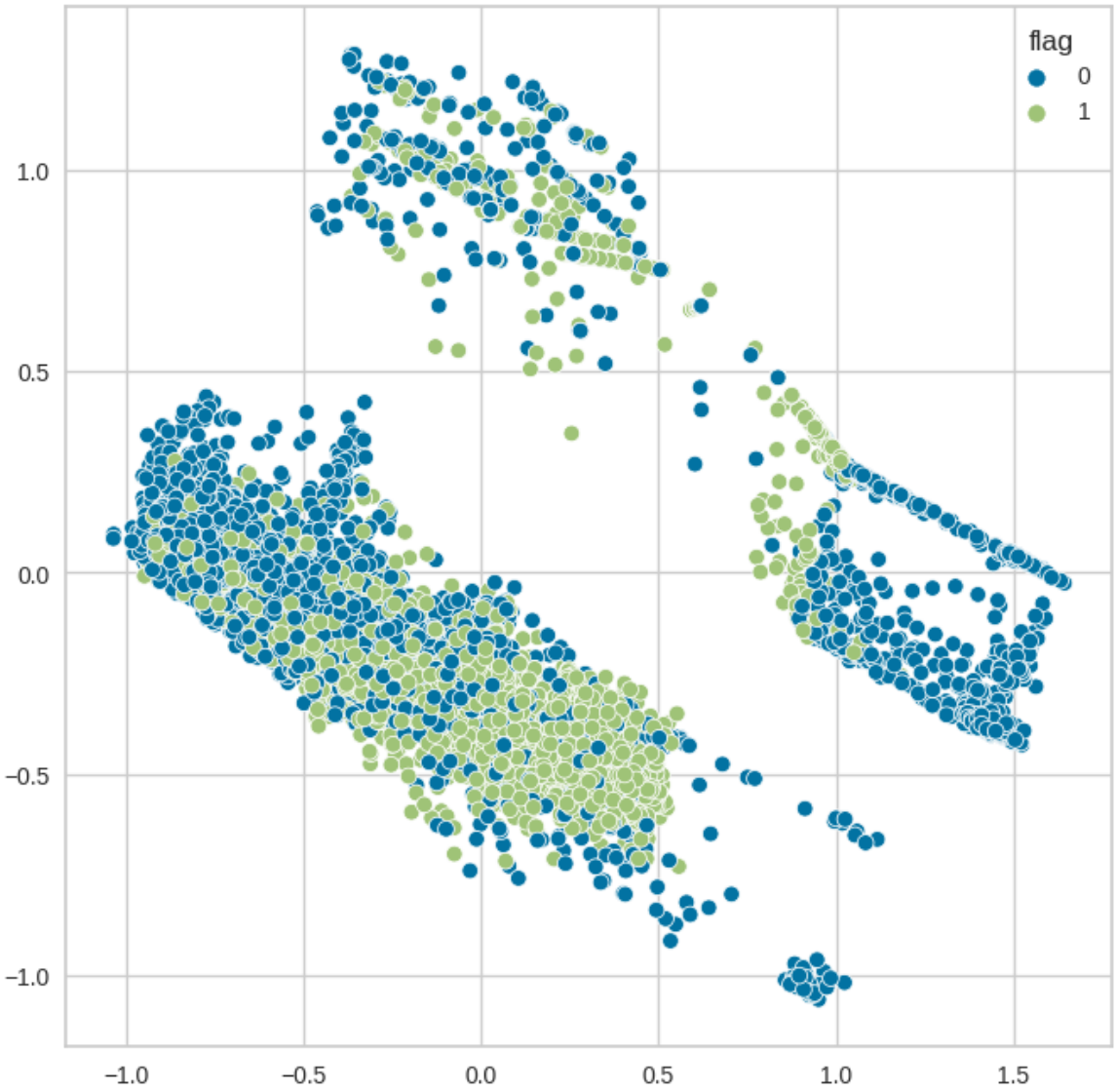

Figura 6: Representación en 2 dimensiones de los datos

Aquí podemos ver una representación de los datos usando las dos primeras dimensiones del PCA. Podemos ver que existe algún tipo de separabilidad, pero esta es muy débil.
Puesto que el número de variables de las que disponemos no es mu elevado, y la reducción obtenida usando el PCA no es muy significativa, usaremos las variables tal cual para entrenar los modelos.

3. Entrenamiento de modelos

En este apartado del trabajo, veremos como hemos entrenado cada uno de los modelos que usaremos, así como los resultados obtenidos, también veremos alguna métrica para entender cuan bueno es cada modelo, más allá de un número de precisión. Hemos dividido los modelos a evaluar en dos grupos, los modelos lineales y los modelos no lineales.

3.1. Como evaluaremos los modelos

Antes de ponernos a entrenar ningún modelo, tenemos que tener claro todos los parámetros a tener en cuenta para su evaluación. Usualmente, primero evaluaremos el modelo con el conjunto de entrenamiento mediante validación cruzada.

Este método consiste en dividir el conjunto de datos en partes iguales (usualmente 10) i entrenar el modelo con 9 de esas partes y evaluarlo con la restante. Una vez realizadas todas las permutaciones,

sacaremos la media de esas evaluaciones. Esta validación la usaremos solo con el conjunto de **train**. Este resultado nos dará una idea de cuan bueno será el modelo, y si merece la pena usarlo para predecir nuestros datos.

Para poder evaluar el modelo de forma correcta, tenemos que entrenarlo con los datos de *train* y luego predecir con los datos de *test* y evaluar estas predicciones. Para medir los resultados, usaremos el *classification report* que nos genera una evaluación respecto distintos parametros.

	precision	recall	f1−score	support
Normal	0.77	0.82	0.79	2152
Fraudulent	0.77	0.71	0.74	1857
accuracy			0.77	4009
macro avg	0.77	0.77	0.77	4009
weighted avg	0.77	0.77	0.77	4009

Aquí tenemos un ejemplo del resultado. Lo primero que podemos ver es que está dividido en 4 columnas: *precision*, *recall*, *f1-score* y *support*.

La precisión corresponde a la proporción entre positivos y los clasificados como tal, es decir que si tenemos que la precisión de carteras normales es del 0.77, sígnica que solamente el 77 % de las carteras que realmente son normales, las hemos clasificado como tal. (vp/p')

La recuperación (*recall*) hace referencia a qué proporción de verdaderos positivos se encuentran en la clase predicha, es decir, que si tenemos un recall de 0.82 para las carteras normales, significa que solo un 82 % de las carteras que hemos clasificado como normales, son realmente normales, el 18 % restante son fraudulentas que hemos clasificado erróneamente. (vp/p)

El *f1-score* representa la relación entre la precisión y la recuperación ($2\frac{precision \cdot recuperacion}{precison+recupracion}$)

El soporte para una clase es la cantidad de ejemplos que hemos clasificado en esa clase, es decir que si tenemos un *support* de 2152 para la clase normal, significa que hemos clasificado 2152 ejemplos a esa clase.

