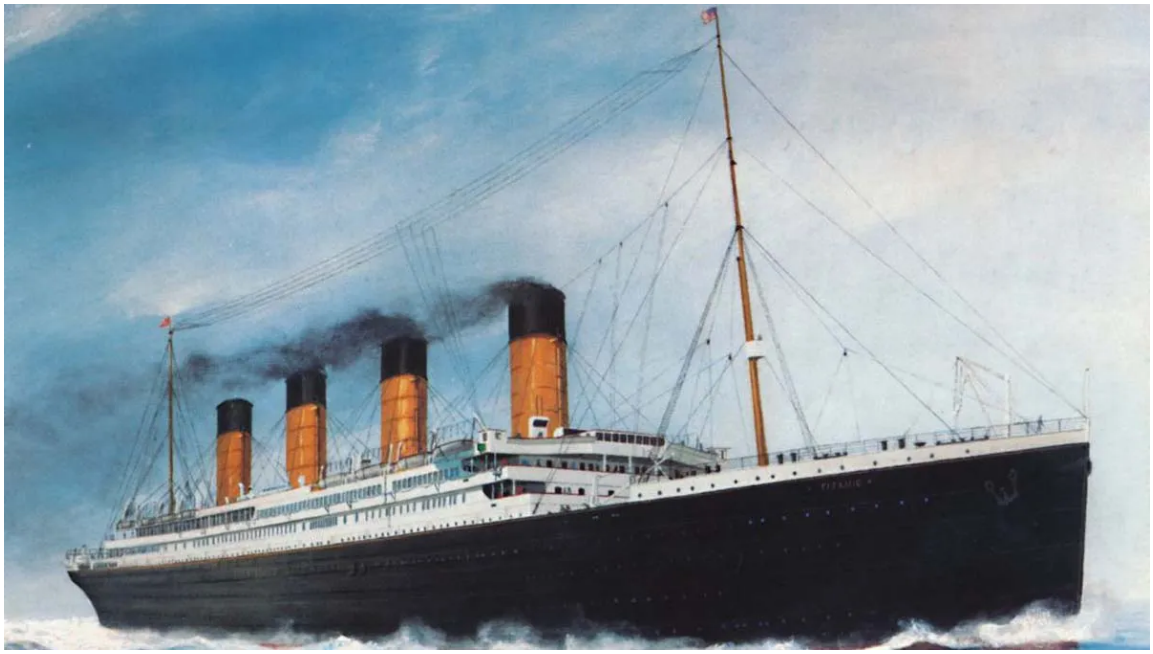
Predicción de supervivéncia del Titanic

Esta práctica se divide en dos partes:

1. En la primera parte entrenaremos un conjunto de modelos de clasificación de machine learning (perceptron, regression logística y arbol de decisión) para predecir la probabilidad que tendría un pasajero del Titánic de sobrevivir.
2. En la segunda parte realizaremos el *feature importance*, que básicamente consiste en analizar la importancia que tiene cada característica en cada uno de los modelos predictivos



Parte 1

1.1 Importar librerias

Importamos la librerias que usaremos

In [96]:
```
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

```
import numpy as np
import math
import statistics
```

1.2 Cargamos el dataset

Cargamos el dataset para poder visualizar los datos que usaremos.

In [37]:
```
df = pd.read_csv("./dades.csv")
```

Para poder ver todas las columnas usamos la sigüiente instrucción de *pandas*

In [38]:
```
pd.set_option('display.max_columns',None)
df.head()
```

Out[38]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	C
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	

1.3 Limpieza de datos

Para poder entrenar a nuestros modelos previamente debemos "*masticar*" los datos

1.3.1 Eliminar columnas (Ruido)

Lo primero que haremos será eliminar las columnas que contienen variables que no afectarán a nuestra predicción como el Id, el nombre del pasajero o el identificador del ticket

In [39]:
```
```

```
df = df.drop('PassengerId', axis=1)
df = df.drop('Name', axis=1)
df = df.drop('Ticket', axis=1)
```

In [5]:
```
df.head()
```

Out[5]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Cabin	Embarked
0	0	3	male	22.0	1	0	7.2500	NaN	S
1	1	1	female	38.0	1	0	71.2833	C85	C
2	1	3	female	26.0	0	0	7.9250	NaN	S
3	1	1	female	35.0	1	0	53.1000	C123	S
4	0	3	male	35.0	0	0	8.0500	NaN	S

1.3.2 Gestion de valores Nan

Ahora vamos a mostrar la cantidad de valores que hay en cada columna

In [52]:
```
df.count()
```

Out[52]:
```
Survived    891
Pclass      891
Sex         891
Age         714
SibSp       891
Parch       891
Fare        891
Cabin       204
Embarked    889
dtype: int64
```

El número de filas es 891 y las variables que no alcanzan ese valor nos muestran la cantidad de valores nulos que contienen

La característica 'Cabin', que hace referencia al identificador de la cabina donde se hospedaba el cliente, contiene una gran cantidad de valores Nan

In [53]:
```
(df['Cabin'].isna().sum()/891)*100 #Porcentaje de valores Nan
```

Out[53]:
```
77.10437710437711
```

Estos valores Nan representan que el pasajero no tenia cabina. No podemos especular sobre si esto afecta o no a la predicció de su supervivencia, por tanto, también podriamos eliminar la columna. Primero dividiremos la columna en un valor binario para identificar si tenía columna o no. Más adelante evaluaremos si esta era la mejor opción cuando entrenoms el modelo.

In [40]:
```
def set_bin(cabin:str):
    if cabin != cabin:
        return 0
    else:
        return 1

df['Cabin'] = [set_bin(x) for x in df['Cabin']]
```

Otra cantidad un poco más pequeña de falta de valores se encuentra en la columna '*Age*', la cual podemos completar con la media de todos los pasajeros

In [41]:
```
df['Age'].fillna(df['Age'].median(), inplace = True)
```

Por último nos faltaria gestionar la columna '*Embarked*'. Los valores nulos de esta característica también los podriamos completar con la media si no fuera porque no son un valor numérico. Debido a esto, usaremos otro estadístico que es la moda. Este estadísctico reflejará el valor que mas se repite dentro de la muestra

In [42]:
```
df['Embarked'].fillna(df['Embarked'].mode(), inplace = True)
```

1.3.3 Valores categóricos

Dentro de este dataset econtramos características como el sexo que son incompatibles con el aprendizaje de los modelos predictivos. Para ello debemos conseguir que todas la columnas de nuestro conjunto de datos contengan valores numéricos. Para solucionar esto usaremos la técnica de **One hot encoding** que consiste en ampliar el número de columnas según los posibles valores y rellenarlas con un valor binario.

Original Data

Team	Points
A	25
A	12
B	15
B	14
B	19
B	23
C	25
C	29

One-Hot Encoded Data

Team_A	Team_B	Team_C	Points
1	0	0	25
1	0	0	12
0	1	0	15
0	1	0	14
0	1	0	19
0	1	0	23
0	0	1	25
0	0	1	29

Realizaremos esto con las columnas '*Sex*', '*Embarked*' y '*Pclass*'. En el caso del sexo mantendremos únicamente una columna para no tener información redundante.

In [43]:
```
#Para el sexo
clb = df.pop("Sex")
ohe_clb = pd.get_dummies(clb, prefix='sexo')
df = pd.concat([df.reset_index(drop=True),
ohe_clb.reset_index(drop=True)], axis=1, sort=False)
df = df.drop('sexo_female', axis=1)
```

In [44]:
```
#Para el Embarque
clb = df.pop("Embarked")
ohe_clb = pd.get_dummies(clb, prefix='embarque')
df = pd.concat([df.reset_index(drop=True),
ohe_clb.reset_index(drop=True)], axis=1, sort=False)
```

In [45]:
```
#Para el Embarque
clb = df.pop("Pclass")
ohe_clb = pd.get_dummies(clb, prefix='Clase')
df = pd.concat([df.reset_index(drop=True),
ohe_clb.reset_index(drop=True)], axis=1, sort=False)

