Big Data y Machine Learning con Python

Parte practica

Prediciendo la supervivencia en el Titanic

Para introducir los conceptos comentados en la presentacion teorica, vamos a participar en una competicion de Kaagle sobre la supervivencia de los pasajeros del Titanic.

Esta competencion es una de las más populares para introducirse en el mundo de Machine Learning. La idea es conociendo lo que les ocurrio a una parte de los pasajeros del Titanic, predecir lo que le pasaría al resto.

Para poder participar necesitariamos en primer lugar instalar python en nuestro sistema. También necesitamos instalar algunas bibliotecas para administrar datos.

En concreto vamos a utilizar esas bibliotecas:

```
pandas: Nos permitira manejar diferentes formatos de datos (en nuestro caso trabajaremos con archivos.csv)
numpy: para trabajar con matrices, vectores y otros conjuntos de datos.
scikit-learn: Una biblioteca específica para Machine Learning que incluye diferentes estructuras de prediccion.
```

Utilizamos también otra biblioteca para ignorar los 'warnings' ya que python lanza un montón de advertencias sobre las diferentes versiones del lenguaje y nos distareria de nuestro verdadero objetivo.

La forma más fácil de empezar a trabajar en Python es instalar Anaconda, un gestor de paquetes que incluye una gran cantidad de bibliotecas populares incluidas por supuesto, éstas.

Anaconda instala ademas este Notebook que estamos utilizando, Jupyter, que como vereis es muy util para presentar codigo ya que permite ir ejecutandolo paso a paso e ir comentadolo.

Para participar en una competicion de Kagle debemos ir a su sitio web descargar los documentos necesarios, que en este caso son dos:

Https://www.kaggle.com/c/titanic/data

El primero es una lista de una parte de los pasajeros del Titanic e incluyen un campo que nos dice si sobrevivieron o no. El otro es la lista del resto de pasajeros pero sin este campo y es el que necesitamos para construir nuestras predicciones y enviar a Kaggle.

Lo primero que haremos sera cargar nuestro documento y transformarlo en un DataFrame que es el tipo de objeto que maneja la libreria panda y con el que trabajaremos.

Ahora podemos echar un vistazo a nuestros datos usando un método muy útil de la biblioteca panda:

```
head ()
```

que nos muestran las cinco primeras filas de nuestro archivo

In [1]:
```
# Ignore warnings
import warnings
warnings.filterwarnings('ignore')

# Import the Pandas library
import pandas as pd

# Load the train dataset
train_url = "dataset/train.csv"
train = pd.read_csv(train_url)

# Take a look to the content
train.head()
```

Out[1]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Con este metodo podemos ver el nombre de las columnas que contiene nuestro

Otro comando interesante para saber lo que contiene nuestro documento es

```
describe()
```

que hace un resumen del contenido

```
In [2]: train.describe()
```

Out[2]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Todos los DataFrame contienen tambien un atributo

```
shape
```

que nos dira las dimensiones de nuestros datos.

```
In [3]: train.shape
```

```
Out[3]: (891, 12)
```

Como hemnos visto, nuestro documento incluye una columna llamada ‘Survived’ a la que podemos acceder con la anotacion estandar de corchetes. y luego como el metodo .value_counts() podemos hacernos una idea de lo que ocurrio.

In [4]:
```
print(train["Survived"].value_counts())
```

```
0    549
1    342
Name: Survived, dtype: int64
```

Cuando se trabaja con datos suele ser mas normal que en lugar de trabajar con datos absolutos se trabaje con porcentajes, y por supuesto para esto tambien panda nos brinda un parametro que podemos pasar al metodo .value_counts(); normalize

In [5]:
```
# Percentage of Survivor column
print(train["Survived"].value_counts(normalize = True))
```

```
0    0.616162
1    0.383838
Name: Survived, dtype: float64
```

Asi pues, con estos datos ya sabemos que lo mas habitual es que cualquiera que se subiera a bordo del Titanic acabara ahogado, asi que nuestra primera prediccion podria ser decir que todos los que suban moriran. Asi que adelante, creemos nuestra primera prediccion.