Puesto que en nuestro caso estamos tratando de detectar carteras fraudulentas, nos conviene maximizar la recuperación de la clase *Fraudulent*, ya que un *recall* de 1, significaría que todas las carteras fraudulentas, han sido clasificadas como tal. No obstante, hay que encontrar un equilibrio con la precisión, porque una recuperación muy alta junto con una precisión baja, lo que indica es que estamos clasificando muchos ejemplos como fraudulentos, aunque muchos no lo sean.
Vistos los criterios de evaluación, podemos empezar a analizar los modelos.

3.2. Modelos Lineales

Los primeros modelos a evaluar serán los modelos lineales. Estos modelos son los que asumen que se puede predecir la variable objetivo usando relaciones lineales entres las variables. Los modelos que hemos elegido son:

- LDA: Linear discriminant Analisis
- Linear SVC: Linear Support Vector for Clasification
- KNN: K-nearest Neigbours

3.2.1. LDA: Linear Discriminant Analisis

El primer modelo que analizaremos será el Linear discriminant Analisis. Este modelo asume una distribución gausiana de los datos. No es nuestro cao, así que no esperamos grandes resultados de este

modelo. No obstante, este modelo es habitualmente usado para discriminar entre dos clases de datos, que es justamente nuestro objetivo, por eso lo hemos incluido en el informe Vamos a ver como lo hemos implementado:
Para empezar, vamos a realizar lo que es conocido como *validación cruzada*.

```
lda = LinearDiscriminantAnalysis()
print(np.mean(cross_val_score(lda,X_train_n,y_train,cv=10)))
```

```
0.7648997706193279
```

Este resultado tiene un rango de 0 a 1, donde 0 significa que el modelo no predice nada bien los datos, y 1 que los predice a la perfección. El resultado obtenido no es del todo malo, pese a no cumplir con los requisitos establecidos por el modelo. Hay que tener en cuenta que este es el resultado es con el conjunto de **train** y es meramente orientativo, el resultado a tener en cuenta es el que se obtiene de entrenar el modelo con el conjunto de *train* y validarlo con el conjunto de *test*

```
lda.fit(X_train_n, y_train)
print(classification_report(
        lda.predict(X_test_n),
        y_test,
        target_names=target_names
        )
    )
```

```
              precision    recall  f1-score   support

      Normal       0.77      0.82      0.79      2152
  Fraudulent       0.77      0.71      0.74      1857

    accuracy                           0.77      4009
   macro avg       0.77      0.77      0.77      4009
weighted avg       0.77      0.77      0.77      4009
```

Vemos por el resultado obtenido, que este modelo no es del todo malo. Podemos ver que es bastante equilibrado, y que el resultado del *accuracy* es parecido al obtenido en la validación cruzada. Obtener un resultado mucho peor indicaría que el modelo se está sobre ajustando al conjunto de *train*

Para poder entender un poco mejor la clasificación realizada por el modelo, usaremos dos de las medidas más habituales: La matriz de confusión y la curva R.O.C.

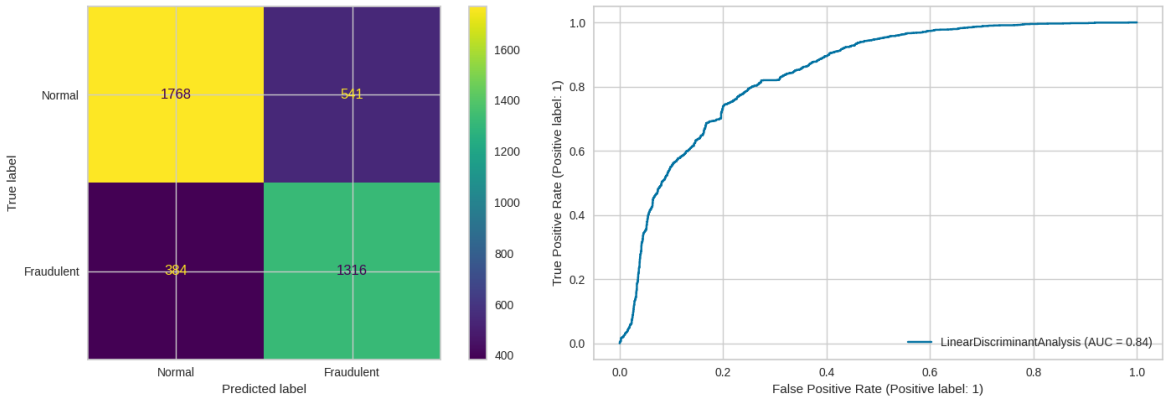

Figura 7: Matriz de confusión y Curva R.O.C del modelo LDA

La matriz de confusión es bastante simple de entender, representa la distribución de los ejemplos clasificados por el modelo en función de su verdadera clase y la clase predicha. La matriz de confusión óptima sería una que tuviera ejemplos solo en la diagonal. Vemos que en nuestro caso, estamos clasificando de forma errónea aproximadamente un 30 % de ejemplos para ambas clases.

La curva R.O.C representa que proporción de ejemplos de una clase acertamos, en función de cuantos clasificamos mal para esa clase. Las curvas R.O.C siempre empiezan en el punto (0.0, 0.0) y terminan en el punto (1.0, 1.0). El punto (0.0, 0.0) representa no meter a ningún ejemplo en esa clase, no obtienes ningún ejemplo bien clasificado, pero tampoco ninguno mal clasificado. El punto (1.0, 1.0) representa clasificar todos los ejemplos en esa clase, obtienes todos los ejemplos de la clase bien clasificados, y todos los de la otra clase mal clasificados.
En esta métrica lo que se busca es maximizar el área bajo la curva, es decir, que con pocos ejemplos mal clasificados (0.1, 0.2) obtengamos una gran proporción de ejemplos bien clasificados para la clase (0.8, 0.9), en el caso del LDA, el área es de 0.84 y para llegar al 0.8 de ejemplos bien clasificados, estaríamos metiendo cerca del 0.3 de la otra clase.