df.head()
```

Out[45]:

	Survived	Age	SibSp	Parch	Fare	Cabin	sexo_male	embarque_C	embarque_Q	emba
0	0	22.0	1	0	7.2500	0	1	0	0	
1	1	38.0	1	0	71.2833	1	0	1	0	
2	1	26.0	0	0	7.9250	0	0	0	0	
3	1	35.0	1	0	53.1000	1	0	0	0	
4	0	35.0	0	0	8.0500	0	1	0	0	

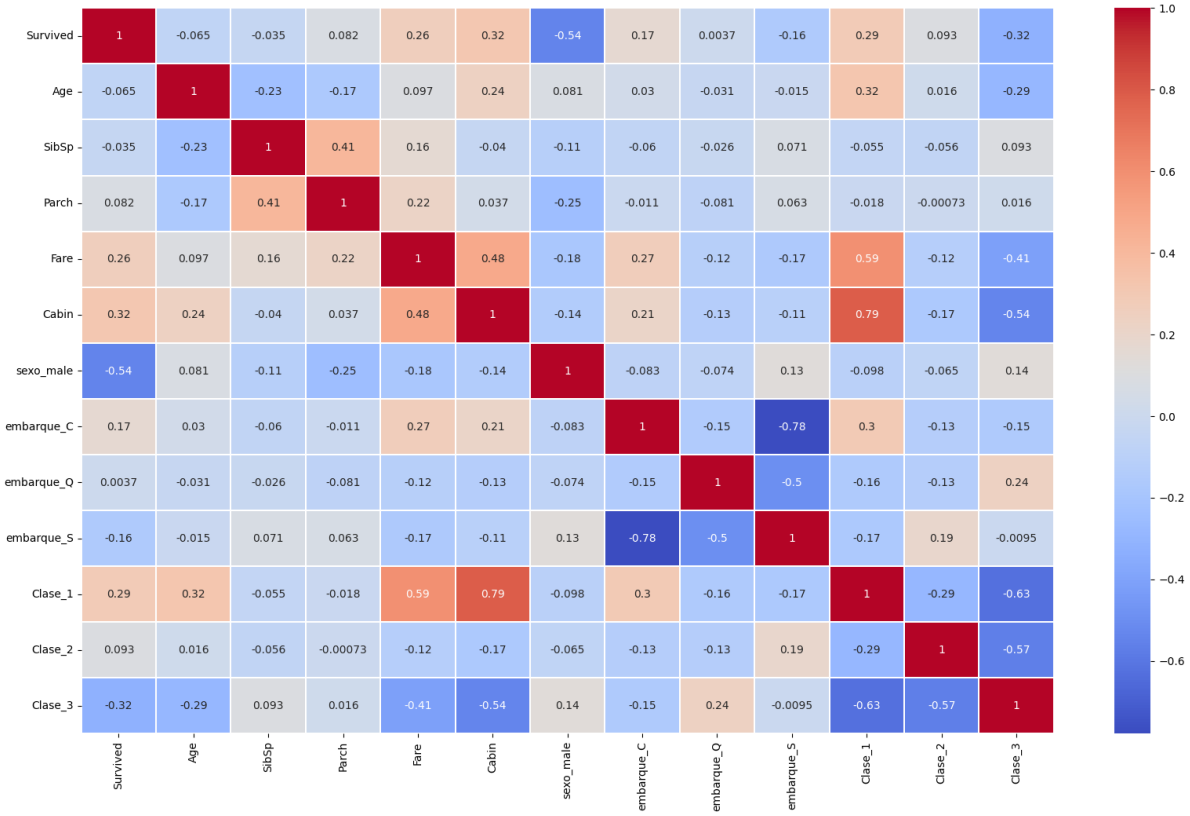
Ahora debemos pasar todos los valores de las columnas a el mismo tipo de dato. En este caso haremos que todos sean floats de 64 bits para mantener los decimales en '*Age*' y '*Fare*'.

In [48]:
```
for column in df:
    df[column]=df[column].astype(float)
```

1.3.4 Analisis Exploratorio de los Datos (EDA)

Ahora mostramos la matriz de correlación para intentar entcontrar valores dependientes o redundantes

In [47]:
```
sns.heatmap(df.corr(),annot=True,cmap='coolwarm',linewidths=0.2)
fig=plt.gcf()
fig.set_size_inches(20,12)
plt.show()
```



Al analizar la matriz podemos observar que hay una serie de valores relacionados con la columna '*Survived*' que destacan sobre otros, es decir, que estan mas cerca de 1 o -1. Para analizar estos valores hay que saber que si una casilla toma un valor positivo entonces los valores de las dos columnas relacionadas incrementan cuando el otro incrementa. En cambio si es negativo, cuando uno sube el otro baja.

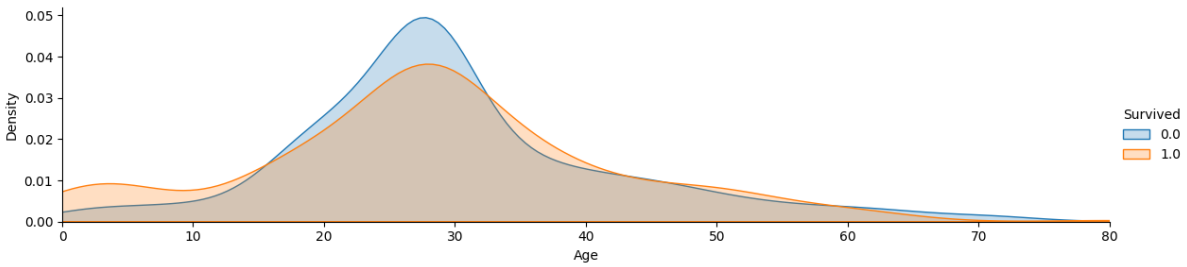
Por ejemplo, el valor mas representativo es el que relaciona '*sexo_male*' con 'survived'. Después de analizar estos dos primeros ejemplos podemos llegar a una conclusión de que el sexo femenino tiene más posibilidades de sobrevivir que el masculino, ya que en la matriz de correlación tiene un valor positivo elevado.

Otra columna interesante es la de '*Pclass*' con un valor de -0.34 lo que nos indica que cuanto mayor sea la clase menos probabilidades de sobrevivir. En el Titanic había tres clases (1,2 y 3) por lo que los de clase 1 tenían más posibilidades de sobrevivir que los de clase 2 o 3.

Por último hay que destacar la columna '*Fare*' que tiene un valor de 0.26 (cuanto más vale fare, más vale survived). En nuestro dataset el valor '*Fare*' indica la cantidad que han

pagado las personas por el billete, por lo que los que más han pagado tiene más posibilades de sobrevivir que los que han pagado una cantidad menor.

In [49]:
```
facet = sns.FacetGrid(df, hue="Survived",aspect=4)
facet.map(sns.kdeplot,'Age',fill=True)
facet.set(xlim=(0, df['Age'].max()))
facet.add_legend()
plt.show()
```

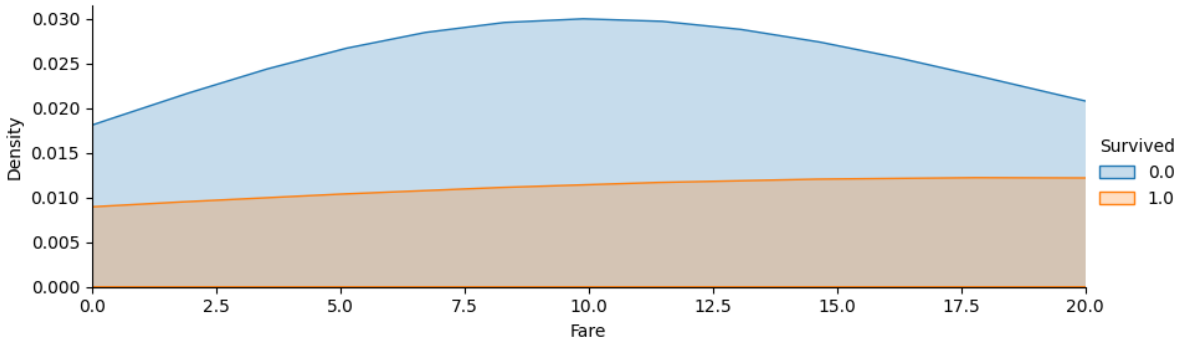


Como podemos observar en el gráfico anterior entre el rango de edad de 18-40 años nos encontramos con más cantidad de personas que en los otros rangos, por lo que es normal que muera y sobrevive más gente, aunque se puede ver como en este rango de edad hay más densidad de personas que no sobreviven a que sobreviven. Además, en el rango de edad de 0 a 18 años nos encontramos con que hay más supervivientes, y que a partir de los 45 años se igualan los supervivientos con los no supervivientes superando estos últimos a los primeros una vez superados los 60.

Con estos podemos sacar la conclusión que la edad era un factor importante si eras menor de edad ya que los niños eran prioritarios para salvarse, pero que a medida que aumentan los años las opciones de sobrevivir disminuyen.

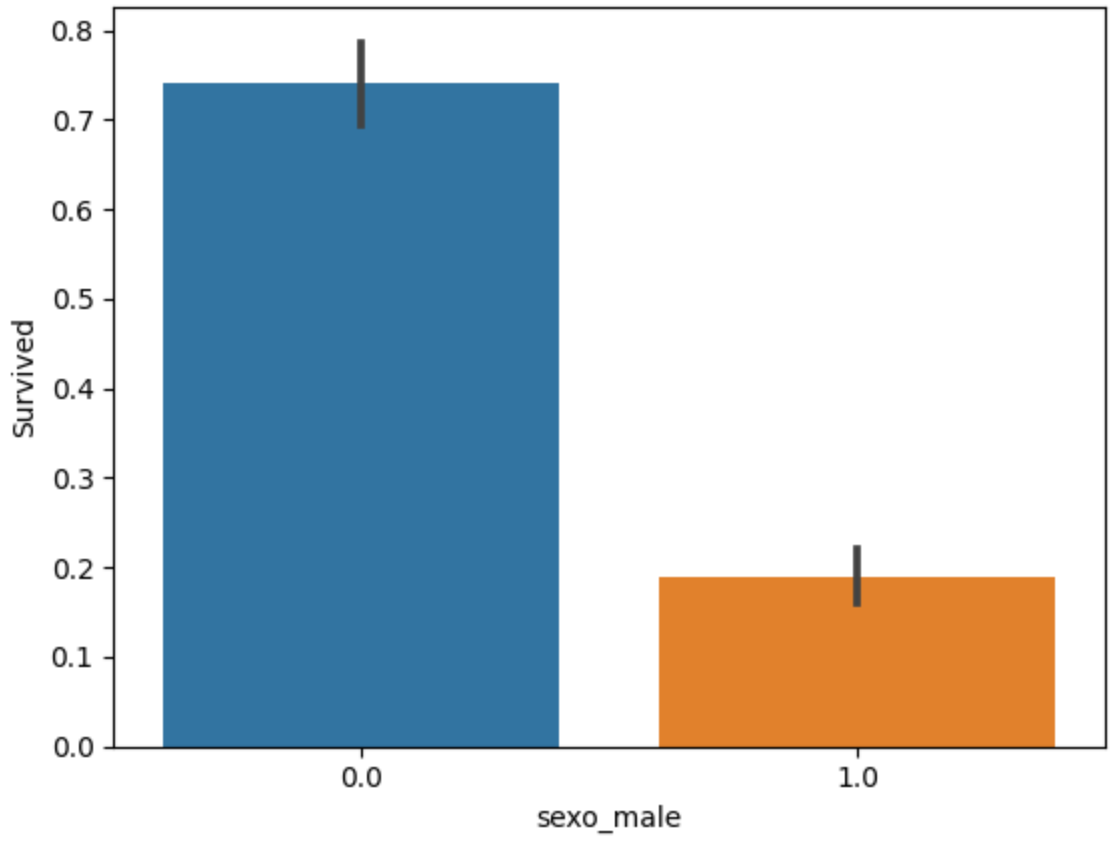
In [50]:
```
facet = sns.FacetGrid(df, hue="Survived",aspect=3)
facet.map(sns.kdeplot,'Fare',fill=True)
facet.set(xlim=(0, df['Fare'].max()))
facet.add_legend()
plt.xlim(0,20)
```

Out[50]:
```
(0.0, 20.0)
```