Para hacerla cogeremos el archivo llamado test que es el que se nos proporciona para probar nuestos algoritmos, haremos una copia del mismo y realizaremos nuestra prediccion.

Si queremos subir nuestra prediccion a Kaggle necesitamos cumplir los requisitos de participacion y que podemos ver en: https://www.kaggle.com/c/titanic#evaluation (https://www.kaggle.com/c/titanic#evaluation)

Ahi podemos leer:

```
Submission File Format
You should submit a csv file with exactly 418 entries plus a header row.
Your submission will show an error if you have extra columns (beyond PassengerId and Survived) or rows.
The file should have exactly 2 columns:
PassengerId (sorted in any order)
Survived (contains your binary predictions: 1 for survived, 0 for deceased)
```

Asi que manos a la obra.

In [6]:
```
# Import the Numpy library
import numpy as np

# Load the test dataset
test_url = "dataset/test.csv"
test = pd.read_csv(test_url)

# Duplicate our dataset
myTestTest = test.copy()

# Set Survived column to 0
myTestTest['Survived'] = 0

# Create a data frame with two columns: PassengerId & Survived. Survived contains our predictions
PassengerId = np.array(myTestTest["PassengerId"]).astype(int)
Survived = np.array(myTestTest["Survived"]).astype(int)
testSolution = pd.DataFrame(Survived, PassengerId)

# Write our solution to a csv file with the name testSolution.csv
testSolution.columns = ['Survived']
testSolution.to_csv("testSolution.csv", index_label = ["PassengerId"])
```

Si subimos esta solucion a Kaggle veremos que obtenemos una puntuacion del 62% lo que no esta mal para empezar, pero como hemos visto ha sido una prediccion manual, vamos a intentar mejorar nuestra puntuacion utilizando tecnicas de Machine Learning.

En primer lugar vamos a utlizar un arbol de decisiones.

- #### Arbol de decisiones (Decision Tree): Un arbol de decisiones es una estructura logica a la que se le proporciona cierta informacion y siguiendo unas reglas nos ofrece un resultado.

Lo primero vamos a duplicar nuestros datos para no modificar los originales. Despues los normalizamos y harmonizamos.

- #### Normalizamos y harmonizar datos: Estos dos conceptos son muy similares aunque incluyen un pequeño matiz que los diferencia: Normalizar se refiere al hecho de comprobar que nuestros datos se ajustan a los requerimientos que los datos deben tener, y harmonizar se refiere a ajustar los datos a los requerimientos necesarios para un trabajo especifico. Como se puede ver es la diferencia es minima y de hecho en muchos casos las palabras se usan indistintamente.

Vamos a asignar valores numericos a nuestros datos que son mas faciles de tratar. Asignaremos un 1 a las mujeres y un 0 a los hombres. Tambien vamos a asignar 0 a la clase S, 1 a la clase C y 2 a la clase Q de la columna "Embarked". Tambien, para asegurarnos de que todos los pasajeros pertenecen a una clase, vamos a rellenar los posibles huecos con la clase mas comun. Veamos cual es.

In [7]:
```
print(train["Embarked"].describe())
```

```
count     889
unique      3
top         S
freq      644
Name: Embarked, dtype: object
```

Asi pues, rellenaremos los posibles huecos con la S. Para normalizar tambien la columna "Age", vamos a asignar a aquellos pasajeros que no tengan edad, la media de la edad de los que si la tienen.