En esencia, este modelo asigna pesos a cada una de las variables y se vale de estos pesos a la hora de predecir la clase de cada uno de los ejemplos. Podemos ver cuáles han sido los pesos asignados por el modelo:

```
coefs = pd.DataFrame(lda.coef_)
coefs.columns = X.columns

plt.figure(figsize=(20,2));
sns.heatmap(
    coefs.abs().round(3),
    annot=True,
    linewidths=.5
    ,cbar=True,
    xticklabels=True,
    cmap='Blues',
    annot_kws={'size':12})
```

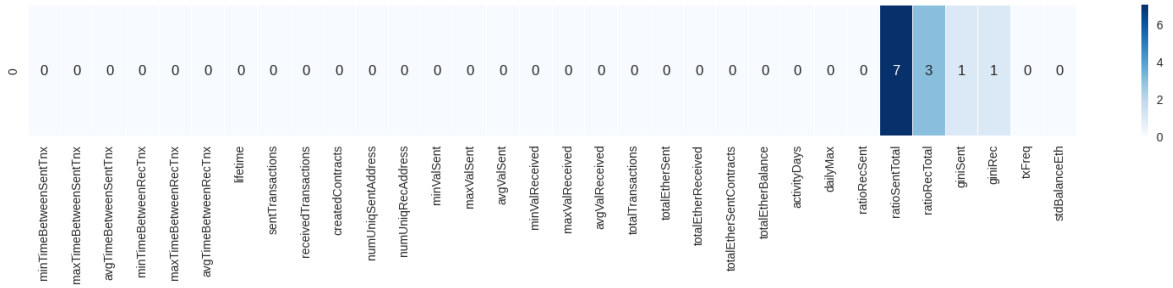

Figura 8: Pesos asignados por el modelo LDA

Vemos que el modelo prácticamente ignora todas las variables excepto el ratio entre transacciones enviadas entre las totales.

Por último, veremos que ha usado el modelo para predecir la clase de los modelos. Para eso, usaremos el explainer:

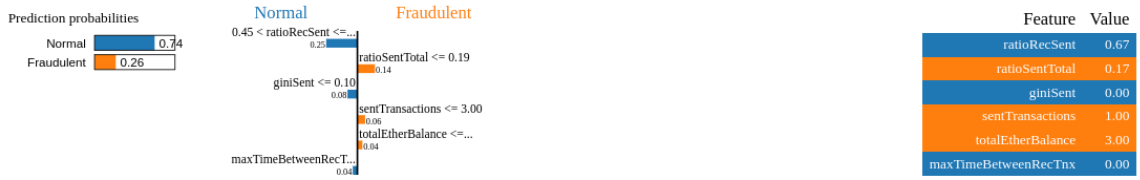

Figura 9: explainer del modelo LDA

Aquí podemos observar los criterios seguidos por el modelo para clasificar uno de los ejemplos, en concreto el primero de ellos. Vemos que lo ha clasificado como normal con una probabilidad del 74 %. También podemos ver cuáles son las variables que han tenido más peso para tomas su decisión.

Para poder comparar los modelos, usaremos siempre el mismo ejemplo para analizar.

3.2.2. Linear SVC: Linear Support Vector for Clasification

Los modelos basados en SVC, tienen como objetivo encontrar el hiperplano que separe mejor las clases de los datos.

Empezaremos con la validación cruzada:

```
lsvc = LinearSVC()
print(np.mean(cross_val_score(lsvc,X_train_n,y_train,cv=10)))
```

```
0.7814893726332841
```

Obtenemos un resultado similar al anterior usando solo la validación cruzada.
Este modelo, sin embargo, consta de unos parámetros que podemos ajustar para obtener un mejor resultado. Los parámetros que ajustaremos son los siguientes:

- C: tomaremos 100 valores entre 10^3 y 10^{-3}
- penalty: probaremos con la penalización cuadrática (L2) o la lineal (L1)
- loss: probaremos con los valores *hinge* y *squared hinge*

```
param = {'C':10**np.linspace(-3,3,101),
         'penalty':['l1','l2'],
         'loss':['hinge','squared_hinge']
        }
lsvc = LinearSVC(max_iter=25000)
lsvc_gs = GridSearchCV(lsvc,param,cv=cv, n_jobs=-1, refit=True)
lsvc_gs.fit(X_train_n, y_train);
show_html(
    pd.DataFrame(lsvc_gs.cv_results_).loc[:,
    ['params','mean_test_score','rank_test_score']
    ].sort_values(by='rank_test_score').head().to_html()
    )
```

	params	mean_test_score	rank_test_score
0	{'C': 758.5775750291835, 'loss': 'squared_hinge', 'penalty': 'l2'}	0.792	1
1	{'C': 501.18723362727246, 'loss': 'squared_hinge', 'penalty': 'l2'}	0.791	2
2	{'C': 436.51583224016565, 'loss': 'squared_hinge', 'penalty': 'l2'}	0.791	3
3	{'C': 575.4399373371566, 'loss': 'squared_hinge', 'penalty': 'l2'}	0.790	4
4	{'C': 380.1893963205613, 'loss': 'squared_hinge', 'penalty': 'l2'}	0.790	5

Aquí podemos ver, ordenados de mejor a menor, los mejores parámetros que hemos encontrado. Parece

evidente que la mejor función de perdida es la *squared hinge* y la mejor penalización es la L2. Vemos también que la C óptima que hemos encontrado es de 758 cerca de la máxima que habíamos puesto de 1000.

```
print(classification_report(
    lsvc_gs.predict(X_test_n),
    y_test,
    target_names=target_names
    )
)
```

	precision	recall	f1-score	support
Normal	0.79	0.84	0.82	2168
Fraudulent	0.80	0.74	0.77	1841
accuracy			0.80	4009
macro avg	0.80	0.79	0.79	4009
weighted avg	0.80	0.80	0.80	4009

Visto el análisis, podemos decir que este modelo es ligeramente mejor que el LDA visto anteriormente. No solo tiene mejor recuperación en la clase de carteras fraudulentas, sino que, en términos generales, es mejor en todos los aspectos.

Veamos ahora la matriz de confusión y la curva R.O.C que genera este modelo:

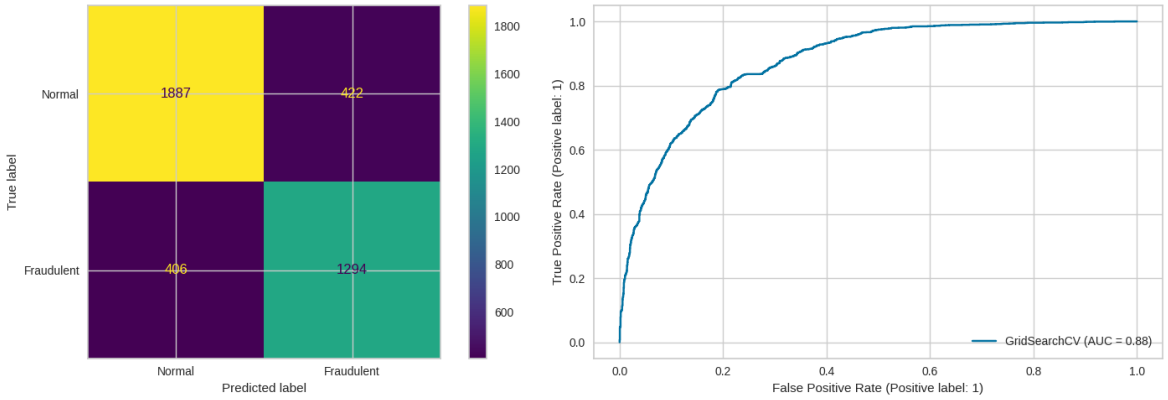

Figura 10: Matriz de confusión y Curva R.O.C del modelo lineal SVC

Viendo la matriz de confusión del modelo, vemos resultados muy parecidos al modelo anterior, el LDA, ambos tienen un número decente de aciertos, y aproximadamente confunden la misma cantidad de ejemplos para ambas clases.

La curva R.O.C, sin embargo, vemos que es ligeramente mejor. No solo encierra un área bajo la curva mayor, 0.88 a comparación con el 0.84 que obtenía el LDA, sino que vemos que con tan solo un 20 % de ejemplos mal clasificados, obtenemos un 80 % de los ejemplos fraudulentos bien clasificados.

Podemos ver ahora cuáles son las variables a los que ha dado más peso el modelo a la hora de hacer las predicciones.

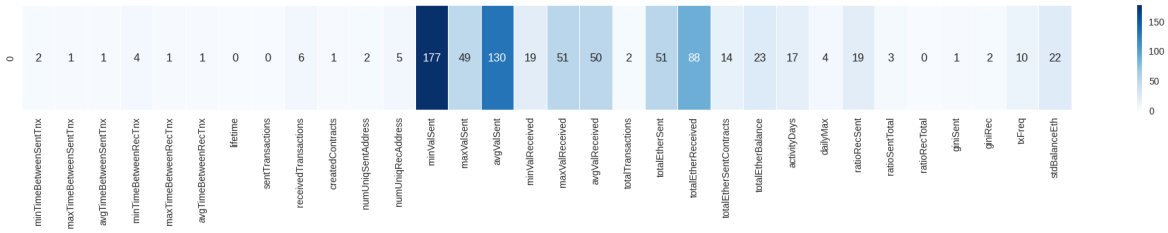

Figura 11: Pesos asignados por el modelo Linear SVC

Aquí es donde podemos ver la primera diferencia significativa con el modelo LDA, mientras que en el modelo LDA, los atributos considerados importantes eran los ratios de transacciones, en este, damos mucho más peso al los valores de las transacciones de envío, siendo el mínimo el más importante.

Como este modelo no es probabilístico, no podemos mostrar el explainer como hemos hecho con el anterior modelo, sin embargo, podemos usar el *CalibratedClasifier* para simular las probabilidades del modelo.

```
from sklearn.calibration import CalibratedClassifierCV
clf = CalibratedClassifierCV(lsvc_gs.best_estimator_)
clf.fit(X_train_n, y_train)
```

Ahora, con la ayuda del modelo auxiliar, podemos ver qué atributos han hecho decantar el modelo para una o la otra clase.

Figura 12: explainer del modelo linear SVC

Vemos que este modelo ha clasificado el ejemplo como fraudulento, con una probabilidad del 100 %, puesto que no es un modelo probabilístico. Vemos que prácticamente el único factor que ha hecho decidir el modelo por una clase o por otra ha sido el mínimo tiempo entre transacciones recibidas, a diferencia del modelo LDA, que le dio más peso al ratio entre transacciones recibidas y enviadas.

3.2.3. KNN: K-Nearest Neigbours

El siguiente y último modelo lineal que veremos es el modelo KNN. Este modelo se basa en la proyección de los ejemplos en un espacio multidimensional, y luego asignar la clase de cada ejemplo en función de la proximidad a otros ejemplos. Este modelo es basa en la premisa que ejemplos similares, es decir, cercanos, tienen altas probabilidades de compartir clase. El siguiente y último modelo lineal que veremos es el modelo KNN. Este modelo se basa en la proyección de los ejemplos en un espacio multidimensional, y luego asignar la clase de cada ejemplo en función de la proximidad a otros ejemplos. Este modelo es basa en la premisa que ejemplos similares, es decir, cercanos, tienen altas probabilidades de compartir clase.

Como en los modelos anteriores, vamos a usar validación cruzada para ver que tal lo hace:

```
knn =  KNeighborsClassifier()
print(np.mean(cross_val_score(knn, X_train_n, y_train, cv=10)))
```

```
0.8414639564086059
```

Este resultado, de momento es el mejor que hemos obtenido (y con diferencia) respecto a los anteriores modelos. Esto ya nos puede dar una idea de lo que nos vamos a encontrar cuando exploremos valores para los parámetros de los que disponemos.

- nNeighbors: representa el número de vecinos a considerar.
- weigth: hace referencia a si se le da más peso a los vecinos más cercanos (distance) o si no (uniform)
- leafSize: representa el número de hojas del algoritmo
- metric: representa la fórmula usada para medir la distancia entre vecinos

Empezamos con la exploración:

```
param = {'n_neighbors':[5, 10, 15, 20, 25, 30],
         'weights':['uniform', 'distance'],
         'leaf_size':[20, 30, 40, 50],
         'metric': ['l2', 'l1', 'cosine']}

knn_gs = GridSearchCV(knn,param,cv=cv, n_jobs=-1)
knn_gs.fit(X_train_n, y_train)
show_html(
    pd.DataFrame(knn_gs.cv_results_).loc[:,
    ['params','mean_test_score','rank_test_score']
    ].sort_values(by='rank_test_score').head().to_html()
    )
```

{'leaf_size': 30, 'metric': 'l1', 'n_neighbors': 10, 'weights': 'distance'}	0.868
{'leaf_size': 50, 'metric': 'l1', 'n_neighbors': 10, 'weights': 'distance'}	0.868
{'leaf_size': 40, 'metric': 'l1', 'n_neighbors': 10, 'weights': 'distance'}	0.868
{'leaf_size': 20, 'metric': 'l1', 'n_neighbors': 10, 'weights': 'distance'}	0.868
{'leaf_size': 30, 'metric': 'l1', 'n_neighbors': 20, 'weights': 'distance'}	0.867

Una vez hecha la exploración de parámetros, los resultados son incluso mejores que con los parámetros por defecto. Así a simple vista, nos damos cuenta de que el tamaño de hoja es totalmente irrelevante para el problema, ya que con todos los valores obtenemos el mismo resultado. También vemos que el tamaño óptimo de vecinos es de 10 junto con la distancia L1.

Podemos ver el reporte de clasificación para entender mejor los resultados.

```
print(classification_report(
    lsvc_gs.predict(X_test_n),
    y_test,
    target_names=target_names
    )
)
```

	precision	recall	f1-score	support
Normal	0.91	0.89	0.90	2371
Fraudulent	0.84	0.87	0.86	1638
accuracy			0.88	4009
macro avg	0.88	0.88	0.88	4009
weighted avg	0.88	0.88	0.88	4009

Vemos que, en general, todos los campos son bastante mejores que los obtenidos tanto con el modelo LDA, como con el modelo de Lineal SVC. Podemos destacar que estamos cerca de obtener un 0.90 de recuperación de la clase de carteras fraudulentas, eso significaría que el 90 % de las carteras fraudulentas están siendo clasificadas como tal.

Como hemos hecho anteriormente, vamos a ver la matriz de confusión y la curva R.O.C generadas por el modelo.

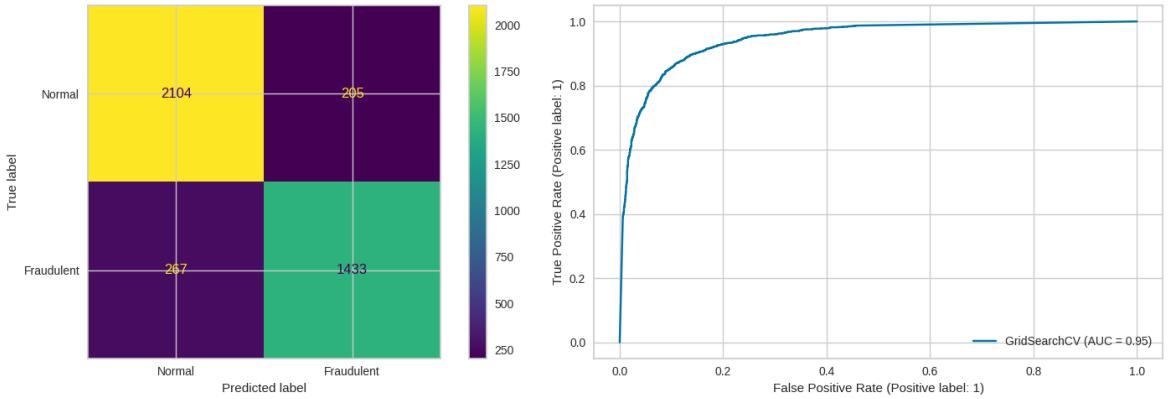

Figura 13: Matriz de confusión y curva R.O.C del modelo KNN

En la matriz de confusión, podemos ver que el número de ejemplos mal clasificados ha pasado de ser aproximadamente 400 en cada clase, a ser de solo 200 o 250 en el peor de los casos.

La curva R.O.C es mucho más interesante, ya que podemos ver que no solo el área bajo la curva ha aumentado de forma significativa, de un 0.84 o 0.88 a un 0.95, sino que para obtener el 80 % de los ejemplos de la clase fraudulenta bien clasificados, solo meteríamos un 10 % (o menos) de ejemplos de la otra clase. Metiendo un 20 % de carteras normales mal clasificadas, estaríamos detectando el casi 95 % de las carteras fraudulentas.

Por la naturaleza del modelo, no podemos ver qué pesos le ha asignado a cada variable porque simplemente no le asigna pesos a las variables, se basa simplemente en la cercanía de los ejemplos en el espacio.

Pasamos directamente a ver el explainer para este modelo.

Figura 14: explainer del modelo KNN

Vemos que este modelo clasifica este ejemplo como Normal igual que ha hecho el primer modelo, el LDA, esta vez con una probabilidad del 100 %, es decir, el modelo no tiene ninguna duda. Esto es porque el KNN, al igual que el linear SVC no es un modelo probabilístico.

3.2.4. Resultados

Una vez vistos todos los modelos lineales, vamos a echar un vistazo a sus resultados:

```
results_df_lineal.sort_values(by=['test_acc'], ascending=False)
```

	test acc	precision score (Fraudulent)	recall score (Fraudulent)	f1 score (Fraudulent)
KNN	0.882	0.875	0.843	0.859
LSVC	0.793	0.756	0.754	0.755
LDA	0.704	0.625	0.756	0.684

Figura 15: resultados de los tres modelos lineales

En esta tabla, podemos ver los resultados de los tres modelos lineales que hemos estudiado, ordenados por su evaluación en el conjunto de test, también tenemos resultados para la puntuación sobre la precisión, la recuperación y el *f1-scrore* sobre la clase de carteras fraudulentas, que es en la que nos interesa centrarnos.

Podemos ver claramente que el mejor modelo es, con diferencia, el KNN, obteniendo casi más de un 10 % de precisión que el segundo modelo, el Lineal SVC. Vemos claramente que el KNN es el modelo que obtiene mejores resultados en todas las categorías, es decir, es sin duda el mejor de los tres modelos.

Podemos destacar que, tanto el modelo LDA como el lineal SVC, obtienen la misma puntuación en la recuperación de ejemplos de la clase de carteras fraudulentas, esto puede ser interesante, ya que el modelo LDA es considerablemente más fácil de entrenar que el lineal SVC así que para una tarea donde únicamente nos interese maximizar la recuperación, podría ser interesante usarlo en lugar del lineal SVC.

3.3. Modelos no Lineales

Una vez vistos los modelos Lineales y su desempeño, pasamos a ver los modelos no lineales que hemos elegido. Por lo general, estos modelos son más complejos, es decir, son más costosos de entrenar, pero deberían obtener un mejor resultado. Los modelos que veremos serán los siguientes:

- SVC: Support Vector Machine
- MLP: Multi-Layer Perceptron
- RF: Random Forest

3.3.1. SVC: Suppoer Vector for Clasification

El primer modelo no lineal que veremos es el SVC. A diferencia del que hemos visto anteriormente, esta es una versión no lineal del mismo modelo. Como hemos comentado anteriormente, los modelos no lineales son más complejos, y, por lo tanto, más costosos de entrenar. Es por esto que esperamos un mejor rendimiento que su hermano pequeño, la versión lineal.
Para empezar, realizaremos una prueba de validación cruzada:

```
svc = SVC()
print(np.mean(cross_val_score(svc,X_train_n,y_train,cv=10)))
```

```
0.8490821750969353
```

Solamente con la validación cruzada, ya podemos observar un claro aumento de rendimiento respecto a la versión lineal del modelo que hemos visto anteriormente. Hemos pasado de un 0.78 a casi un 0.85 en la validación cruzada, una mejora sustancial sin duda.

Podemos ahora explorar los parámetros de los que disponemos, a ver si conseguimos aún un mejor rendimiento. En este caso, solo exploraremos el parámetro de regularización C, ya que no disponemos de los que hemos visto anteriormente.

```
param = {'C':10**np.linspace(-4,4,101)}

svc = SVC(max_iter=250000, random_state=0, probability=True)
svc_gs = GridSearchCV(svc,param, cv=cv, n_jobs=-1, refit=True)
svc_gs.fit(X_train_n, y_train)
show_html(
    pd.DataFrame(svc_gs.cv_results_).loc[:,
    ['params','mean_test_score','rank_test_score']
    ].sort_values(by='rank_test_score').head().to_html()
    )
```

	params	mean_test_score	rank_test_score
99	{'C': 8317.63771102671}	0.926	
98	{'C': 6918.309709189362}	0.926	
97	{'C': 5754.399373371566}	0.925	
93	{'C': 2754.228703338169}	0.925	
100	{'C': 10000.0}	0.925	

Lo primero de lo que nos damos cuenta es de que el resultado obtenido es muy bueno. Mucho mejor de lo que habíamos obtenido con la versión lineal del modelo, vemos que con los mejores parámetros, llegamos hasta el 0.92 mientras que con la versión lineal no llegábamos siquiera al 0.8.

Veamos ahora el informe de clasificación a ver qué resultados obtenemos con el conjunto de test.

```
print(classification_report(
    svc_gs.predict(X_test_n),
    y_test,
    target_names=target_names
    )
)
```

```
                precision    recall  f1-score   support

        Normal       0.93      0.95      0.94      2269
    Fraudulent       0.93      0.91      0.92      1740

      accuracy                           0.93      4009
     macro avg       0.93      0.93      0.93      4009
  weighted avg       0.93      0.93      0.93      4009
```

Vemos que en el conjunto de test el resultado sigue siento bueno. Obtenemos más de un 0.9 tanto en la precisión como en la recuperación de ambas clases. A falta de ver aún los otros dos modelos no lineales, este es un muy buen candidato a ser el modelo definitivo.

Vamos a ver la matriz de confusión y la curva R.O.C del modelo.

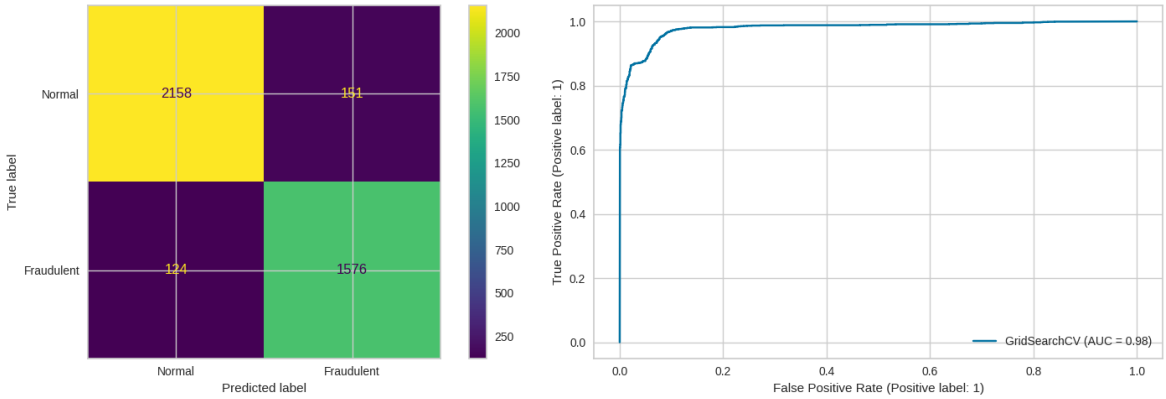

Figura 16: Matriz de confusión y curva R.O.C del modelo SVC

Podemos ver, en la matriz de confusión que la mayoría de ejemplos se encuentran en la diagonal, es decir, bien clasificados, tan solo son unos pocos, 150 y 120, que aún clasificamos de forma errónea.

La curva R.O.C ya empieza a tener una forma mucho más favorable, vemos que el área que encierra bajo ella es cercana al 1, de 0.98, siento 1 el máximo posible. No solo el valor del área es bueno, sino que podemos ver que para clasificar el 90 % de los ejemplos fraudulentos como tal, solo nos estaríamos equivocando con cerca del 1 % o 2 % de los ejemplos normales, unos resultados muy buenos.

Veamos ahora el explainer del modelo con el mismo ejemplo que analizábamos con los modelos lineales.

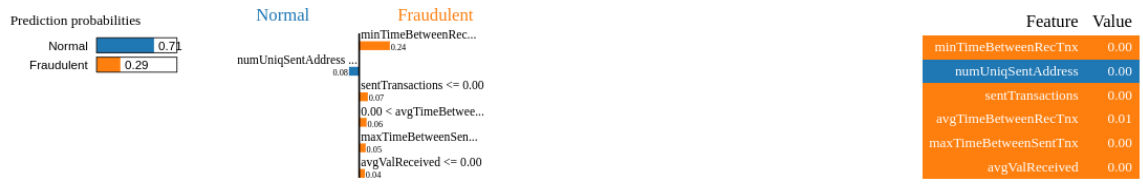

Figura 17: explainer del modelo SVC

Vemos que el modelo ha clasificado el ejemplo como Normal, con una probabilidad del 71 %. Nos resulta curioso, ya que cinco de las seis características más importantes, indican que es fraudulento. Suponemos que las demás indican mayoritariamente que es Normal y eso le ha dado más peso.

3.3.2. MLP: Multilayer-Percepton classifier

Para el segundo modelo no lineal que veremos seguiremos con la misma estructura, en este caso el MLP. Como los anteriores es un modelo que permite la clasificación binaria que nos servirá para nuestros datos, no obstante también permite regresión y clasificación de multi-clases. Para iniciar el entrenamiento del modelo veremos el resultado de la validación cruzada con los datos de entrenamiento:

```
mlp = MLPClassifier(max_iter=1000, early_stopping=True, random_state=10)
print(np.mean(cross_val_score(mlp,X_train_n,y_train,cv=10)))
```

```
0.8474827057852888
```

En este caso es ligeramente peor que el modelo SVC pero no de manera significativa y evidentemente mejor que los modelos lineales. Como valor introductorio nos sirve para pensar que se comportara de manera similar al primer modelo no lineal, al menos con la precisión de las predicciones.

Seguidamente, veremos como podemos mejorar los resultados con una profundización de los parámetros de la red neuronal, en este caso tenemos:

```
param = {'hidden_layer_sizes':[1, 25, 50, 75, 100, 125, 150],
         'activation':['logistic', 'relu', 'tanh', 'identity'],
         'learning_rate_init': [0.0001, 0.001, 0.01, 0.1]  }
```

El parámetro *hiddenlayersizes* nos permite determinar el número de capas intermedias de nuestra red, por otra parte, el parámetro *activation* nos permite decidir la función de activación que recibirán los resultados de cada neurona, por último, tenemos el *learningrateinit* que nos da información del primer valor del *learningrate* al empezar a entrenar. De esta manera podemos ver como se comporta la red con los diferentes parámetros:

```
mlp =  MLPClassifier(max_iter=1000, early_stopping=True
    , n_iter_no_change=10,random_state=10)
mlp_gs =  GridSearchCV(mlp,param,cv=cv, n_jobs=-1, refit=True)
mlp_gs.fit(X_train_n, y_train);
show_html(pd.DataFrame(mlp_gs.cv_results_)
    .loc[:,['params', 'mean_test_score','rank_test_score']]
    .sort_values(by='rank_test_score').head().to_html())
```

{'activation': 'relu', 'hidden_layer_sizes': 125, 'learning_rate_init': 0.01}	0.902
{'activation': 'relu', 'hidden_layer_sizes': 100, 'learning_rate_init': 0.01}	0.900
{'activation': 'relu', 'hidden_layer_sizes': 150, 'learning_rate_init': 0.01}	0.899
{'activation': 'relu', 'hidden_layer_sizes': 75, 'learning_rate_init': 0.01}	0.894
{'activation': 'relu', 'hidden_layer_sizes': 50, 'learning_rate_init': 0.01}	0.893

Vemos que los resultados sobre el conjunto de entrenamiento han mejorado un poco, respecto a la validación cruzada que hemos hecho al principio. Podemos ver que los parámetros elegidos son la función de activación *relu* y el *learningrate* de 0.01, elegidos por los 5 mejores modelos. Vemos también que el número óptimo de capas parece ser 125.

Con estos parámetros, vamos a ver el informe de clasificación:

```
print(classification_report(
        mlp_gs.predict(X_test_n),
        y_test,
        target_names=target_names
        )
    )
```

	precision	recall	f1-score	support
Normal	0.94	0.91	0.92	2379
Fraudulent	0.88	0.91	0.89	1630
accuracy			0.91	4009
macro avg	0.91	0.91	0.91	4009
weighted avg	0.91	0.91	0.91	4009

Vemos que el resultado es bastante parecido al obtenido con el modelo SVC; sin embargo, podemos ver que es ligeramente peor en la mayoría de campos, solo lo mejora en la precisión de la clase Normal, cosa que no es muy relevante para nuestro problema

Vamos a ver tanto la matriz de confusión como la curva R.O.C de este modelo.

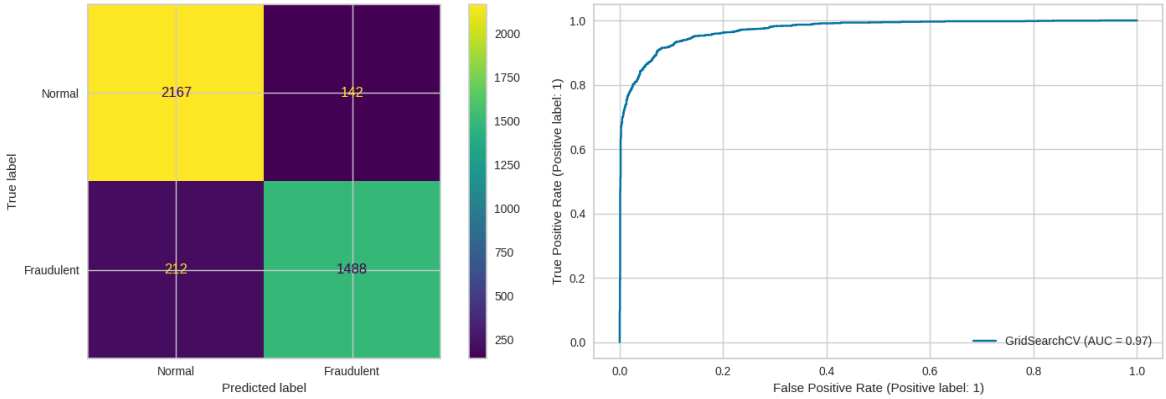

Figura 18: Matriz de confusión y curva R.O.C del modelo MLP

Tanto como la matriz de confusión como la curva R.O.C, nos indican que el resultado es bueno. Seguimos viendo que le es más difícil clasificar los ejemplos fraudulentos que los normales, aun así los ejemplos mal clasificados no son muchos. Respecto a la curva R.O.C, vemos que el área bajo la curva es de 0.97, bastante buena. Podemos destacar que esta curva es bastante más constante que la vista en el modelo SVC, que tenía una especie de zona en donde, añadir ejemplos mal clasificados, no ayudaba a clasificar más correctamente. Eso nos puede indicar que este modelo puede ser preferible al SVC en algunos casos.

Para terminar, veamos el explainer de este modelo:

Figura 19: explainer del modelo MLP

Vemos que este modelo ha clasificado el ejemplo como Normal, igual que la mayoría de modelos, con una probabilidad del 75 %. Este modelo le está dando mucho peso al número de transacciones recibidas de direcciones únicas

3.3.3. Random Forest

Para terminar, el último modelo que veremos es el RandomForest. Este modelo está basado en el concepto de árboles de decisión. Este modelo es basa en la implementación de varios árboles de decisión. Un árbol de decisión se vale de decisiones, valga la redundancia, para clasificar los ejemplos, podríamos considerarlo, para simplificar, como una cadena de $if - else$.

Como en los demás ejemplos, empezamos con la validación cruzada:

```
rf = RandomForestClassifier(random_state=0)
print(np.mean(cross_val_score(rf,X_train_n,y_train,cv=10)))
```

```
0.9370745955616805
```

Simplemente con la validación cruzada, sin siquiera optimizar los parámetros, obtenemos el mejor resultado hasta el momento. Supera ligeramente el resultado obtenido con el modelo SVC, incluso después de explorar sus parámetros.

Vamos a ver si podemos mejorar un poco este resultado probando con distintos parámetros. Los parametros que exploraremos son los siguientes:

- *nestimators*: el número de árboles que usara el modelo
- *minsamplessplit*: el número mínimo de ejemplos para dividir un nodo
- *minsamplesleaf*: el número mínimo de ejemplos para que el nodo sea un nodo hoja
- *maxfeatures*: El número máximo de características a considerar

```
param = {'n_estimators':[10,50, 75, 100, 150, 200],
       'min_samples_split': [1, 2,4,32, 64],
       'min_samples_leaf': [1, 2,4,32],
       'max_features': [1,2,32, 64, 128]
       }

rf=RandomForestClassifier(warm_start=True, random_state=0)
rf_bs = BayesSearchCV(rf,param,n_iter=15, cv=cv, n_jobs=-1, refit=True, random_stat
rf_bs.fit(X_train_n, y_train);
show_html(
     pd.DataFrame(rf_bs.cv_results_).loc[:,
     ['params','mean_test_score','rank_test_score']
     ].sort_values(by='rank_test_score').head().to_html()
     )
```

	params	mean_test_score	rank_test_score
13	{'max_features': 128, 'min_samples_leaf': 1, 'min_samples_split': 4, 'n_estimators': 150}	0.940	1
14	{'max_features': 64, 'min_samples_leaf': 2, 'min_samples_split': 4, 'n_estimators': 150}	0.939	2
2	{'max_features': 32, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}	0.938	3
10	{'max_features': 64, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}	0.938	3
0	{'max_features': 32, 'min_samples_leaf': 4, 'min_samples_split': 4, 'n_estimators': 150}	0.937	5

Figura 20: Optimización de parametros del modelo RandomForest

Vemos que, explorando algunos de los parámetros, el resultado mejora ligeramente. Veamos el informe de clasificación.

```
print(classification_report(
     rf_bs.predict(X_test_n),
     y_test,
     target_names=target_names
     )
)
```

```
              precision    recall  f1-score   support

      Normal       0.96      0.93      0.94      2377
  Fraudulent       0.90      0.94      0.92      1632
```

```
              precision    recall  f1-score   support

    accuracy                         0.94      4009
   macro avg       0.93      0.94      0.93      4009
weighted avg       0.94      0.94      0.94      4009
```

Como era de esperar, en el conjunto de test, también obtenemos un excelente resultado, el mejor de los que hemos visto. Obtenemos hasta un 0.94 de recuperación en la clase Fraudulent.

Veamos la curva R.O.C y la matriz de confusión para entender un poco mas el modelo:

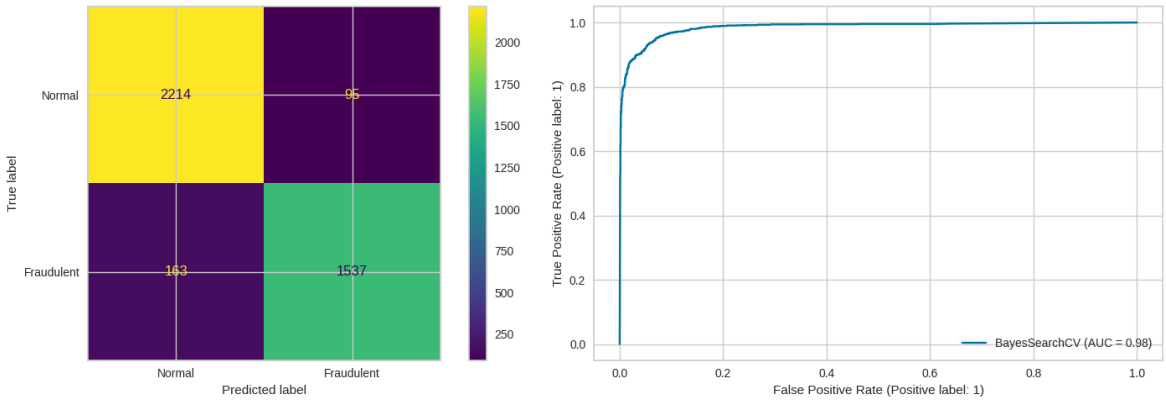

Figura 21: Matriz de confusión y curva R.O.C del modelo RF

Vemos que, como de costumbre, al modelo le cuesta más predecir la clase de carteras fraudulentas, ya que el número de carteras normales mal clasificadas, ha disminuido mucho, en cambio, el de carteras fraudulentas clasificadas como normales se mantiene más o menos constante.

La curva R.O.C nos muestra un resultado parecido al del anterior modelo, el MLP. Contamos con un área del 0.98 igual que el anterior, y más o menos de las mismas características.

Veamos el explainer del modelo:

Figura 22: Explainer del modelo RF

3.3.4. Resultados

Una vez vistos todos los modelos no lineales, vamos a echar un vistazo a sus resultados:

```
results_df_no_lineal.sort_values(by=['test_acc'], ascending=False)
```

	test acc	precision score (Fraudulent)	recall score (Fraudulent)	f1 score (Fraudulent)
RF	0.935645	0.941789	0.904118	0.922569
SVC	0.932652	0.910920	0.932353	0.921512
MPL	0.911699	0.912883	0.875294	0.893694

Figura 23: resultados de los tres modelos no lineales

Vemos que los tres modelos han obtenido resultados muy buenos, los tres están por encima del 0.9 de puntuación sobre el conjunto de test.

Podemos ver que, a priori, los dos mejores modelos son el RandomForest y el SVC, ya que el MPL obtiene una puntuación ligeramente inferior. Si miramos los demás indicadores, podemos ver que el RandomForest obtiene mejor puntación en precisión sobre la clase fraudulenta, mientras que el SVC obtiene mejor recuperación sobre la misma clase.

4. Conclusiones

Vistos ya todos los modelos. Veamos a ver sus resultados y a extraer conclusiones:

```
pd.concat([results_df_lineal, results_df_no_lineal], axis=0)
  .sort_values(by=['test_acc'], ascending=False)
```

	test acc	precision score (Fraudulent)	recall score (Fraudulent)	f1 score (Fraudulent)
RF	0.935645	0.941789	0.904118	0.922569
SVC	0.932652	0.910920	0.932353	0.921512
MPL	0.911699	0.912883	0.875294	0.893694
KNN	0.882265	0.874847	0.842941	0.858598
LSVC	0.793465	0.768804	0.733529	0.750753
LDA	0.704166	0.624879	0.756471	0.684407

Figura 24: resultados de todos los modelos

Podemos ver claramente que los tres modelos no lineales han obtenido resultados significativamente mejores que los modelos lineales, que han quedado relegados al final de la tabla.

Como hemos comentado anteriormente, el modelo que ha obtenido la mejor puntuación, el Random Forest, no es necesariamente el modelo a elegir en todos los casos. Vemos que el SVC, el segundo mejor modelo, obtiene una puntuación superior en recuperación que el Random Forest. Para esta tarea, nos centrábamos en detectar las carteras fraudulentas, y no tanto en la precisión de las detecciones, es por esto, que el modelo que consideramos como el mejor, es el SVC.

Este modelo no solo obtiene una puntuación muy buena (0.93) sino que obtiene la mejor recuperación de todos, llegando a detectar el 93 % de las carteras fraudulentas del conjunto, con un 91 % de precisión.

También tenemos que mencionar el modelo Random Forest, pues es el modelo que obtiene una precisión más alta en el conjunto fraudulento, llegando al 94 %, en un trabajo donde diésemos prioridad a la veracidad de las afirmaciones de una cartera fraudulenta, sin duda el mejor modelo para ello sería el RandomForest (de los que hemos visto).

Una prueba interesante que no hemos podido incluir en este trabajo es usar el parámetro $ClassWeigth$ con el que cuentan la algunos de los modelos que hemos entrenado. Este parámetro nos permite dar prioridad a una de las clases sobre las demás, esto nos hubiera permitido especificarle al modelo que la clase importante es la clase de carteras fraudulentas y, probablemente, enfocar su rendimiento en predecir mejor esa clase.

Tanto el SVC como el Random Forest cuentan con esta opción, así que sería interesante ver cuál de los dos modelos se ajusta mejor a la clase fraudulenta.