También hemos realizado un gráfico para ver si el precio que han pagado los pasajeros influye en las opciones de sobrevivir. Claramente se puede decir que si que influye ya que vemos que donde hay más densidad de muertes es al principio del eje x, lo que significa que los que menos pagaron tuvieron menos posibilidades de sobrevivir. Y a medida que el eje x aumenta nos encontramos con más supervivientes.

In [51]:
```
sns.barplot(x = "sexo_male", y = "Survived", data = df)
plt.show()
```



Este gráfico representa la media (entre 0 y 1) de supervivientes según el sexo. El color azul representa el sexo femenino y el naranja el masculino. A primera vista podemos observar como hay muchas más mujeres que sobreviven por encima de los hombres lo que nos lleva a la conclusión que tenían prioridad respecto al género masculino y por lo tanto había más posibilidades de no morir siendo mujer.

1.3.5 Normalización

Para mantener los mismos rangos entre los valores de las columnas procedemos a normalizar.

In [52]:
```
scaler = StandardScaler()
df.Age=scaler.fit_transform(df.Age.values.reshape(-1,1))
df.Fare=scaler.fit_transform(df.Fare.values.reshape(-1,1))
df.SibSp=scaler.fit_transform(df.SibSp.values.reshape(-1,1))
df.Parch=scaler.fit_transform(df.Parch.values.reshape(-1,1))
df.head()
```

Out[52]:

	Survived	Age	SibSp	Parch	Fare	Cabin	sexo_male	embarque_C	embar
0	0.0	-0.565736	0.432793	-0.473674	-0.502445	0.0	1.0	0.0	
1	1.0	0.663861	0.432793	-0.473674	0.786845	1.0	0.0	1.0	
2	1.0	-0.258337	-0.474545	-0.473674	-0.488854	0.0	0.0	0.0	
3	1.0	0.433312	0.432793	-0.473674	0.420730	1.0	0.0	0.0	
4	0.0	0.433312	-0.474545	-0.473674	-0.486337	0.0	1.0	0.0	

Este último paso de normalización antes del entrenamiento de los modelos, es especialmente crítico a la hora de obtener la importancia de cada característica

1.4 Modelos

Ahora procedemos a crear y entrenar los distintos modelos con los datos anteriormente revisados, para así evaluar su precisión. Importamos las librerias necesarias

In [53]:
```
#Dividir datos para evitar overfitting y poder evaluar
from sklearn.model_selection import train_test_split
#Precisión
from sklearn.metrics import accuracy_score
#Modelos
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import Perceptron
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
```

Dividimos los datos entre entrenamiento y evaluación, habiendo separado previamente el objetivo que queremos predecir, en este caso '*Survived*'

In [54]:
```
X_data = df.drop("Survived",axis=1)
Objetivo = df["Survived"]
X_train,X_test,y_train,y_test =
train_test_split(X_data,Objetivo,test_size=0.33,random_state=42)
```

Preparamos otro conjunto de datos pero sin contener la columna '*Cabin*' para comprobar que forma de entrenar los modelos es más óptima

In [55]:
```
X_data = df.drop("Survived",axis=1)
X_data2 = X_data.drop("Cabin",axis=1)
Objetivo = df["Survived"]
X_train2,X_test2,y_train2,y_test2 =
train_test_split(X_data2,Objetivo,test_size=0.33,random_state=42)
```

1.4.1 Regresión Logística

Este método estadístico trata de modelar la probabilidad de una variable cualitativa binaria, en este caso '*Survived*', en función de una o más variables como las que tenemos en el dataset. Esta modelo sigue una función sigmoide: $$y = \frac{1}{1 + e^{-f(X)}}$$

Esta función es continua y derivable. En este caso, no la podemos observar gráficamente puesto que el numero de dimensiones no es observable. Con la libreria sklearn podemos crear este modelo y reducir el error de su '*loss function*' o lo que se conoce como entrenamiento.

Creamos el modelo y lo entrenamos

In [56]:
```
modeloLR = LogisticRegression()
modeloLR.fit(X_train,y_train)
prediction_lr=modeloLR.predict(X_test)
print('La precisión del modelo de Regresión Logística es del ',
      round(accuracy_score(prediction_lr,y_test)*100,2),'%')
```