In [8]:
```
# Duplicate our dataset
myTrain = train.copy()

# Convert the male and female groups to integer form
myTrain["Sex"][myTrain["Sex"] == "male"] = 0
myTrain["Sex"][myTrain["Sex"] == "female"] = 1

# Fill the gaps in the Embarked variable
myTrain["Embarked"] = myTrain["Embarked"].fillna("S")

# Convert the Embarked classes to integer form
myTrain["Embarked"][myTrain["Embarked"] == "S"] = 0
myTrain["Embarked"][myTrain["Embarked"] == "C"] = 1
myTrain["Embarked"][myTrain["Embarked"] == "Q"] = 2

# Fill the gaps in the Age variable
myTrain["Age"] = myTrain["Age"].fillna(myTrain["Age"].median())
```

Una vez que hemos harmonizado nuestros datos podemos empezar a crear nuestro primer arbol de decision. Vamos a usar otra popular libreria para hacerlo: sklearn. Con esta libreria podemos crear el arbol, y despues decirle lo que queremos conseguir con el (target), y con que elementos queremos que lo logre (features).

Ahora podemos ver la importancia que nuestro algoritmo da a cada elemento con el atributo .feature*importances*, y comprobar que puntuacion obtendremos con el

In [9]:
```
# Import 'tree' from scikit-learn library
from sklearn import tree

# Create the target and features
target = myTrain["Survived"].values
features_one = myTrain[["Pclass", "Sex", "Age", "Fare"]].values

# Fit your first decision tree: my_tree_one
my_tree_one = tree.DecisionTreeClassifier()
my_tree_one = my_tree_one.fit(features_one, target)

# Look at the importance and score of the included features
print(my_tree_one.feature_importances_)
print(my_tree_one.score(features_one, target))
```

```
[ 0.12985597  0.31274009  0.22882793  0.328576  ]
0.977553310887
```

Guauuuuu!!!!!! Obtendremos mas de un 97% de acierto. Con un algoritmo asi podemos hacernos ricos, teniendo en cuanta lo dificil que es realizar algoritmos que superen el 80% de acierto.

Pero antes de subir nuestra prediccion a Kaggle echemos un ojo al arbol de decision que hemos creado

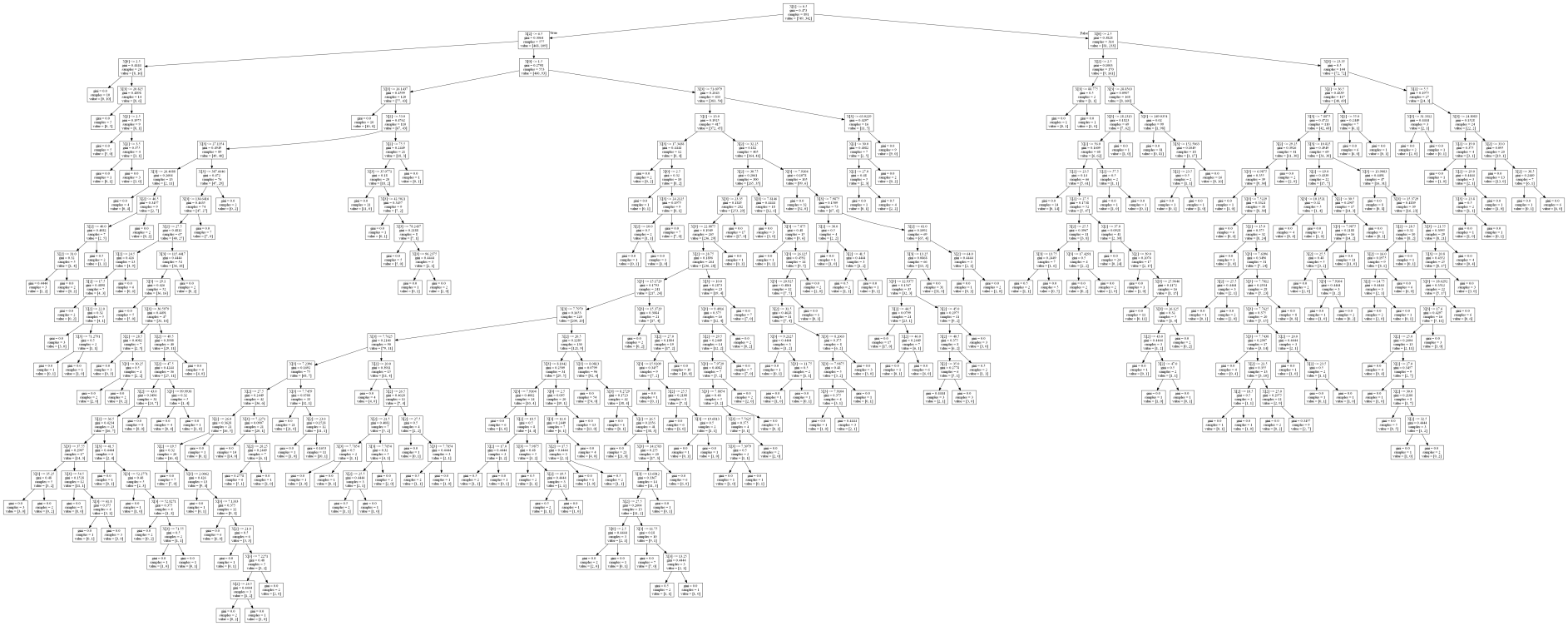
Para hacerlo vamos a descargarnos el archivo .dot que es el formato usado por la libreria sklearn.

In [10]:
```
with open("decisionTree.dot", 'w') as archivo_dot:
            tree.export_graphviz(my_tree_one, out_file = archivo_dot)
```

Y luego podemos convertir este archivo a png usando el programa del site de graphviz (herramientas open source para el manejo de esos archivos .dot)

http://www.graphviz.org/Download_windows.php (http://www.graphviz.org/Download_windows.php)

Para evitar perdidas de tiempo he realizado yo anteriormente todo este proceso y este es el grafico del arbol de decision que hemos utilizado:



Bueno, como podeis ver es bastante complicado, pero eso no nos importa, lo importante es su eficacia y ya hemos visto que alcanza casi un 98%.

Bueno, usemos nuestro arbol de decision para hacer nuestra primera prediccion usando ya tecnicas de Machine Learning.

Para ello tenemos que normalizar y harmonizar tambien los datos de test que vamos a usar, y ademas, si estudiamos los datos, veriamos que en este caso tambien esta vacia la posicion 152 referente a la 'fare'(tarifa), asi que nuevamente le asignaremos la media del resto de datos que si poseemos.

Despues, creamos nuestra pimera preddiccion con los datos del test, y creamos nuestro archivo .csv que subiremos a Kaggle.

In [11]:
```
# Duplicate our dataset
myFirstTest = test.copy()

# Impute the missing value with the median
myFirstTest.Fare[152] = myFirstTest["Fare"].median()

# Convert the male and female groups to integer form
myFirstTest["Sex"][myFirstTest["Sex"] == "male"] = 0
myFirstTest["Sex"][myFirstTest["Sex"] == "female"] = 1

# Fill the gaps in the Embarked variable
myFirstTest["Embarked"] = myFirstTest["Embarked"].fillna("S")

# Fill the gaps in the Age variable
myFirstTest["Age"] = myFirstTest["Age"].fillna(myFirstTest["Age"].median())

# Convert the Embarked classes to integer form
myFirstTest["Embarked"][myFirstTest["Embarked"] == "S"] = 0
myFirstTest["Embarked"][myFirstTest["Embarked"] == "C"] = 1
myFirstTest["Embarked"][myFirstTest["Embarked"] == "Q"] = 2

# Extract the features from the test set: Pclass, Sex, Age, and Fare.
myFirstTest_features = myFirstTest[["Pclass", "Sex", "Age", "Fare"]].values

# Make a prediction using the test set
myFirstprediction = my_tree_one.predict(myFirstTest_features)

# Create a data frame with two columns: PassengerId & Survived. Survived contains our predictions
PassengerId = np.array(myFirstTest["PassengerId"]).astype(int)
firstSolution = pd.DataFrame(myFirstprediction, PassengerId)

# Write our solution to a csv file with the name firstSolution.csv
firstSolution.columns = ['Survived']
firstSolution.to_csv("firstSolution.csv", index_label = ["PassengerId"])
```

Si subimos nuestra solucion a Kaggle obtendremos una decepcionante puntuacion del 72%... que ha pasado?