```
La precisión del modelo de Regresión Logística es del  82.03 %
```

Entrenamos el otro modelo sin la columna '*Cabin*'

In [57]:
```
modeloLR2 = LogisticRegression()
modeloLR2.fit(X_train2,y_train2)
prediction_lr2=modeloLR2.predict(X_test2)
```

```
print('La precisión del modelo de Regresión Logística es del',
      round(accuracy_score(prediction_lr2,y_test2)*100,2),'%')
```

```
La precisión del modelo de Regresión Logística es del 80.68 %
```

Como se puede comprobar, la precisión del modelo entrenado con la columna '*Cabin*', es considerablemente superior

Una vez hemos decidido el conjunto de datos con el que entrenaremos los modelos debemos optimizar los parametros de los predictores para que sean los más óptimos posible. Esto se puede realizar haciendo uso de la libreria **GridSearchCV**

In [58]:
```
#PARAMETROS
#numero de iteraciones
iteraciones = np.arange(700,1200,100)
#Algoritmo usado en la optimización
optimizadores = np.array(['lbfgs', 'liblinear', 'newton-cg',
'sag', 'saga'])

grid=GridSearchCV(estimator = modeloLR,
                  param_grid = dict(max_iter = iteraciones,
                                    solver = optimizadores
                                    ),
                  cv = 3)
grid.fit(X_train,y_train)
#Mejores parametros para usar
print(grid.best_params_)
```

```
{'max_iter': 700, 'solver': 'liblinear'}
```

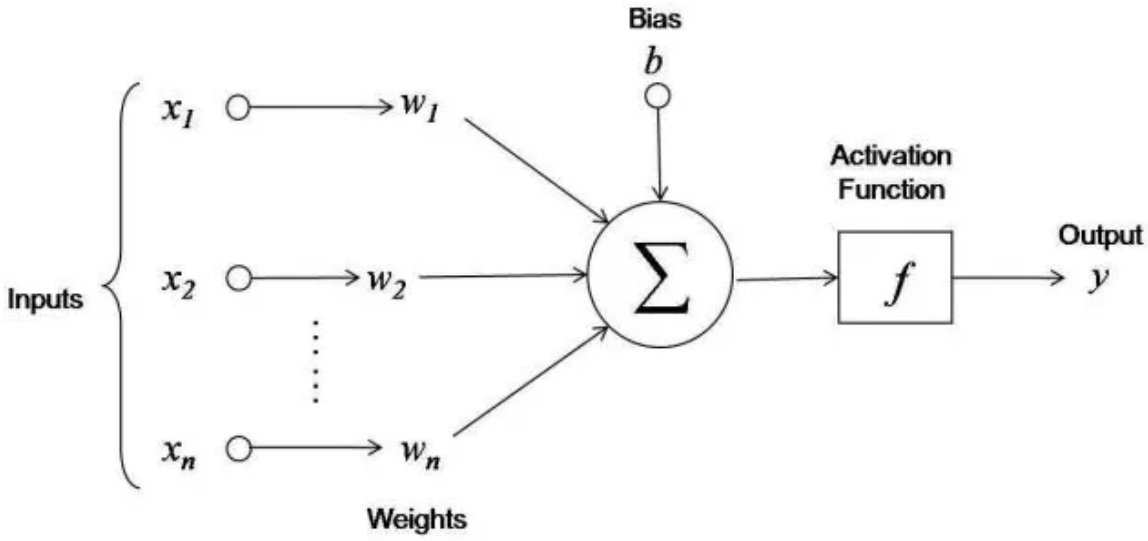
Una vez que optimizamos los parámetros posibles (a pesar de que la libreria cuente con más posiblilidades nosotros contemplamos estos) pasamos a crear el modelo de regresión logística definitivo.

In [59]:
```
new_modeloLR = LogisticRegression(max_iter=700,
solver='liblinear')
new_modeloLR.fit(X_train,y_train)
new_prediction_lr=new_modeloLR.predict(X_test)
print('La precisión del modelo de Regresión Logística es del ',
      round(accuracy_score(new_prediction_lr,y_test)*100,2),'%')
```

```
La precisión del modelo de Regresión Logística es del  82.03 %
```

1.4.2 Perceptron

Un perceptrón es un modelo matemático inspirado en una estructura y función simplificadas de una única neurona biológica. Los valores de entrada múltiples alimentan el modelo del perceptrón, el modelo se ejecuta con los valores de entrada, y si el valor estimado es el mismo que la salida requerida, entonces el rendimiento del modelo se encuentra satisfecho, por lo que los pesos no exigen cambios. De hecho, si el modelo no cumple con el resultado requerido, entonces se realizan algunos cambios en los pesos para minimizar los errores, así hasta converger un una zona con un mínimo error.



Como anteriormente hemos visto que la columna '*Cabin*' si que aumenta el rendimiento, la usaremos para entrenar el resto de nuestros modelos de predicción. Vovemos a consultar los mejores parametros para el modelo

In [60]:
```
#PARAMETROS
#numero de iteraciones
iteraciones = np.arange(700,1200,100)
#Constante para regularizar
alphas = np.array([1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01])
#Mezclar o no después de cada ciclo
mezcla = [True, False]

grid=GridSearchCV(estimator = Perceptron(),
                  param_grid = dict(max_iter = iteraciones,
                                    alpha = alphas,
                                    shuffle = mezcla),
                  cv = 3)
grid.fit(X_train,y_train)
#Mejores parametros para usar
print(grid.best_params_)
```

```
{'alpha': 1.0, 'max_iter': 700, 'shuffle': False}
```

Ahora cogemos estos parametros para entrenar el modelo

In [61]:
```
modeloPer = Perceptron(alpha=1.0,max_iter=700,shuffle=True)
modeloPer.fit(X_train,y_train)
prediction_per=modeloPer.predict(X_test)
print('La precisión del Perceptron es del ',
      round(accuracy_score(prediction_lr,y_test)*100,2),'%')
```

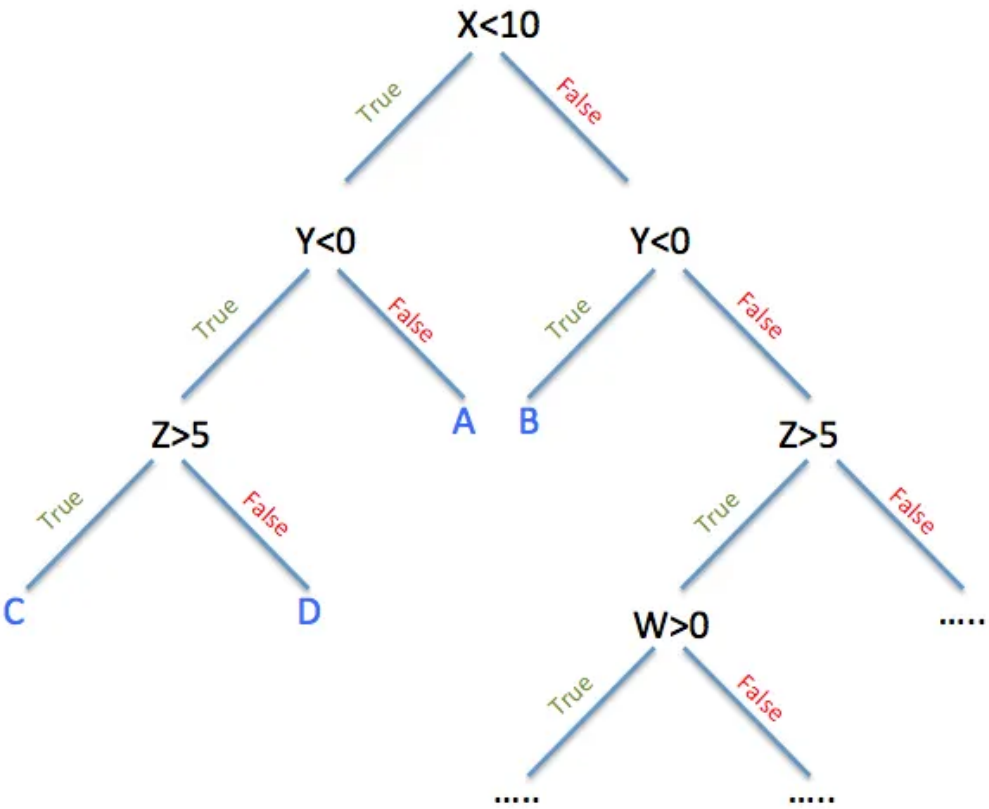
```
La precisión del Perceptron es del  82.03 %
```

1.4.3 Clasificador de Árbol de Decision

Como su nombre indica, esta técnica de machine learning toma una serie de decisiones en forma de árbol. Los nodos intermedios (las ramas) representan soluciones. Los nodos finales (las hojas) nos dan la predicción que vamos buscando. decision

En cada nodo del árbol se debe tomar una decisión entre un valor atribuido a dicho nodo. Para asignar los atributos de decisión más predecibles a los nodos se usa la entropia, que representa la homogeneidad de las muestras en ese nodo

Una vez decidios los parametros de los nodos ya está formado el modelo que decide:



Para problemas más complejos se puede utilizar el '**Random Forest Classifier**', que en pocas palabras, utiliza varios arboles de decisión, de ahí el nombre *forest*.

Como hemos visto antes, estos modelos necesitan de unos atributos para poder predecir. Estos atributos los inicializan a través de diferentes criterios, que són *gini, entropy y log_loss*, que hacen referencia a:

-La impureza de Gini (desigualdad): $H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$

-La ganancia de información: $H(Q_m) = -\sum_k p_{mk}\log(p_{mk})$

respectivamente.

Para ver que modelo y criterio es mejor entrenaremos cada uno de ellos con los mejores criterios para evaluar su rendimiento.

Creamos un dataset para comparar sus rendimientos

In [62]:
```
#PARAMETROS NECESARIOS
#criterio para predecir atributo
criterios = np.array(['gini','entropy','log_loss'])
#estrategia para separar en cada nodo
separa = np.array(['best', 'random'])
#numero de arboles en el bosque
trees = np.arange(20,120,10)
#El minimo de numero de muestras para dividir un nodo interno
divide = np.arange(2,8,1)
#El minimi numero de muestras para ser un nodo hoja
hoja = np.arange(1,9,1)

#Mejores parametros para el arbol de decisión
gridDT=GridSearchCV(estimator = DecisionTreeClassifier(),
                    param_grid = dict(criterion=criterios,
                                      splitter=separa,
                                      min_samples_split=divide,
                                      min_samples_leaf=hoja),
                    cv = 3)
gridDT.fit(X_train,y_train)
#Mejores parametros para usar
print('Parámetros para el Árbol de Decisión
són:',gridDT.best_params_)
```

```
#Mejores parametros para el RandomForest
gridRF=GridSearchCV(estimator = RandomForestClassifier(),
                    param_grid = dict(criterion=criterios,
                                      n_estimators=trees),
                    cv = 3)
gridRF.fit(X_train,y_train)
#Mejores parametros para usar
print('Parámetros para el Random Forest són:',gridRF.best_params_)
```

```
Parámetros para el Árbol de Decisión són: {'criterion': 'log_loss', 'min_sa
mples_leaf': 5, 'min_samples_split': 3, 'splitter': 'random'}
Parámetros para el Random Forest són: {'criterion': 'entropy', 'n_estimator
s': 100}
```

Como hemos visto no hemos puesto todos los paramtros para optimizar porque el tiempo de ejecución sería muy elevado

Ahora entrenamos los dos modelos con sus parametros óptimos y comparamos sus rendimientos.

Árbol de Decisión

In [63]:
```
modeloArbol= DecisionTreeClassifier(criterion='entropy',
min_samples_split=3,
                                    min_samples_leaf=5,
max_features='sqrt')
modeloArbol.fit(X_train,y_train)
prediction_tree=modeloArbol.predict(X_test)
print('La precisión del Árbol de Decisión es del ',
      round(accuracy_score(prediction_tree,y_test)*100,2),'%')
```

```
La precisión del Árbol de Decisión es del  80.0 %
```

Random Forest

In [64]:
```
modeloBosque= RandomForestClassifier(criterion='log_loss',
n_estimators=20, max_features='sqrt')
modeloBosque.fit(X_train,y_train)
prediction_forest=modeloBosque.predict(X_test)
print('La precisión del Random Forest es del ',
      round(accuracy_score(prediction_forest,y_test)*100,2),'%')
```

```
La precisión del Random Forest es del  76.61 %
```

Como podemos comprobar el rendimiento de ambos es muy parecido, siendo

prácticamente irrelevante la decisión de escoger uno frente a otro.

Otra cosa a destacar es el hecho a que debido a impedimentos de las versiones de las librerias hemos tenido que seleccionar el parámetro 'max_features' en 'sqrt'. Este parámetro únicamente hace referencia al el número de muestras máximo para considerar un split en el nodo.

Parte 2

Esta segunda parte consiste en obtener la importáncia de cada característica para cada uno de los modelos, lo que se conoce como '*feature importance*'

Los modelos asignan pesos, w_i, a los parámetros , x_i, para poder obtener su predicción.

Para realizar esta parte, primero debemos obtener los pesos de cada uno de los parametros para todos los modelos

In [175…
```
#Creamos data set para ordenar
df1=pd.DataFrame(columns=df.columns)
df1=df1.drop('Survived',axis=1)
```

Debido a la naturaleza estocástica de los algoritmos, estos pesos pueden variar. Es por eso que obtendremos la media de cada uno de ellos. Para poder hacer esto crearemos una función que nos devuelva la media de los pesos después de un número de iteraciones, según el modelo que le pasemos por parámetro

In [176…
```
def get_weights(modelo, num_iter):
    weights = []
    #Para cada columna sacamos su peso
    for i,caract in enumerate(df1.columns):
        aux = []
        for j in range(num_iter):
            #en cada iteración sacamos de nuevo los pesos
            w = modelo.coef_.tolist()
            aux.append(w[0][i-1])
        #calculamos la media
        weights.append(sum(aux)/len(aux))

    return weights
```

Cabe mencionar que para poder realizar esta comparación de importancia de las

características según los pesos del modelo necesitamos que las columnas de los datos de aprendizaje estén normalizados, es decir, han de estar en la misma escala. Es por eso que hemos realizado ese paso en la **Parte 1**

Obtenemos los valores medios de los coeficientes de la regresión logística y los añadimos a el dataset

In [177…

```
pesos = []
keys =  df1.columns
iterations = 10
#Obtenemos pesos de la regresión logística
pesos = get_weights(new_modeloLR, iterations)
dictionary = dict(zip(keys, pesos))

dflr = pd.DataFrame(dictionary, index=['Pesos Regresión Logística'])
#añadimos al dataset
df1 = pd.concat([df1, dflr], axis=0)
dflr.head()
```

Out[177]:

	Age	SibSp	Parch	Fare	Cabin	sexo_male	embarque_C	emba
Pesos Regresión Logística	-0.363951	-0.31946	-0.270631	-0.104104	0.208459	1.21697	-2.410574	0

Obtenemos los valores medios de los coeficientes del Perceptron y los añadimos al dataset

In [178…

```
pesos = []
# keys =  df1.columns
iterations = 10
#Obtenemos pesos del Perceptron
pesos = get_weights(modeloPer, iterations)
dictionary = dict(zip(keys, pesos))

dfper = pd.DataFrame(dictionary, index=['Pesos Perceptron'])
#añadimos al dataset
df1 = pd.concat([df1, dfper], axis=0)
dfper.head()
```

Out[178]:

	Age	SibSp	Parch	Fare	Cabin	sexo_male	embarque_C	embarque_
Pesos Perceptron	-1.0	-0.787064	-1.381884	0.76763	5.577751	7.0	-9.0	4

Como esto es un problema de clasificación de clases entre {0,1}, nos damos cuenta de que hemos obtenido tanto valores positivos como negativos. Los valores positivos indican una característica que predice la clase 1, mientras que los valores negativos indican características que predicen la clase 0. En otras palabras los valores mayores que cero predicen que se sobrevive, mientras que los valores menores que 0 hacen lo contrario

Debido a que todos los datos estaban normalizados, podemos afirmar o deducir que los valores más grandes ayudan a predecir esa clase con un mayor efecto. Con otras palabras, cuanto mayor sea el número (por arriba o por abajo) significa que ha sido más determinante en la predicción.

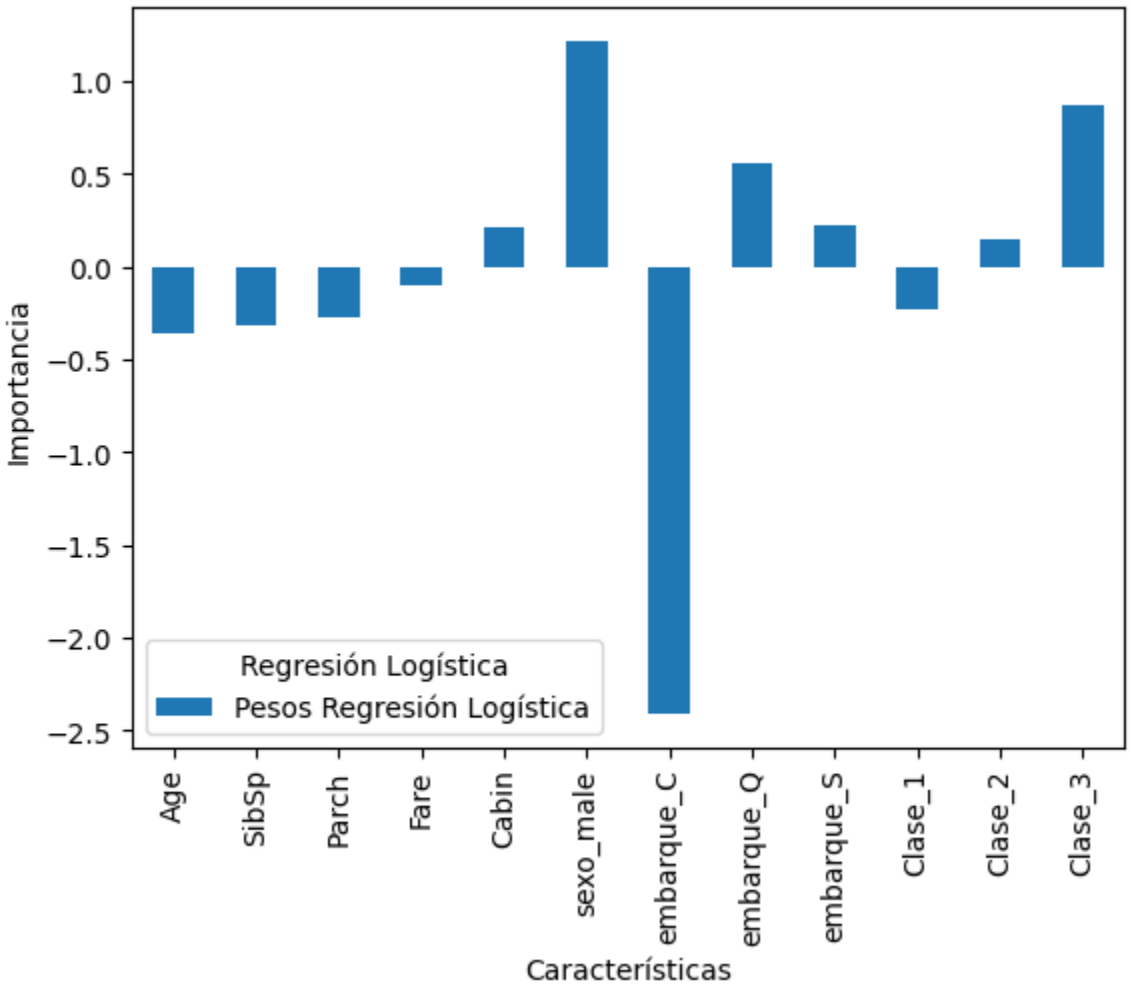
Por tanto, una vez hecho esto podemos representar las importancias

In [179…

```
rowlr = df1.iloc[0]
impLR= rowlr.plot.bar()
impLR.set_xlabel('Características')
impLR.set_ylabel('Importancia')
impLR.legend(title='Regresión Logística')
```

Out[179]:

```
<matplotlib.legend.Legend at 0x7f2d0b5f95a0>
```

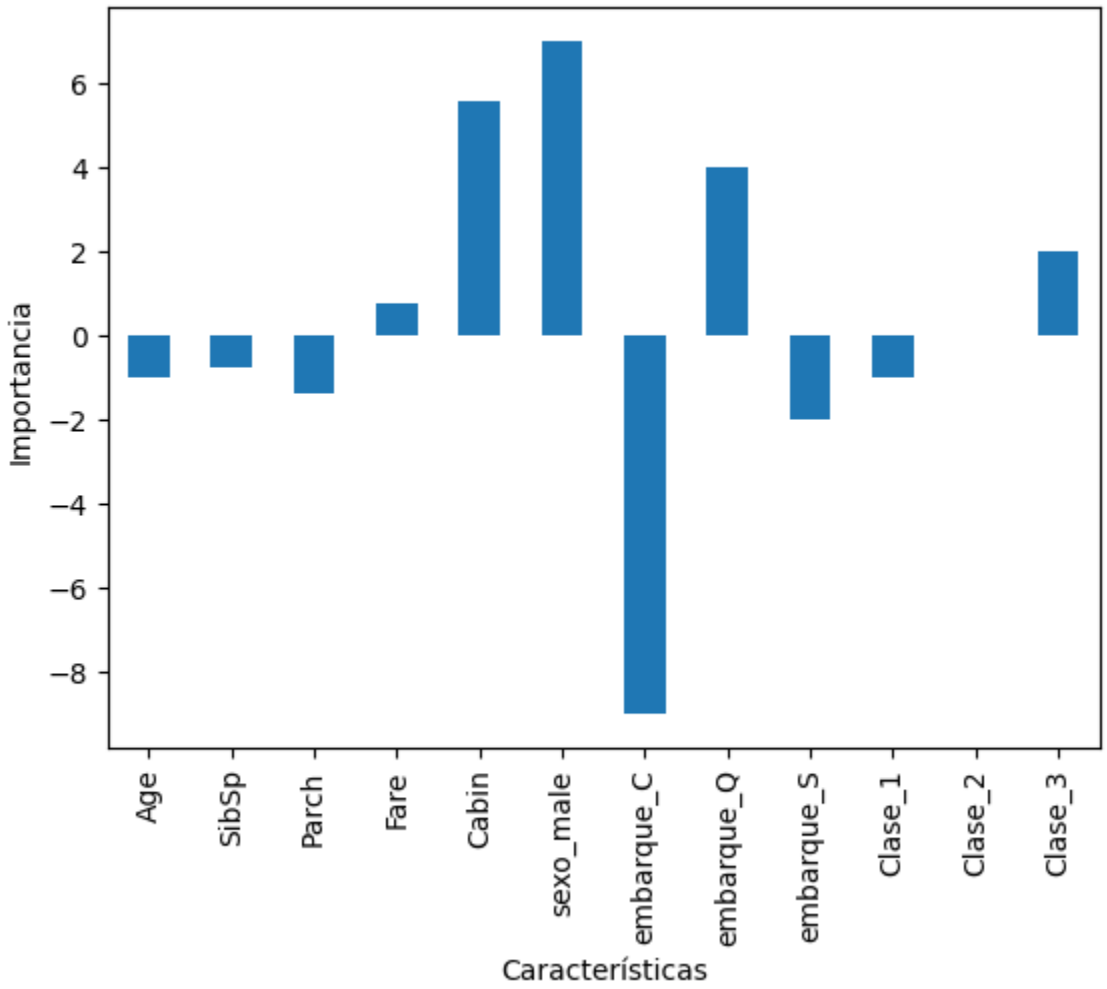


In [180…

```
rowper = df1.iloc[1]
impPercep = rowper.plot.bar()
impPercep.set_xlabel('Características')
impPercep.set_ylabel('Importancia')
impLR.legend(title='Perceptrón')
```

Out[180]:

```
<matplotlib.legend.Legend at 0x7f2d0be72ce0>
```



Una vez hecho esto observamos como la proporción de '*importance*' de cada característica es muy parecida en los dos modelos.

Observamos como el embarque C es la característica más importante para predecir tu NO supervivéncia. Entre otros valores, destacamos el sexo o la cabina.

También podriamos optado por representar la importancia en valor absoluto. Hemos decidido no hacerlo así para mantener la información negativa o positiva.

Otra opción sería incluso reescalar estos valora a una escala de, por ejemplo, <0,100>, para poder observar los porcentajes '%' de reelevancia. Pero creemos que es suficiente con ver las características más y menos importantes de esta forma.

Debido a lo anterior, vemos que, aunque los números en el perceptron sean mayores la relación entre ellos es la misma, como se puede deducir de los gráficos anteriores. Es por eso que no hemos encontrado que fuera necesario reescalarlos, pues contamos con la representación visual

Por último, obtenemos los valores medios de las feature importances del último modelo y los añadimos al dataset. Para hacer esto usaremos únicamente el **RandomForestClassifier** ya que es ligéramente superior al **DecisionTreeClassifier**

Como el **RandomForestClassifier** no tiene el atributo .coef__ (más adelante explicaremos el porqué), debemos crear otra función para obtener los pesos. Para ello utilizaremos el atributo `*feature_importances_*'.

In [181…

```
pesos = []
keys =  df1.columns
iterations = 10
#Obtenemos las feature importances del modelo
pesos = modeloBosque.feature_importances_

dictionary = dict(zip(keys, pesos))

dfrf = pd.DataFrame(dictionary, index=['Pesos RandomForest'])
#añadimos al dataset
df1 = pd.concat([df1, dfrf], axis=0)
dfrf.head()
```

Out[181]:

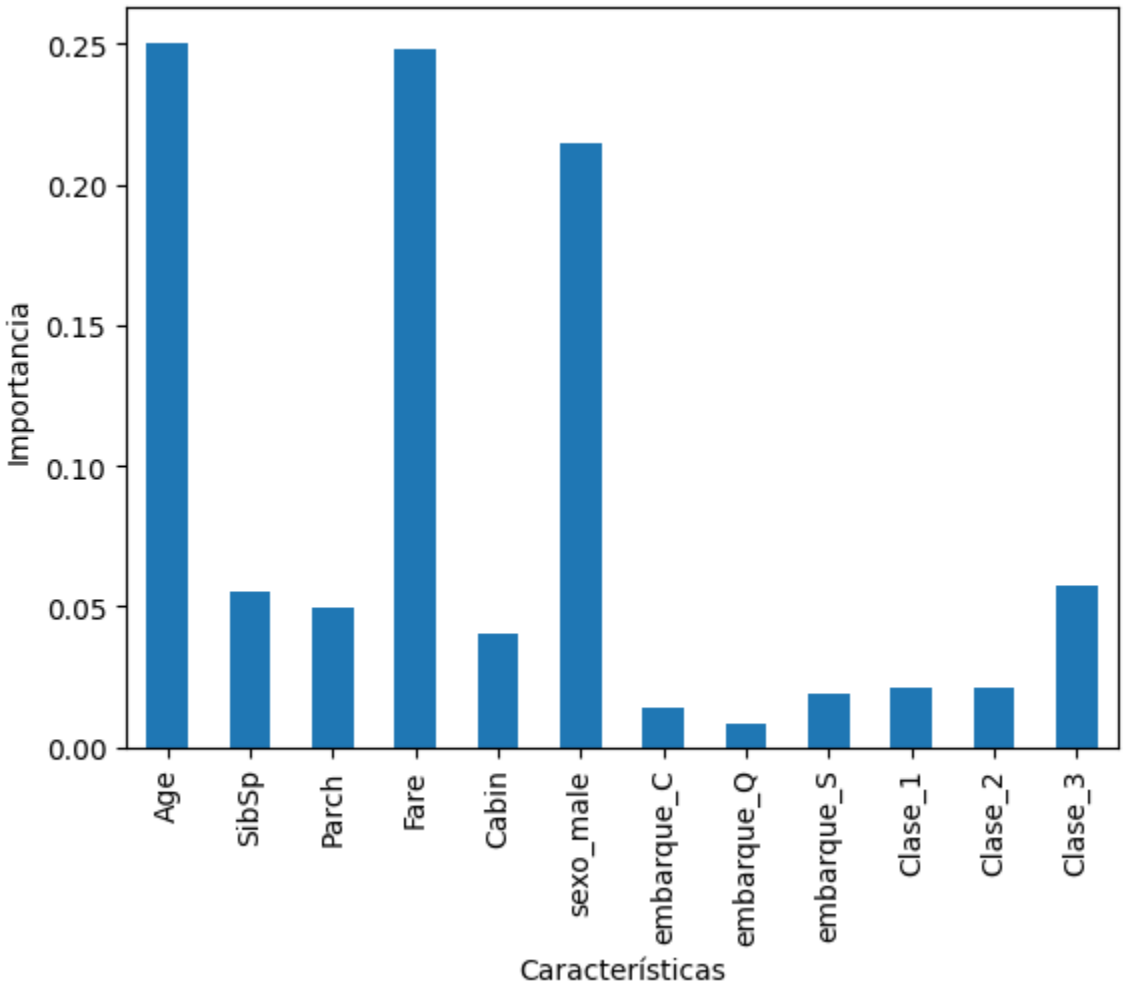
	Age	SibSp	Parch	Fare	Cabin	sexo_male	embarque_C	em
Pesos RandomForest	0.250379	0.055424	0.049692	0.248036	0.040539	0.214865	0.014193	

In [182…

```
rowper = df1.iloc[2]
impPercep = rowper.plot.bar()
impPercep.set_xlabel('Características')
impPercep.set_ylabel('Importancia')
impLR.legend(title='RandomForest')
```

Out[182]:

```
<matplotlib.legend.Legend at 0x7f2d0bcb8730>
```



En grandes rasgos podemos ver como la edad es la característica que más pesa en este modelo junto con el precio que han pagado

2.4 Analisis, evaluación y explicación

Una vez hemos hecho los anteriores cálculos encontramos una clara diferencia entre los valores de la Regresion logística y Perceptron con el RandomForest.

Esta diferencia se debe a que los dos primeros modelos són paramétricos, es decir, realizan una expresión matemática dependiendo de los parámetros. Estos pesos w_i que asigna a las características són los coeficientes obtenidos.

Estos coeficientes solo reflejarían la importancia en el caso de que estuvieran normalizados.

Por el otro lado, los modelos no paramétricos tienen diferentes formas de medir la importáncia. En el caso de los árboles, esto se mide comprobando como de cerca esta la variable del nodo raíz.

Esto refleja que los modelos no paramétricos como los arboles de decisión necesitan algo

de trabajo extra para obtener la '*feature importance*', a pesar de que nosotros la podamos obtener con un único método.

Debido a esto, el atributo feature_importances_ del modelo **RandomForest** devuelve los valores ya en un rango <0,1>

Para ver el resultado final, vamos a mostrar todas las '*feature importances*' de cada modelo en un dataframe.

In [183…
```
df1=df1.transpose()
```

Para ello, esta vez formatearemos las filas de los dos primeros modelos para que muestren un rango de 0 a 1 de valores positivos y podamos comparar con el último modelo

In [184…
```
for c in df1.columns:
    #Obtenemos el valor absoluto
    df1[c]=df1[c].abs()
    #Ponemos en un rango entre {0,1} de
    #forma que sume 1
    df1[c]=df1[c]/df1[c].sum()
    #obtenemos el porcentaje
    df1[c]=df1[c]*100

df1
```

Out[184]:

	Pesos Regresión Logística	Pesos Perceptron	Pesos RandomForest
Age	5.249445	2.897347	25.037893
SibSp	4.607731	2.280398	5.542370
Parch	3.903448	4.003797	4.969190
Fare	1.501541	2.224090	24.803620
Cabin	3.006710	16.160683	4.053939
sexo_male	17.552953	20.281432	21.486456
embarque_C	34.768898	26.076126	1.419320
embarque_Q	8.110489	11.589390	0.829522
embarque_S	3.244680	5.794695	1.874105
Clase_1	3.301941	2.897347	2.132026
Clase_2	2.120140	0.000000	2.113714
Clase_3	12.632023	5.794695	5.737846

Aqui podemos comparar de forma clara el porcentaje de importancia de las características

para cada modelo. En este recuadro podemos comprobar como como el **RandomForest** da valora a variables que no parecen tan relevantes en los otros modelos como puede ser la edad o el fare, que son casi irrelevantes para los predictores paramétricos.

Por otro lado, el Perceptron y la Regresión logística le dan una gran importancia al embarque C, mientras que al modelo de árbol no lo le da importáncia.

Como conclusión vemos claramente como el sexo es la característica que importa de gran manera en todos y cada uno de los modelos predictores entrenados.