Es un buen momento para explicar otro interesante concepto del Machine Learning: el Sobreentrenamiento.

- #### Sobreentrenamiento (Overfitting): Cuanto entrenamos a nuestro modelo es posible que lo hagamos tan bien que solo se adapte a los datos que usamos para entrenarlo. La idea es que nuestro modelo pueda generalizarse con cualquier tipo de datos que usemos, no solo los que usamos de entrenamiento.

En nuestro caso, hemos construido nuestro arbol de decision tan complicado que solo es capaz de amoldarse bien a los datos usados para entrenarlo, y de hecho, con esos datos, obtenia hasta un 97% de aciertos, pero al usar otro datos su rendimiento baja hasta el 72%
Para evitar el sobreentrenamiento podemos limitar la profundidad de nuestro arbol de decisiones y asi aumentar su capacidad de genaralizacion.
Para hacer eso usamos dos argumentos a la hora de construir nuestro arbol: max_depth, que determina la profuncidad del mismo, y min_samples_split que indica las minimas muestra necesarias para seguir profundizando.
Uno de los decisiones que se deben tomar a la hora de crear un buen algoritmo es el uso de estas variables.

In [12]:
```
# Fit your first decision tree: my_tree_one
max_depth = 4
min_samples_split = 5
my_tree_two = tree.DecisionTreeClassifier(max_depth = max_depth, min_samples_split = min_samples_split, rando
m_state = 1)
my_tree_two = my_tree_two.fit(features_one, target)

print(my_tree_two.score(features_one, target))

# Make a prediction using the test set
mySecondPrediction = my_tree_two.predict(myFirstTest_features)

# Create a data frame with two columns: PassengerId & Survived. Survived contains our predictions
PassengerId = np.array(myFirstTest["PassengerId"]).astype(int)
secondSolution = pd.DataFrame(mySecondPrediction, PassengerId)

# Write our solution to a csv file with the name secondSolution.csv
secondSolution.columns = ['Survived']
secondSolution.to_csv("secondSolution.csv", index_label = ["PassengerId"])
```

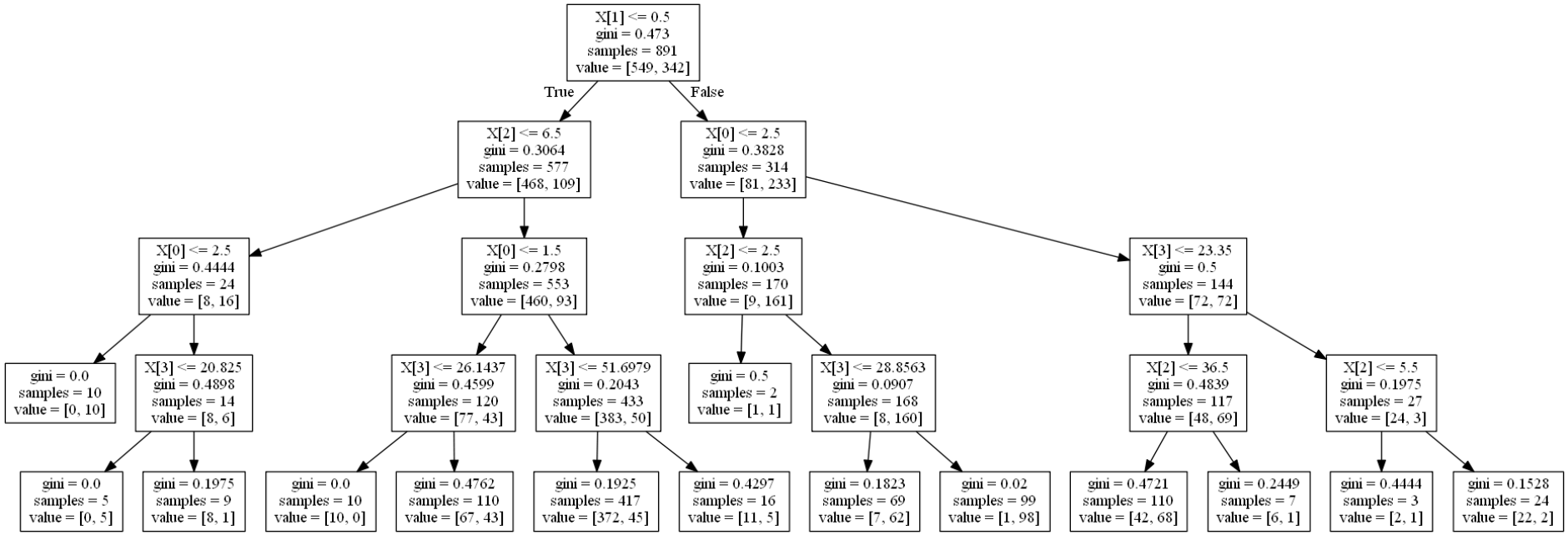
```
0.832772166105
```

Como vemos el resultado de este algoritmo es sensiblemente menor, un 83%, pero por el contrario, si subimos el resultado de su uso en los datos de prueba alcanzamos un 76%, que es mayor que el anterior.

Ahora podemos tambien crear la imagen para este nuevo arbol

```
In [13]: with open("secondSolution.dot", 'w') as archivo_dot:
             tree.export_graphviz(my_tree_two, out_file = archivo_dot)
```

Como podemos ver, es mucho mas simple



Como hemos visto, este arbol de decision consiguio mejorar nuestra puntuacion en Kaggle, pero aun podemos mejorarla usando otro interesante concepto de Machine Learning: los Bosques Aleatorios

- #### Bosques Aleatorios (Random Forest): Un Bosque Aleatorio es basicamente un bosque de arboles de decisiones. Podemos fabricar gran cantidad de arboles de decisiones y evaluar el resultado de cada una de ellos, y asignar el resultado final a la decision mas comun de esos arboles.

Vamos a crear un algoritmo usando esta tecnica.

In [14]:
```
# Create train_two with the newly defined feature
myTrain_two = myTrain.copy()

# Import the `RandomForestClassifier`
from sklearn.ensemble import RandomForestClassifier

# Creating features for train
features_forest = myTrain_two[["Pclass", "Age", "Sex", "Fare", "SibSp", "Parch", "Embarked"]].values


# Building and fitting my_forest
forest = RandomForestClassifier(max_depth = 4, min_samples_split=2, n_estimators = 100, random_state = 1)
my_forest = forest.fit(features_forest, target)

# Print the score of the fitted random forest
print(my_forest.score(features_forest, target))

# Duplicate our test dataset
mySecondTest = myFirstTest.copy()

# Creating features for test
test_features = mySecondTest[["Pclass", "Age", "Sex", "Fare", "SibSp", "Parch", "Embarked"]].values

# Make a prediction using the test set
myLastPrediction = my_forest.predict(test_features)

# Create a data frame with two columns: PassengerId & Survived. Survived contains our predictions
PassengerId = np.array(mySecondTest["PassengerId"]).astype(int)
LastSolution = pd.DataFrame(myLastPrediction, PassengerId)

# Write our solution to a csv file with the name secondSolution.csv
LastSolution.columns = ['Survived']
LastSolution.to_csv("LastSolution.csv", index_label = ["PassengerId"])
```

```
0.841750841751
```

Con este nuevo algoritmo conseguimos un resultado de cerca del 79%. Es un buen resultado. De todos modos seguro que aun podriamos mejorarlo usando nuevas formulas y encontrando nuevas variables. Por ejemplo, era importante la situacion de las cabinas de los supervivientes? Tiene alguna influencia la edad de los pasajeros? Encontrar esta correlaciones y crear algoritmos que los contemplen es el trabajo de los expertos en Machine Learning, que a veces les lleva a hacer del mundo un lugar mejor, y otras tan solo ha ganar un millon de dolares.

Muchas gracias.

In [ ]: