

Machine Learning

By Daniel Lázaro Lázaro

ÍNDICE

Preprocesamiento de datos

Dimensionality Reduction (RD)

RD. 1. PCA

RD. 2. LDA

RD. 3. ICA

APRENDIZAJE SUPERVISADO REGRESIÓN (SL.R)

SL.R. 1. Linear Regression

SL.R. 2. Multiple Linear Regression

*SL.R. *. Métodos de filtrado/selección de variables*

SL.R. 3. Polynomial Regression

SL.R. 4. Support Vector Machine (Regression) (SVM)

SL.R. 5. Decision Trees (Regression)

*SL.R. *. Métodos de Ensamblaje de Modelos*

SL.R. 6. Random Forest (Regression)

*SL.R. *. Métricas de Evaluación de Modelos*

APRENDIZAJE SUPERVISADO CLASIFICACIÓN (SL.C)

SL.C. 1. Logistic Regression

SL.C. 2. K-Nearest Neighbors (KNN)

SL.C. 3. Support Vector Machine (Classification) (SVM)

SL.C. 4. Naive Bayes (NB)

SL.C. 5. Decision Trees (Classification)

SL.C. 6. Random Forest (Classification)

NEURAL NETWORKS (NN)

NN.1. Binary Classification**NN.2. Convolutional Neural Networks (CNN)****NN.3. Recurrent Neural Networks (RNN)****NN.4. Artificial Neural Networks (ANN)****APRENDIZAJE NO SUPERVISADO. CLUSTERS (NSL. C.)****Métodos de Selección del Número de Clusters (MSNC)**

MSNC. 1. Método del Codo

MSNC. 2. Método de la Silueta

NSL. C. 1. K-Means

NSL. C. 2. Agrupamiento Jerárquico Aglomerativo (AHC)

NSL. C. 3. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

NSL. C. 4. Expectation Maximization algorithm(EM) / Gaussian Mixture Model (GMM)

NSL. C. 5. Mean Shift

Usaremos SciPy, NumPy, y Pandas, ya que permiten al ordenador aprender álgebra lineal y métodos kernel

In [1]:

```
# Librería NumPy
#Potentes estructuras de datos, matrices y matrices multidimensionales (también cálculo
s eficientes en estas matrices)
#numpy ocupa menos memoria en cuanto a las listas de python
# np.array() para listas o matrices
import numpy as np
a = np.array([1,2,3]) # 1D
b = np.array([(1,2,3),(4,5,6)]) # 2D

print('a:',a)
print('b:',b)
```

```
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\_distributor_init.py:3
 0: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\.libs\libopenblas.NOIJ
JG62EMASZI6NYURL6JBKM4EVBGM7.gfortran-win_amd64.dll
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\.libs\libopenblas.PYQH
XLVVQ7VESDPUVUADXEVJOBGHJPAY.gfortran-win_amd64.dll
  warnings.warn("loaded more than 1 DLL from .libs:\n%s" %

a: [1 2 3]
b: [[1 2 3]
 [4 5 6]]
```

In [2]:

```
#Algunas formas de generar matrices según numpy

ones = np.ones((2,3))

zeros = np.zeros((3,2))

random = np.random.random((3,5))

empty = np.empty((2,4))

fully = np.full((2,3),6)

arange = np.arange(0,20,5)

linspace = np.linspace(0,2,5)

eye = np.eye(3)

identity = np.identity(4)

print(ones)
print()
print(zeros)
print()
print(random)
print()
print(empty)
print()
print(fully)
print()
print(arange)
print()
print(linspace)
print()
print(eye)
print()
print(identity)
print()
```

```
[[1. 1. 1.]
 [1. 1. 1.]]

[[0. 0.]
 [0. 0.]
 [0. 0.]]]

[[0.18503351 0.32055624 0.66848525 0.46829303 0.74778427]
 [0.79041062 0.78807206 0.51238248 0.78300553 0.75580105]
 [0.16164548 0.47269972 0.08168363 0.05937184 0.95458377]]]

[[1.37962185e-306 1.11260348e-306 1.44635488e-307 1.33511562e-306]
 [1.69121775e-306 1.69122046e-306 1.05700260e-307 1.44635488e-307]]]

[[6 6 6]
 [6 6 6]]

[ 0 5 10 15]

[0. 0.5 1. 1.5 2. ]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]]

[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]]
```

In [3]:

```
print('ndim:',identity.ndim)
print('dtype:',identity.dtype)
print('shape:',identity.shape)
print('size:',identity.size)
```

```
ndim: 2
dtype: float64
shape: (4, 4)
size: 16
```

In [4]:

```
print('linspace:',linspace)
print()
print('shape:',linspace.shape)
print()
print('reshape',linspace.reshape(5,1))
```

```
linspace: [0. 0.5 1. 1.5 2. ]
```

```
shape: (5,)
```

```
reshape [[0.
 [0.5]
 [1.]
 [1.5]
 [2.]]]
```

In [5]:

```
print('b:',b)
print()
print('b[1,1]:',b[1][1])
```

b: [[1 2 3]
[4 5 6]]

b[1,1]: 5

In [6]:

```
print(a)
print()
print('min:',np.min(a))
print()
print('max:',np.max(a))
print()
print('sum:',np.sum(a))
print()
print('sqrt:',np.sqrt(a))
print()
print('std:',np.std(a))
print()
```

[1 2 3]

min: 1

max: 3

sum: 6

sqrt: [1. 1.41421356 1.73205081]

std: 0.816496580927726

In [7]:

```
x=np.array([(1,2,3),(4,5,6)])
y=np.array([(1,2,3),(4,5,6)])

print('x:',x)
print()
print('y:',y)
print()
print('x+y:',x+y)
print()
print('x-y:',x-y)
print()
print('x*y:',x*y)
print()
print('x/y:',x/y)
print()
```

x: [[1 2 3]
[4 5 6]]

y: [[1 2 3]
[4 5 6]]

x+y: [[2 4 6]
[8 10 12]]

x-y: [[0 0 0]
[0 0 0]]

x*y: [[1 4 9]
[16 25 36]]

x/y: [[1. 1. 1.]
[1. 1. 1.]]

In [8]:

```
#Librería Pandas para cargar, preparar, modelar, manipular y analizar datos
import pandas as pd
#Generalmente un dataframe o estructura bidimensional de datos dada generalmente por los datos, los índices y las columnas
#se puede indicar el nombre de filas o columnas
#pandas da índices a las matrices, tanto a las filas como a las columnas
#datos heterogéneos

data = np.array([['', 'Col 1', 'Col 2'], ['Fila 1', 11, 22], ['Fila 2', 33, 44]])

po=pd.DataFrame(data=data[1:,1:],index=data[1:,0],columns=data[0,1:])

print('data:')
print(data)
print('po:')
print(po)

data:
[['' 'Col 1' 'Col 2']
 ['Fila 1' '11' '22']
 ['Fila 2' '33' '44']]
po:
      Col 1  Col 2
Fila 1    11    22
Fila 2    33    44
```

In [9]:

```
serie = pd.Series({ "patatas" : "fritas",
                    "patatas":"con leche",
                    "pimien":"patatas"})
print('serie:')
print(serie)
print(serie[0][:])

serie:
patatas    con leche
pimien     patatas
dtype: object
con leche
```

In [10]:

```
print(serie)
print('Altura de la serie:')
print(len(serie.index))

patatas    con leche
pimien     patatas
dtype: object
Altura de la serie:
2
```

In [11]:

```
dataframe = np.array([[ ' ',0,1,2],[0,1,2,3],[1,4,5,6],[2,7,8,9]])  
  
po_df=pd.DataFrame(data=dataframe[1:,1:],index=dataframe[1:,0],columns=dataframe[0,1:])  
  
print('dataframe:')  
print(dataframe)  
print()  
print('po_df:')  
print(po_df)  
print()  
print('shape:')  
print(po_df.shape)  
print()  
  
dataframe:  
[[ ' ' '0' '1' '2']  
 ['0' '1' '2' '3']  
 ['1' '4' '5' '6']  
 ['2' '7' '8' '9']]  
  
po_df:  
   0  1  2  
0  1  2  3  
1  4  5  6  
2  7  8  9  
  
shape:  
(3, 3)
```

In [12]:

```
print('Describe: ')
print(po_df.describe())
print()
print('Mean: ')
print(po_df.mean())
print()
print('Corr: ')
print(po_df.corr())
print()
print('Count: ')
print(po_df.count())#valores no nulos de un dataframe
print()
print('Max: ')
print(po_df.max())
print()
print('Min: ')
print(po_df.min())
print()
print('Median: ')
print(po_df.median())
print()
print('Std: ')
print(po_df.std())
print()
```

Describe:

	0	1	2
count	3	3	3
unique	3	3	3
top	4	8	9
freq	1	1	1

Mean:

0	49.0
1	86.0
2	123.0

dtype: float64

Corr:

Empty DataFrame

Columns: []

Index: []

Count:

0	3
1	3
2	3

dtype: int64

Max:

0	7.0
1	8.0
2	9.0

dtype: float64

Min:

0	1.0
1	2.0
2	3.0

dtype: float64

Median:

0	4.0
1	5.0
2	6.0

dtype: float64

Std:

Series([], dtype: float64)

In [13]:

```
print('po_df:')
print(po_df)
print()
print('Segunda columna:')
print(po_df.iloc[:,1])
print()
print('Segunda fila:')
print(po_df.iloc[1,:])
print()
```

```
po_df:
   0   1   2
0   1   2   3
1   4   5   6
2   7   8   9
```

Segunda columna:

```
0    2
1    5
2    8
Name: 1, dtype: object
```

Segunda fila:

```
0    4
1    5
2    6
Name: 1, dtype: object
```

Para importar los datos

```
df = pd.read_csv(r'...Ruta hasta el csv ... \algo.csv')
( pd.to_tipoarchivo(nombre_archivo) )
```

para ver los missing values

```
pd.isnull() #matriz booleana, False si el valor sí que está y True si es un missing value
pd.isnull().sum() #para ver cuantos missing values hay
```

si queremos borrar las filas ó columnas con missing values

```
pd.dropna() #filas
df.dropna(axis = 1) #columnas
```

si queremos reemplazar en las filas ó columnas los missing values

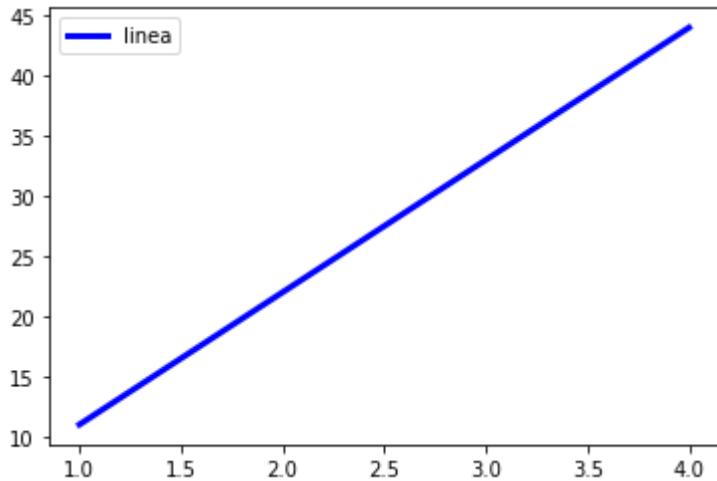
```
df.fillna(x) #se llenan con 'x'
df.fillna(df.mean()) #se llenan con el valor medio
```

In [14]:

```
# Librería Matplotlib
# Librería de trazado con muchos valores predeterminados

import matplotlib.pyplot as plt
a = [1, 2, 3, 4]
b = [11, 22, 33, 44]

plt.plot(a, b, color='blue', linewidth=3, label='linea')
plt.legend()
plt.show()
```



In [15]:

```
#Ejemplo con gráficos de Líneas
```

```
#Definir los datos
```

```
x1 = [3, 4, 5, 6]  
y1 = [5, 6, 3, 4]  
x2 = [2, 5, 8]  
y2 = [3, 4, 3]
```

```
#Configurar el gráfico
```

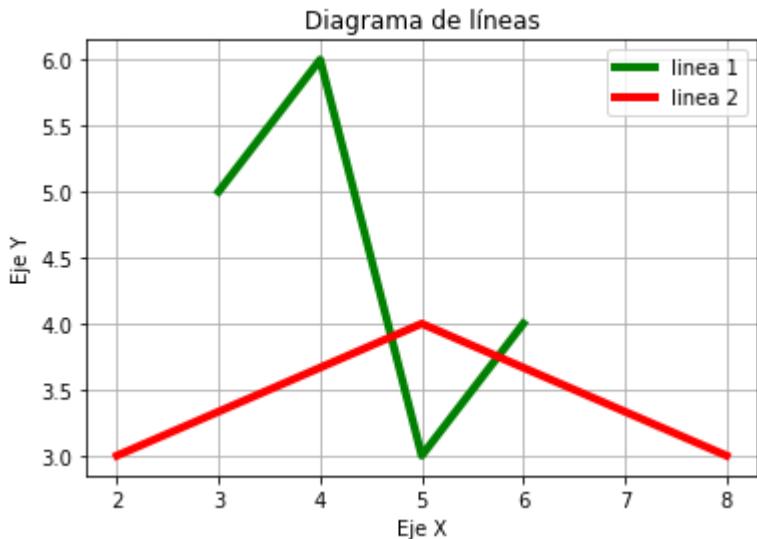
```
plt.plot(x1, y1, label='linea 1', linewidth=4, color='green')  
plt.plot(x2, y2, label='linea 2', linewidth=4, color='red')
```

```
#Título
```

```
plt.title('Diagrama de líneas')  
plt.ylabel('Eje Y')  
plt.xlabel('Eje X')
```

```
#Mostrar leyenda, cuadrícula y figura
```

```
plt.legend()  
plt.grid()  
plt.show()
```



In [16]:

```
#Ejemplo con gráficos de barras
```

```
#Definimos los datos
```

```
x1 = [0.25, 1.25, 2.25, 3.25, 4.25]  
y1 = [10, 55, 80, 32, 40]  
x2 = [0.75, 1.75, 2.75, 3.75, 4.75]  
y2 = [42, 26, 10, 29, 66]
```

```
#Configurar las características del gráfico
```

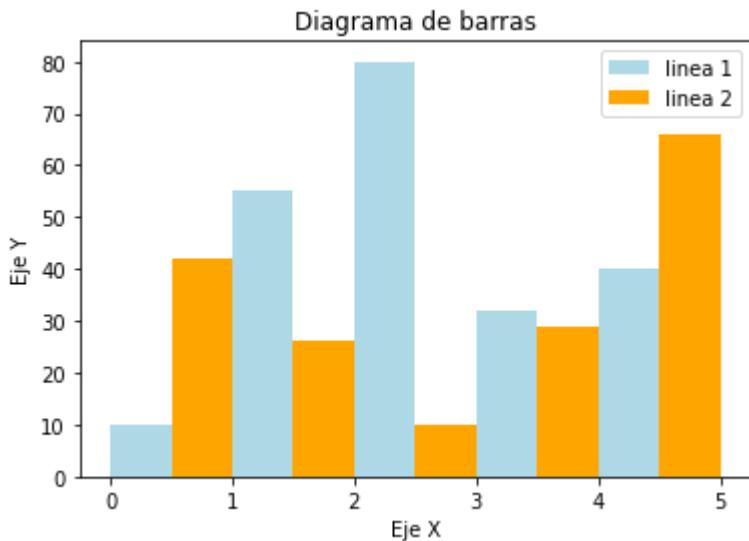
```
plt.bar(x1, y1, label='linea 1', width=0.5, color='lightblue')  
plt.bar(x2, y2, label='linea 2', width=0.5, color='orange')
```

```
#Título
```

```
plt.title('Diagrama de barras')  
plt.ylabel('Eje Y')  
plt.xlabel('Eje X')
```

```
#Mostrar leyenda, cuadrícula y figura
```

```
plt.legend()  
#plt.grid()  
plt.show()
```



In [17]:

```
#Ejemplo con gráficos de histogramas
```

```
#Definimos los datos
```

```
a = [22, 55, 62, 45, 21, 22, 34, 42, 42, 4, 2, 102, 95, 85, 55, 110, 120, 70, 65, 55, 11, 115, 80, 75, 65, 54, 44, 43, 42, 48]  
bins = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

```
#Configurar las características del gráfico
```

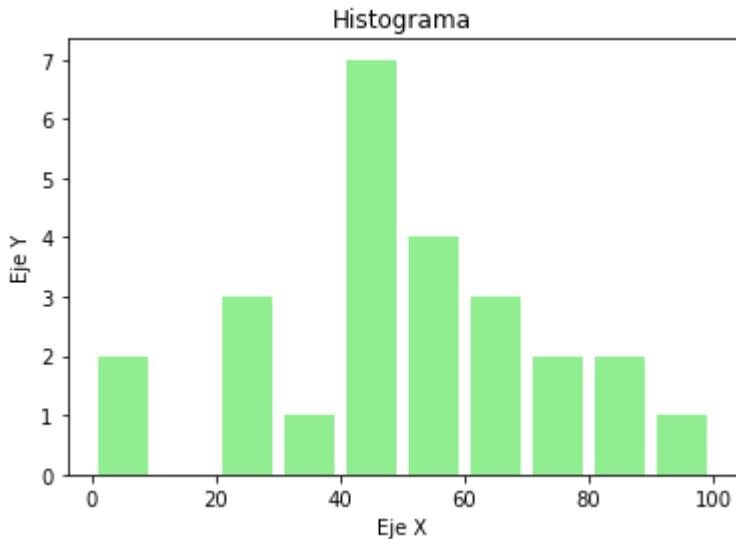
```
plt.hist(a, bins, histtype = 'bar', rwidth = 0.8, color = 'lightgreen')
```

```
#Título
```

```
plt.title('Histograma')  
plt.ylabel('Eje Y')  
plt.xlabel('Eje X')
```

```
#Mostrar Leyenda, cuadrícula y figura
```

```
#plt.Legend()  
#plt.grid()  
plt.show()
```



Preprocesamiento de datos

El preprocesamiento o análisis de datos; cómo preparar los datos para luego poder usar los datos en los algoritmos de Machine Learning.

Los datos están en todas partes, nos ayudan a responder preguntas, descubrir información útil y aplicando algoritmos de ML, a poder hacer predicciones.

In [18]:

```

import pandas as pd

# Importamos el dataframe, (diferentes formas posibles):
# pd.read_csv('ruta')
# pd.read_excel('ruta')
# pd.read_json('ruta')
# pd.read_sql('ruta')
df = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\train.csv')

# head(n) presenta las n primeras filas de la base de datos
# tail(n) presenta las n últimas filas de la base de datos
df.head(10)
#df.tail(10)

# Para ver las columnas
#df.columns
print('Columnas:', df.columns)

# Se podría modificar el archivo y podríamos guardarla en nuestro ordenador, es decir,
# crear un .csv nosotros desde aquí
# pd.to_csv('ruta')
# pd.to_excel('ruta')
# pd.to_json('ruta')
# pd.to_sql('ruta')

```

Columnas: Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
 dtype='object')

Exploración de los datos usando python

Señala tipo de datos incorrectos, nos muestra la variedad de datos. Tenemos varios tipos de datos como los numéricos, que a su vez se dividen en 'int' y 'float', también tenemos los categóricos 'object', y finalmente 'datetime' que nos da una fecha y hora.

Pandas da automáticamente un tipo de datos al leer esos datos ya los interpreta y les asigna un tipo basándose en la codificación. Se puede ver en 'Age', al igual que todos los demás, lo clasifica de tipo 'object' cuando este debería de ser un 'int'. Por lo que se debería de hacer un cambio en los datos.

Siempre al principio del análisis de datos comprobaremos los tipos de las variables para evitar complicaciones futuras y además ya que algunas funciones de Machine Learning solo se aplican a datos numéricos.

In [19]:

```
df.dtypes

# Podemos ver PassengerID, Survived, Pclass, Name, Sex, Age, SibSp (Hermanos), Parch, Ticket, Fare(Precio ticket),
# Cabin, Embarked

# Todas las columnas numéricas int64, float64 lo podemos utilizar tal cual en los algoritmos, pero en las variables
# de tipo 'object' veremos tenemos también información útil que desde luego no queremos perder
```

Out[19]:

```
PassengerId      int64
Survived         int64
Pclass           int64
Name             object
Sex              object
Age              float64
SibSp            int64
Parch            int64
Ticket           object
Fare             float64
Cabin            object
Embarked         object
dtype: object
```

In [20]:

```
### Análisis estadístico para ver si tenemos valores atípicos o missing values
df.describe()
# Podemos ver se ha omitido algunas de las columnas tipo objeto ya que solo analiza las
# numéricas, si queremos se incluyan todas

df.describe(include = "all")
# Ahora vemos tenemos todos ellos y con nuevas filas, tenemos ahora:
# 'unique', para los únicos en la columna
# 'top', para el dato más frecuente que se produce
# 'freq', cantidad de veces que aparece el object top en la columna

#NaN Not a Number, esto se debe a que esta métrica en concreto no se puede aplicar a los
# datos de ese tipo de esa columna
```

Out[20]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
count	891.000000	891.000000	891.000000	891	891	714.000000	891.000000
unique	Nan	Nan	Nan	891	2	Nan	Nan
top	Nan	Nan	Nan	Perreault, Miss. Anne	male	Nan	Nan
freq	Nan	Nan	Nan	1	577	Nan	Nan
mean	446.000000	0.383838	2.308642	Nan	Nan	29.699118	0.523008
std	257.353842	0.486592	0.836071	Nan	Nan	14.526497	1.102743
min	1.000000	0.000000	1.000000	Nan	Nan	0.420000	0.000000
25%	223.500000	0.000000	2.000000	Nan	Nan	20.125000	0.000000
50%	446.000000	0.000000	3.000000	Nan	Nan	28.000000	0.000000
75%	668.500000	1.000000	3.000000	Nan	Nan	38.000000	1.000000
max	891.000000	1.000000	3.000000	Nan	Nan	80.000000	6.0



In [21]:

```
df.info()
# Para ver las dimensiones y otras cosas

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   PassengerId 891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object  
 4   Sex          891 non-null    object  
 5   Age          714 non-null    float64 
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object  
 9   Fare          891 non-null    float64 
 10  Cabin        204 non-null    object  
 11  Embarked     889 non-null    object  
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Manipulando missing values de nuestro dataframe

Pueden aparecer estos missing values como: NaN (más frecuente), ?, N/A, 0, espacio_en_blanco

Existen muchas maneras de corregir estos valores según el programador y el problema en sí. Las formas de arreglar estos problemas son:

- Verificar con la persona que recogió los datos si ese es un missing value o no

- Eliminar los datos, ya sea el atributo o la fila, en caso de ser fila es una opción conveniente en el caso de no tener muchos datos faltantes

- Reemplazar los datos, calculando el valor promedio de la variable completa, una mejor opción porque no se pierde los datos, pero es menos preciso ya que estamos asumiendo un valor que puede ser cierto y puede no serlo. Además, pueden ser variables categóricas las perdidas y por ello no podemos usar el valor medio sino que tendremos que sustituir por el más frecuente.

- Dejar los datos faltantes así como datos perdidos, en ocasiones puede ser una buena opción.

Una vez vistas las posibles filosofías o soluciones, vamos a ver diferentes opciones o soluciones

Para el caso de querer eliminar datos

dropna(), con axis = 0 si se quiere eliminar una/s fila/s y axis = 1 si se quiere eliminar una/s columna/s

Tenemos que eliminar las filas de todos los datos perdidos

df.dropna(axis = 0)

Podemos ver se han eliminado las filas con missing values

df.dropna(subset = ["Age"], axis = 0)

df

Se eliminan las filas con missing values pero solo de la columna "Age"

df.dropna(axis = 1)

Podemos ver se han eliminado las columnas enteras con missing values. No recomendable, perdemos muchos datos

En caso de querer borrar esas columnas de verdad y que se implemente esa modificación en el propio dataframe es poner

df.dropna(subset = ["Age"], axis = 0, inplace = True)

Para reemplazar valores con valores reales

`replace(data a reemplazar, nueva data)`

Una forma muy sencilla de eliminar esto, hay formas más complejas

In [22]:

```
import numpy as np

# Calculamos la edad promedio
df["Age"].mean()

# Ahora vamos a reemplazar los NaN por el valor medio
df["Age"].replace(np.nan, df["Age"].mean())
```

Out[22]:

```
0      22.000000
1      38.000000
2      26.000000
3      35.000000
4      35.000000
...
886    27.000000
887    19.000000
888    29.699118
889    26.000000
890    32.000000
Name: Age, Length: 891, dtype: float64
```

In [23]:

df

Out[23]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.25
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	38.0	1	0	PC 17599	71.28
2	3	1	3	Allen, Mr. William Henry	male	35.0	0	0	STON/O2. 3101282	7.92
3	4	1	1	Montvila, Rev. Juozas Graham, Miss. Margaret Edith	female	27.0	0	0	113803	53.10
4	5	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.45
886	887	0	2	Behr, Mr. Karl Howell	male	26.0	0	0	211536	30.00
887	888	1	1	Dooley, Mr. Patrick	male	32.0	0	0	112053	7.75
888	889	0	3							
889	890	1	1							
890	891	0	3							

891 rows × 12 columns

Cómo cambiar datos de variables categóricas a numéricas

Como ya lo hemos mencionado, los algoritmos no pueden presentar ni como entrada ni para el entrenamiento variables categóricas o cadenas.

Tenemos muchas variables categóricas "Sex" que va a ser binaria en este dataframe ('male' ó 'female')

```
pd.get_dummies(df, columns = ["Sex"])
```

Lo que hace es que elimina la columna "Sex" y nos va a devolver 2 columnas, una por cada categoría o tipo de este atributo concreto. Y podemos ver la info es redundante ya que tenemos ahora 2 columnas, "Sex_female", "Sex_male" en caso de 1 único tipo de información, es decir, con una variable llamada podríamos ver 1 para hombres y 0 para mujeres (o viceversa) y sería una única columna, pero hay una función que ya hace eso

```
pd.get_dummies(df, columns = ["Sex"], drop_first = True)
```

Dando como resultado solo "Sex_male", podríamos ver 1 para hombres y 0 para mujeres

In [24]:

```
pd.get_dummies(df, columns = ["Sex"], drop_first = True)
```

Out[24]:

	PassengerId	Survived	Pclass	Name	Age	SibSp	Parch	Ticket	Fare	Cab
0	1	0	3	Braund, Mr. Owen Harris	22.0	1	0	A/5 21171	7.2500	Nan
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...)	38.0	1	0	PC 17599	71.2833	C123
2	3	1	3	Heikkinen, Miss. Laina	26.0	0	0	STON/O2. 3101282	7.9250	Nan
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	35.0	0	0	373450	8.0500	Nan
...
886	887	0	2	Montvila, Rev. Juozas	27.0	0	0	211536	13.0000	Nan
887	888	1	1	Graham, Miss. Margaret Edith	19.0	0	0	112053	30.0000	B-1
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	NaN	1	2	W./C. 6607	23.4500	Nan
889	890	1	1	Behr, Mr. Karl Howell	26.0	0	0	111369	30.0000	C123
890	891	0	3	Doolley, Mr. Patrick	32.0	0	0	370376	7.7500	Nan

891 rows × 12 columns



Agrupamiento de Datos o Binning

Agrupamiento de los datos en compartimentos y podemos ver puede mejorar la comprensión de la distribución de los datos y mejorar la precisión de los algoritmos de ML.

Se puede hacer con pandas con la función:

```
pd.cut
```

Podemos separar la gente en grupos de, por ejemplo [0,5], [6,12], [13,18], [19,35], [36,60],[61,100], rangos al azar y se pueden cambiar en función de nuestros análisis. Ahora creamos una variable con solo los puntos de comienzo del primer rango y donde finalizan todos los del resto de estos rangos

```
bins = [0, 5, 12, 18, 35, 60, 100]
```

Nombre de rango de estas edades, podríamos poner 'Baby', 'Child', ... pero son categóricas y eso no nos gusta

```
names = ["1", "2", "3", "4", "5", "6", "7"]
```

```
df["Age"] = pd.cut(df["Age"], bins, labels = names)
```

In [25]:

```
age_bins = [0, 5, 12, 18, 35, 60, 100]
#age_bins = [0, 15, 30, 45, 60, 75, 90]
names = ["1", "2", "3", "4", "5", "6"]

df[ "Age" ] = pd.cut(df[ "Age" ], age_bins, labels = names)
```

In [26]:

df

Out[26]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	4	1	0	A/5 21171	7.25
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...)	female	5	1	0	PC 17599	71.28
2	3	1	3	Heikkinen, Miss. Laina	female	4	0	0	STON/O2. 3101282	7.92
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	4	1	0	113803	53.10
4	5	0	3	Allen, Mr. William Henry	male	4	0	0	373450	8.05
...
886	887	0	2	Montvila, Rev. Juozas	male	4	0	0	211536	13.00
887	888	1	1	Graham, Miss. Margaret Edith	female	4	0	0	112053	30.00
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.45
889	890	1	1	Behr, Mr. Karl Howell	male	4	0	0	111369	30.00
890	891	0	3	Doolley, Mr. Patrick	male	4	0	0	370376	7.75

891 rows × 12 columns



Todos los comandos resumidos

```
titanic_df = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\train.csv')
```

titanic_df = pd.DataFrame.copy(titanic_df)

```
titanic_df.head(10)
```

```
titanic_df.dtypes
```

Análisis estadístico para ver si tenemos valores atípicos o missing values

```
titanic_df.describe()
```

Podemos ver se ha omitido algunas de las columnas tipo objeto ya que solo analiza las numéricas, si queremos se incluyan todas

```
titanic_df.describe(include = "all")
```

```
titanic_df["Age"].mean()
```

```
titanic_df["Age"].replace(np.nan, titanic_df["Age"].mean())
```

```
pd.get_dummies(titanic_df, columns = ["Sex"], drop_first = True)
```

```
titanic_df
```

```
titanic_df.drop(['Name', 'Ticket','Cabin','Sex','Embarked'], 1, inplace = True)
```

En verdad debería de haber convertido 'Cabin','Sex','Embarked' a dummy, estoy perdiendo info :c

```
titanic_df
```

```
print(titanic_df.describe())
```

```
data = np.array(titanic_df.drop(['Survived'], 1).astype(float)) target = np.array(titanic_df['Survived'])
```

```
data
```

```
scaled_data = preprocessing.scale(data) scaled_target = preprocessing.scale(target)
```

```
scaled_data
```

Cross Validation

En un problema de Clasificación a menudo tendremos la pregunta de cuál de todos los algoritmos utilizar, para ello tenemos el Cross Validation o Validación Cruzada, que nos permite comparar los diferentes algoritmos en nuestro problema actual y poder ver cuál de ellos es más predictivo.

Lo primero a hacer con nuestros datos va a ser estimar los parámetros para los diferentes métodos de ML que tenemos, por ejemplo, para la Regresión Logística vamos a ver podemos utilizar un porcentaje de los datos para poder ajustar la curva, esos datos van a ser los datos 'train'. El segundo paso sería evaluar cómo de bien han funcionado estos métodos de clasificación, es decir, averiguar cómo de buen predictor va a ser para nuevos datos, los datos 'test'.

Ahora sí, el asunto es que usamos un 75% train, 25% test, pero, ¿qué pasaría si los datos del test fuesen diferentes en una segunda vez? En vez de preocuparnos por cuáles de los datos deberíamos de pasar como train o como test, Cross Validation ya lo hace, los hace todos y evalúa el mejor y se queda con ese.

En un caso extremo se usa un Leave-One-Out-Cross-Validation, pero hay que hacerlo N veces. En la práctica lo común suele ser hacer la elección de 10 bloques Ten-Fold-Cross-Validation

Reducción de Dimensionalidad

PCA

Principal Component Analysis (PCA)

Si queremos hacer un análisis de los datos tenemos que ver plots de una variable independiente frente a otra para ver posibles correlaciones, pero claro, si tenemos x_i con $i = 1, \dots, N$ con N grande, no lo podemos hacer tan fácilmente. ¿Qué podemos hacer? Tanto dibujar todos x_i frente x_j es demasiados gráficos y no vamos a entender nada, rotar los gráficos anteriores va a ser demasiado jaleo...

Lo mejor es PCA, que convierte las correlaciones (o falta de correlaciones) de x_i con x_j , en todas las características en un gráfico 2D, entonces veremos un mapa 2D de Clusters y ellos serán los elementos que tengan características altamente correlacionadas entre sí, y una vez hecho esto podremos ver relaciones entre ellos.

Un aspecto importante a la hora de interpretar PCA-plots es que tendremos que evaluar los ejes que están relacionados con la importancia, y además el eje horizontal PC1 (Principal Component axis 1) es más importante que el eje vertical PC2 (Principal Component axis 2). Si tenemos 3 clusters (A, B, C), A cerca del origen y tiene a una distancia dada en la horizontal el Cluster B, y de nuevo A tiene a la misma distancia pero en la vertical el Cluster C, podemos ver A y B serán más diferentes entre sí que A y C.

Hay otras formas de ver algoritmos de reducción de la dimensionalidad, como bien pueden ser 'Heatmaps', 't-SNE Plots' y 'Multi-Dimensional Scaling (MDS)'.

In [27]:

```
##### Principal Component Analysis (PCA)
import numpy as np
import pandas as pd
import random as rd
from sklearn.decomposition import PCA
from sklearn import preprocessing
import matplotlib.pyplot as plt

# Vamos a generar un dataset aleatorio

# Lo primero vamos a ver un array con 100 nombres de genes

genes = ['gene'+str(i) for i in range(1,101)]
genes

# Ahora vamos a generar arrays de sample names 5 de "wt" (wild type) y 5 de "ko" (knock out)
wt = ['wt'+str(i) for i in range(1,6)]
ko = ['ko'+str(i) for i in range(1,6)]

# Creamos el dataset
data = pd.DataFrame(columns=[*wt, *ko], index = genes)
#data = pd.DataFrame(columns=[wt1, wt2, wt3, wt4, wt5, ko1, ko2, ko3, ko4, ko5, ko6], index = genes) # forma equivalente
data
# de resultado tenemos un dataset con 10 columnas (5 columnas para los wt y las otras 5 para ko) y 100 filas (genes)
for gene in data.index:
    data.loc[gene, 'wt1':'wt5'] = np.random.poisson(lam = rd.randrange(10,1000), size = 5)
    data.loc[gene, 'ko1':'ko5'] = np.random.poisson(lam = rd.randrange(10,1000), size = 5)
data
# Se generan números aleatorios acordes a una distribución Poissoniana y además en diferentes poissonianas

# Antes de PCA siempre tenemos que escalar y centrar los datos
scaled_data = preprocessing.scale(data.T)
# ó equivalentemente
# StandardScaler().fit_transform(data.T)

# Le estamos pasando el data traspuesto para que las columnas sean los genes y podemos hacer así el escalado a esas columnas
# de genes que es lo que nos importa escalar y centrar
# Tras esto los datos tendrán media 0 y varianza 1
scaled_data

# Definimos el algoritmo
pca = PCA()

pca.fit(scaled_data)

pca_data = pca.transform(scaled_data)

# Y vamos a ver cómo ha quedado

# Lo primero es calcular el porcentaje de variación de cada componente principal
per_var = np.round(pca.explained_variance_ratio_* 100, decimals = 1)
labels = ['PC'+str(x) for x in range(1,len(per_var)+1)] # Una etiqueta por cada componente
```

Introducción principal

```
plt.bar(x = range(1,len(per_var)+1), height = per_var, tick_label = labels)
plt.ylabel('Percentage of Explained Variance')
plt.xlabel('Principal Component')
plt.title('Scree Plot')
plt.show()
# Se puede ver tenemos casi toda la varianza explicada en PC1 y PC2, por lo que podemos hacer un gráfico 2D.

# Tenemos que crear nuestro dataframe
pca_df = pd.DataFrame(pca_data, index = [*wt, *ko], columns = labels )
pca_df

plt.scatter(pca_df.PC1, pca_df.PC2)
plt.title('My PCA graph')
plt.xlabel('PCA1 -{0}%'.format(per_var[0]))
plt.ylabel('PCA2 -{0}%'.format(per_var[1]))

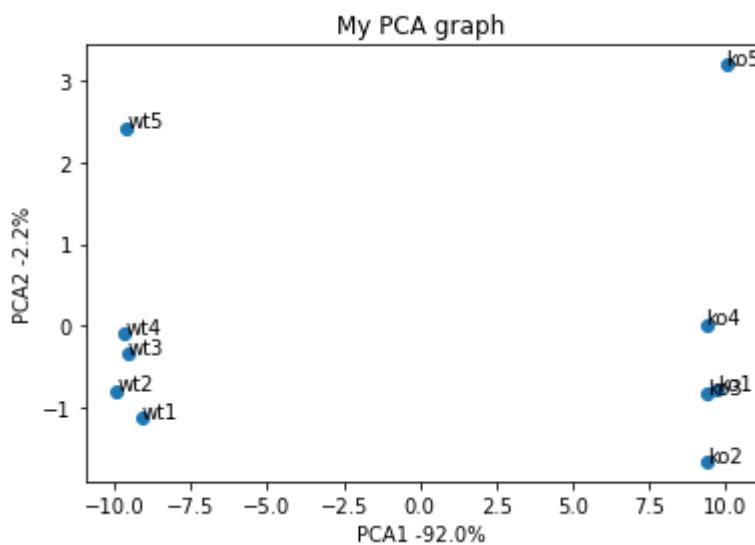
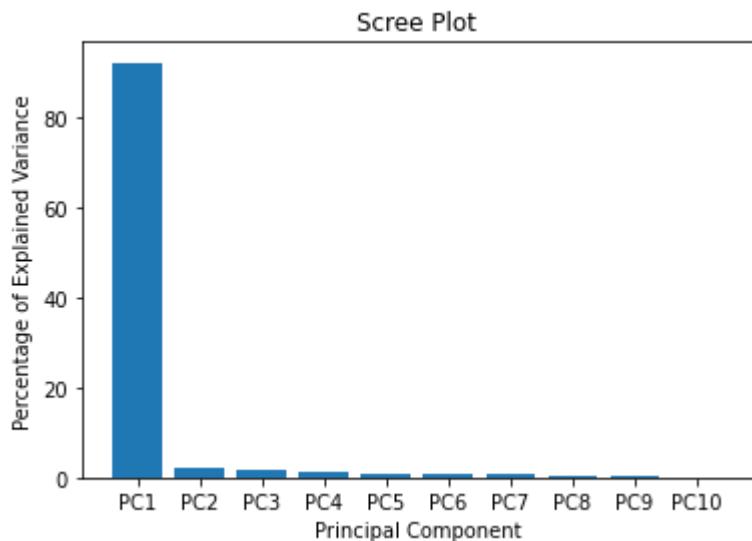
for sample in pca_df.index:
    plt.annotate(sample, (pca_df.PC1.loc[sample], pca_df.PC2.loc[sample]))
plt.show()
# Ahora podemos ver los ko's están separados de los wt's

# Finalmente para verlo de forma cuantitativa se puede ver los 'Loading scores' de PC1 para ver la separación entre ellas
loading_scores = pd.Series(pca.components_[0], index = genes)
# El pca.components_[0] con 0 para que sea PC1

sorted_loading_scores = loading_scores.abs().sort_values(ascending = False)

top10genes = sorted_loading_scores[0:10].index.values

print(loading_scores[top10genes])
```



```
gene65      0.104234
gene88     -0.104230
gene19      0.104198
gene100     0.104191
gene8       0.104176
gene6       0.104175
gene95      0.104160
gene96      0.104155
gene49     -0.104152
gene40     -0.104145
dtype: float64
```

LDA

Linear Discriminant Analysis (LDA)

Igual que con PCA, recapitulando, si queremos hacer un análisis de los datos tenemos que ver plots de una variable independiente frente a otra para ver posibles correlaciones, pero claro, si tenemos x_i con $i = 1, \dots, N$ con N grande, no lo podemos hacer tan fácilmente. ¿Qué podemos hacer? Tanto dibujar todos x_i frente x_j es demasiados gráficos y no vamos a entender nada, rotar los gráficos anteriores va a ser demasiado jaleo...

Básicamente estamos buscando reducir las dimensiones de un problema N-Dimensional como con PCA, lo único es que PCA nos indicaba las correlaciones entre ellos y también la importancia basándose en el porcentaje de varianza explicado, es decir, más correlaciones.

Ahora no buscamos eso, lo que queremos es maximizar la separabilidad en la clasificación de dos grupos conocidos para la toma de decisiones. LDA va a intentar usar todas las características para poder representar los datos de tal forma que sean linealmente separables y se va a proyectar en un nuevo eje resultante de una combinación lineal de dichas características o variables independientes, y va a elegir aquel que sea el que maximice esa posible clasificación para separar esas 2 categorías resultantes.

Ese nuevo eje idóneo es creado de tal forma que elegimos aquel que maximice la distancia entre los valores medios de los grupos a separar, y minimiza las varianzas de cada uno de los grupos por individual.

En caso de querer aplicar LDA cuando tenemos que clasificar 3 categorías diferentes, tendremos que determinar el valor medio de todos los datos y luego el valor medio de cada uno de esos 3 grupos de categorías. Una vez hecho esto calcularemos la distancia de cada uno de estos 3 centros a ese centro total de todos los datos. Ahora tenemos que hacer lo análogo al caso de 2 categorías, encontrar el nuevo eje sobre el que proyectar cuando tenemos que maximizar esas distancias de cada categoría al centro de todas y minimizar la varianza de cada categoría por individual.

Sin embargo tenemos otra diferencia ahora, antes con 2 puntos centrales (2 categorías), se definía una línea como separador, ahora sin embargo tenemos 3 puntos centrales (3 categorías), y como 2 puntos definen una línea, pues 3 puntos definen dos líneas y por ello un plano.

In [28]:

```
##### Linear Discriminant Analysis (LDA)

from sklearn import datasets, linear_model
import numpy as np
import matplotlib.pyplot as plt

iris = datasets.load_iris()

print('iris_keys: ',iris.keys)
print('feature_names: ',iris.feature_names)

X = iris.data
y = iris.target

#plt.scatter(X, y)
#plt.show()

#from sklearn.model_selection import train_test_split

#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

#from sklearn import preprocessing
# Antes de LDA siempre tenemos que escalar y centrar los datos
#scaled_X = preprocessing.scale(X)
#scaled_y = preprocessing.scale(y)
# ó equivalentemente

#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#X = sc.fit_transform(X)
#X = sc.transform(X)

#y = sc.fit_transform(y)
#y = sc.transform(y)
#X_train = sc.fit_transform(X_train)
#X_test = sc.transform(X_test)

print('Shape of data scaled:',X.shape)
print('Shape of data target:',y.shape)
# Tras esto los datos tendrán media 0 y varianza 1
#print('X_train;',X_train)

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
# Definimos el algoritmo
lda = LDA(n_components = 2)

lda.fit(X, y)
X_proyected = lda.transform(X)
#lda.fit(X_train, y_train)
#X_proyected = lda.transform(X_test)

print('Shape of the data after being projected (applying LDA):',X_proyected.shape)

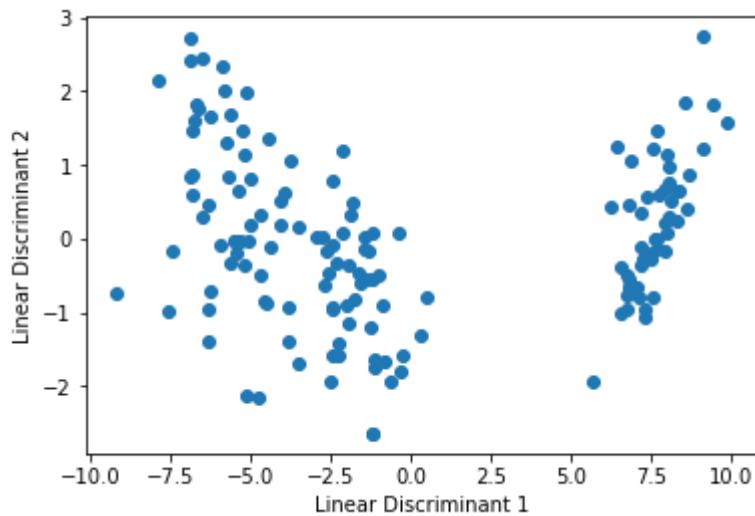
# Y vamos a ver cómo ha quedado

x_1 = X_proyected[:, 0]
x_2 = X_proyected[:, 1]

plt.scatter(x_1, x_2)
```

```
plt.xlabel('Linear Discriminant 1')
plt.ylabel('Linear Discriminant 2')
plt.show()
```

```
iris_keys: <built-in method keys of Bunch object at 0x000002266154D680>
feature_names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Shape of data scaled: (150, 4)
Shape of data target: (150,)
Shape of the data after being projected (applying LDA): (150, 2)
```



ICA

Independent Component Analysis (ICA)

Una técnica estadística para revelar los factores escondidos o la estructura interna de los datos. Es un método que asume que las variables independientes, esos datos, son mezclas lineales de ciertas variables desconocidas latentes. El proceso de mezclado es desconocido. Las variables latentes se asumen no gaussianas y estadísticamente independientes, y son las llamadas componentes independientes, que se pueden encontrar mediante ICA ó PROC ICA.

Se puede ver como ejemplo que si tenemos señales conocidas y creamos nuevas señales como combinación lineales de las señales conocidas, el propósito de PROC ICA es estimar las señales fuente o conocidas a partir de estar observando las señales nuevas.

In [29]:

```
##### Independent Component Analysis (ICA)
import numpy as np
import pandas as pd
import random as rd
from sklearn.decomposition import FastICA
from sklearn.decomposition import PCA
from sklearn import preprocessing
import matplotlib.pyplot as plt
from scipy import signal

# Generamos una muestra de datos
np.random.seed(0)
# x axis
n_samples = 2000
# y axis
time = np.linspace(0, 8, n_samples)

# Signal 1: Sinusoidal Signal
# Signal 2: Square Signal
# Signal 3: Saw tooth Signal
s1 = np.sin(2 * time)
s2 = np.sign(np.sin(3 * time))
s3 = signal.sawtooth(2 * np.pi * time)

S = np.c_[s1, s2, s3]
#print('S:',S)
# Añadimos el ruido
S += 0.2 * np.random.normal(size = S.shape)

# Estandarizamos los datos
S = S/S.std(axis = 0)

# Mezclamos los datos y creamos o inicializamos la matriz
A = np.array([[1, 1, 1], [0.5, 2.0, 1.0], [1.5, 1.0, 2.0]])
X = np.dot(S, A.T)

# Definimos el algoritmo
ica = FastICA(n_components = 3)

# Reconstruir la señal
S_ica = ica.fit_transform(X)

# Reconstruir la matriz
A_ica = ica.mixing_

# Se puede ver que el ICA funciona 'revertiendo' el mix
assert np.allclose(X, np.dot(S_ica, A_ica.T) + ica.mean_)

# Para comparar resultados podemos hacer también PCA
pca = PCA(n_components = 3)
# Reconstruir la señal basandonos en componentes ortogonales
H = pca.fit_transform(X)

# A plotear los resultados

plt.figure()

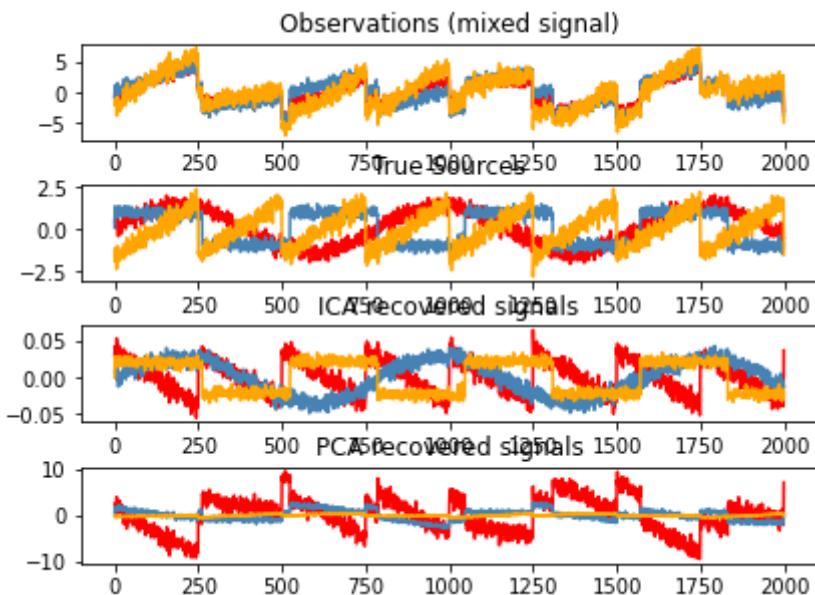
models = [X, S, S_ica, H]
names = ['Observations (mixed signal)', 'True Sources', 'ICA recovered signals', 'PCA r
```

```

    ecovered signals']
colors = ['red', 'steelblue', 'orange']
for i, (model,name) in enumerate(zip(models, names), 1):
    plt.subplot(4, 1, i)
    plt.title(name)
    for sig, color in zip(model.T, colors):
        plt.plot(sig, color = color)

plt.subplots_adjust(0.09, 0.04, 0.94, 0.94, 0.26, 0.46)
plt.show()

```



APRENDIZAJE SUPERVISADO. REGRESIÓN Y CLASIFICACIÓN

Librería Scikit - Learn

Librería de código abierto con algoritmos de aprendizaje supervisado, aprendizaje no supervisado, validación cruzada, varios conjuntos de datasets o datos, extracción y selección de características

Bias y varianza

En Machine Learning es importante la precisión aunque siempre hay errores y queremos buscar esas fuentes de error para poder comprender mejor los fallos y así poder estimar unos resultados con mayor precisión

Hay varios errores, el error de 1. varianza, 2. sesgo o bias e 3. irreducible

1. Error Irreducible o Ruido No se puede reducir ya que no se conoce su causa exacta y aparece como interferencia entre medias en términos de los datos de entrada y salida, es decir por variables desconocidas, variables incompletas o problema mal enmarcado. No importa cuán bueno sea el modelo pero este error siempre va a aparecer
2. Error de bias o sesgo Fallo entre los valores esperados o reales y los obtenidos, ya que algunos algoritmos son demasiado rígidos como para ajustarse con precisión a los modelos. Los algoritmos paramétricos (como un ajuste lineal) tienen un alto bias que los hacen rápidos de aprender y entender pero poco flexibles y con un menor rendimiento en problemas predictivos. Bajo bias supone pocas suposiciones sobre la forma del objetivo:

Árboles de decisión
k-vecinos más cercanos
Máquinas de vectores de soporte

Alto bias supone muchas suposiciones sobre la forma del objetivo

Regresión lineal
Análisis Discriminante Lineal (LDA)
Regresión Logística

1. Error de varianza Fallo por cantidad de la estimación objetivo y cómo va a fallar en función de diferentes datos de entrenamiento. Idealmente no debería de cambiar con un conjunto de datos a otro. Baja varianza supone pocos cambios de un sistema de entrenamiento a otro para obtener el mismo modelo predictivo

Regresión Lineal
Análisis Discriminante Lineal (LDA)
Regresión Logística

Alta varianza supone muchos cambios de un sistema de entrenamiento a otro para obtener el mismo modelo predictivo

Árboles de decisión
k-vecinos más cercanos
Máquinas de vectores de soporte

Compensación bias-varianza es lograr bajas varianzas y bias para que sea un buen predictor

Baja Varianza - Alto Bias : Suelen ser menos complejos y con una estructura subyacente simple o rígida (regresión lineal o el ingenuo bayes)

Bajo Bias - Alta varianza : Más complejos, con una estructura subyacente más flexible (árboles de decisión y k-vecinos más cercanos)

Aumentar Bias => Decrece Varianza Aumentar Varianza = > Decrece Bias

Buen modelo es encontrar un equilibrio óptimo entre Bias-Varianza

Overfitting and Underfitting

Al trabajar con un conjunto de datos tenemos que dividir el conjunto de datos en train y test queremos aumentar la resolución de la predicción al ver ese aumento/reducción de la selección de características de nuestro modelo. Resultados pobres => Puede ser por dos motivos

- a) Modelo demasiado simple que no predice
- b) Modelo demasiado complejo que no es capaz de expresar el objetivo

Underfitting: Alto Bias o Sesgo, modelo demasiado simple, no es capaz de predecir, significa que no se ajusta lo suficientemente bien, puede ser por pocos datos o por estar intentando usar un modelo demasiado sencillo (regresión lineal). Se debe solucionar aumentando los datos de entrada o variando las características de selección.

Overfitting: Alta Varianza, modelo demasiado complejo, modela los datos demasiado bien, ya que es un fallo porque incluye el ruido o valores demasiado altos en el conjunto de valores de entrenamiento. Tiene todo esto un impacto negativo en la predicción con datos nuevos. Imposible de generalizar. Aparece de forma más probable con modelos no paramétricos y no lineales.

Se busca un modelo intermedio, pero es complicado en los modelos de Machine Learning, algunas soluciones son

- Usar más información en el train dataset (pero a veces al tener mucho ruido mal)
- Técnica de validación cruzada; técnica de oro en ML, estimar precisión del modelo en datos no vistos, usar datos para la validación de 5 veces
- Detección temprana, al entrenar un conjunto iteradamente se podrá observar que un conjunto de datos en ML hasta un cierto número de iteraciones el modelo mejora y con alta predicción, pero a partir de un cierto número veremos este decrecimiento en calidad esa predicción.
- Regularización; consiste en simplificar el modelo de ML, consiste en muchos métodos, según el modelo, si es un árbol de decisión, la regularización es podarlos. Si el modelo es una regresión se puede agregar una función de costo para la regularización.

Error en modelos de regresión

Ver los errores en modelos de regresión para poder determinar cómo de buena ha sido una predicción tenemos que ver la desviación de los valores frente al valor esperado. Por ello vamos a usar las siguientes magnitudes

- Error cuadrático medio: Indica cuán cerca están los datos predichos de los reales
- Error absoluto medio
- R^2 : entre 0 y 1, el 1 es lo mejor

Aprendizaje Supervisado. Algoritmos de Regresión

Regresión Lineal

De naturaleza paramétrica para poder aproximar una variable de forma paramétrica como $y=ax+b$ con los parámetros a, b . Entonces tenemos que determinar esos parámetros para conseguir que más se ajusten y minimice la distancia vertical entre todos los puntos a la línea.

La forma es hacerlo intentando reducir el error por medio de mínimos cuadrados, intentando buscar los parámetros con menores distancias a esta recta. Funciona debido a que tiene buenas propiedades matemáticas por lo que se puede usar bien el criterio de descenso por gradiente y computacionalmente es más rápido por ser sencillo y también es rápido de analizar aún a grandes cantidades de datos.

Ejemplo: 100 datos 2 variables: peso y altura

Como tenemos dos variables se puede ver una en función de la otra y tenemos solo una incógnita $y(x)$ y por ello podemos usar regresión lineal. $y = \text{peso}$, $x = \text{altura}$.

Suposiciones, existe una variable lineal e independiente aditiva entre las variables dependientes e independientes. Si las variables independientes están correlacionadas conduce a multicolinealidad, por lo que el modelo no puede predecir los parámetros

Los valores de error deben poseer varianza constante y no deben correlacionarse, esas correlaciones en términos de error es autocorrelación y eso introduce mucho error.

La relación entre variables dependientes y los términos de error deben tener una distribución normal.

Cómo implementar el algoritmo de Rregresión Lineal usando Scikit-Learn

```
from sklearn import linear_model  
  
x_train = variables_Independientes_train  
  
y_train = variables_Dependientes_train  
  
x_test = variables_Independientes_test  
  
y_test = variables_Dependientes_test  
  
algoritmo = linear_model.LinearRegression()  
  
algoritmo.fit(x_train, y_train)  
  
algoritmo.predict(x_test)
```

Ó

```
linear_model.LinearRegression().fit(x_train, y_train)  
  
linear_model.LinearRegression().predict(x_test)
```

Y lo que realmente estamos haciendo es el ajuste como $y = a_1x_1 + a_2x_2 + \dots + a_nx_n + b$ y si lo que queremos obtener es a y b tenemos que ver

```
a = algoritmo.coef_ # tantos valores de a a igualar como queramos ver  
  
a1, a2, a3 = algoritmo.coef_  
  
b = algoritmo.intercept_
```

Ó

```
a = linearmodel.LinearRegression().coef_  
  
b = linearmodel.LinearRegression().intercept
```

Y si ahora queremos ver la precisión de nuestro algoritmo

```
algoritmo.score(x_test, y_test)  
  
linear_model.LinearRegression().score(x_test, y_test)
```

In [30]:

```
### Linear Regression (Práctica)

from sklearn import datasets, linear_model
import numpy as np
import matplotlib.pyplot as plt

boston = datasets.load_boston()
#print(boston)

#print(boston.keys)

#print(boston.DESCR)
#Vemos hay 506 filas, 14 columnas o atributos y la 14 es el valor medio o target. Luego
#nos explican todos los datos de las
#columnas.
#Nos dicen no hay ningún missing value

#print(boston.data.shape)
#Nos da (506, 13), lo cual ya sabíamos

print(boston.feature_names)
#Para nuestro análisis solo vamos a usar las columnas que llevan como nombre RM, o se podría usar cualquiera de ellas

#como los datos están acumulados en numpy hay que llamarlos como tal
rooms = boston.data[:, np.newaxis, 5]
rooms

target = boston.target
target

#Para ver la distribución de los datos vamos a usar una gráfica tipo Scatter
plt.scatter(rooms, target)
plt.xlabel('Número de habitaciones')
plt.ylabel('Valor medio')
plt.show()

#Tenemos que separar nuestro modelo en train y test

from sklearn.model_selection import train_test_split

rooms_train, rooms_test, target_train, target_test = train_test_split(rooms, target, test_size = 0.2)

#definir nuestro modelo
algoritmo = linear_model.LinearRegression()

#entrenar el modelo
algoritmo.fit(rooms_train, target_train)

#realizar la predicción con nuestro modelo
target_predicted = algoritmo.predict(rooms_test)

#vamos a verlo gráficamente
a = algoritmo.coef_
b = algoritmo.intercept_
print('Modelo : \n y=' ,a,'x+',b)

plt.scatter(rooms, target)
```

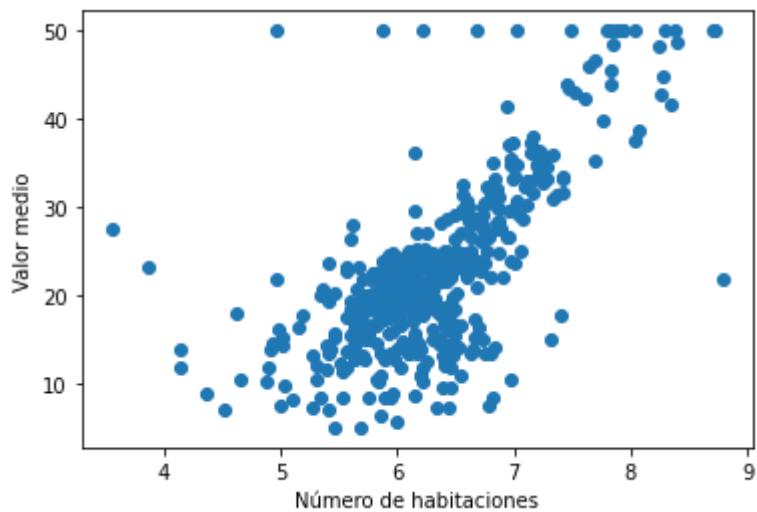
```

plt.xlabel('Número de habitaciones')
plt.ylabel('Valor medio')
plt.plot(rooms_test, target_predicted, color = 'red', linewidth = 2)
plt.show

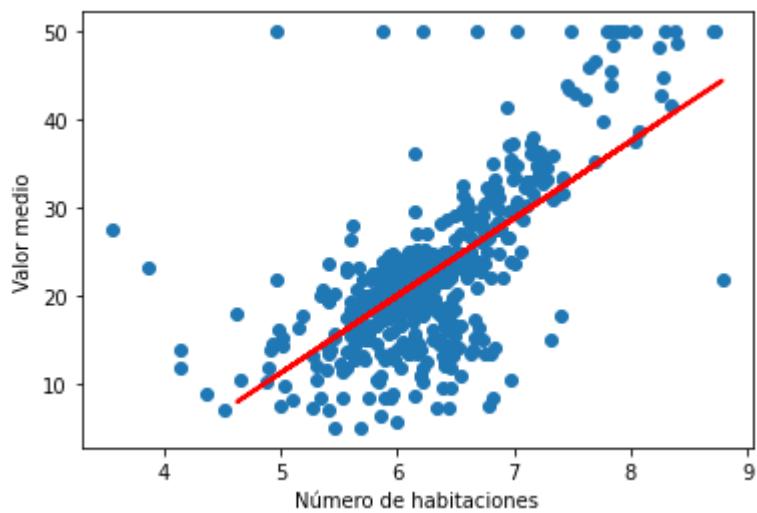
#Precisión del modelo
print('Precisión: ', algoritmo.score(rooms_test, target_test))
# Como es un 0.54 vemos la precisión no es muy buena, no es que el algoritmo sea malo sino que igual no es el más aplicable
# para este tipo de problemas

```

['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']



Modelo :
 $y= [8.78601941] x+ -32.726137703901315$
 Precisión: 0.498421532809463



Regresión Lineal Múltiple

Por lo general una variable dependiente depende de múltiples variables independientes, múltiples coeficientes por ello y también por ello aumenta la complejidad del algoritmo.

$$y = a_1x_1 + a_2x_2 + \dots + a_nx_n + b$$

Ahora con n variables independientes. Por lo general se empieza incluyendo las mejores variables independientes, para ello construir una matriz de correlación de las variables independientes (x_i) frente a la variable dependiente (y). Así podremos ver qué variables son las más significativas y así minimizamos esa función de error. Tenemos que detenernos cuando no hay mejora al incluir la siguiente variable independiente en el modelo a la hora de ver la función de predicción.

Agregar más variables independientes no mejora la predicción (overfitting). Al añadir las variables independientes puede ser que estén correlacionadas entre sí (multicolinealidad). El escenario óptimo es ver las variables independientes no correlacionadas entre sí pero sí correlacionadas con la dependiente.

In [31]:

```
### Multiple Linear Regression (Práctica)

from sklearn import datasets, linear_model
import numpy as np
import matplotlib.pyplot as plt

boston = datasets.load_boston()
#print(boston)

print('boston_keys: ',boston.keys)

#print(boston.DESCR)
#Vemos hay 506 filas, 14 columnas o atributos y la 14 es el valor medio o target. Luego
nos explican todos los datos de las
#columnas.
#Nos dicen no hay ningún missing value por lo que no hay que realizar mucho preprocesamiento

#print(boston.data.shape)
#Nos da (506, 13), lo cual ya sabíamos

print('feature_names: ',boston.feature_names)

multiple_data = boston.data[:, 5:8] # realmente son las columnas 5, 6, y 7
multiple_data

multiple_target = boston.target

from sklearn.model_selection import train_test_split

#usamos un 20% de los datos como test
data_train, data_test, target_train, target_test = train_test_split(multiple_data, multiple_target, test_size=0.2)

#definir nuestro modelo
algoritmo = linear_model.LinearRegression()

#entrenar el modelo
algoritmo.fit(data_train, target_train)

#realizar la predicción con nuestro modelo
target_predicted = algoritmo.predict(data_test)

a1, a2, a3 = algoritmo.coef_
b = algoritmo.intercept_

print('a1=',a1,'a2=',a2,'a3=',a3,'b=',b,'\n')

#Precisión del modelo
print('Precisión: ',algoritmo.score(data_test, target_test))
```

```
boston_keys: <built-in method keys of Bunch object at 0x0000022661991090>
feature_names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD'
'TAX' 'PTRATIO'
'B' 'LSTAT']
a1= 8.555405839258126 a2= -0.09765473439151359 a3= -0.526730440122916 b= -
22.349910364794013
```

Precisión: 0.48771752707595917

Selección de características/variables

Conjuntos de datos a veces muy grandes por tener muchas características y por eso son difíciles de preprocesar. A veces por ser tan numerosas las características pueden actuar como ruido, se puede ver el sistema como extremadamente lento o asignación de recursos innecesarios a esas características. Por todo esto hay que seleccionar las características más importantes/relevantes para poder mejorar la calidad de la predicción (más rápidos, eficientes y con un mejor entendimiento de los procesos subyacentes).

Metodologías y técnicas

- **Métodos de filtro:** Paso de preprocessamiento de datos, selección de características independientes de los algoritmos, las características se evalúan como mejores en función de más elevada sea su correlación con la variable dependiente o resultado. Así podemos buscar un mejor subconjunto de variables independientes y comprobar la eficiencia del algoritmo. No eliminan la multicolinealidad, por lo que se debe lidiar con ellos antes de entrenar los modelos. Algunos tipos:

Correlación de Pearson: características numéricas y resultado numérico, para ver la correlación entre variables indep y dep (varía de -1 a 1).

LDA: características numéricas y resultado categórico, encontrar combinación lineal de característica para separar 2 clases.

Anova: características categóricas y resultado numérico, similar a LDA pero funciona con más variables y proporciona info sobre si la medida de varios grupos son iguales o no.

Chi-cuadrado: características categóricas y resultado categórico, prueba estadística para evaluar la probabilidad de correlación o asociación usando su distribución de frecuencia.

- **Métodos de envoltura:** Necesita de un algoritmo de ML para ver su rendimiento y así poder hacer su evaluación. Probar un subconjunto de características o variables independientes y ver el rendimiento, iterar el proceso y observar el mejor de los subconjuntos. Es decir, un proceso de búsqueda, por lo cual tienen alto costo computacional.

Forward Selection: comenzamos sin características y se le agrega característica a característica por orden de mayor aporte al modelo y parar cuando la siguiente variable a añadir no aporte nada.

Backward Selection: empezar con todas las características y eliminar la característica que menos mejore el modelo e iterar ese proceso hasta que la eliminación de una característica no introduzca ninguna mejoría en la predicción.

Recursive Feature Elimination: busca el subconjunto con mejor rendimiento y construye un modelo con un subconjunto aleatorio, y aparta la mejor de ellas, consigue otro subconjunto y otro modelo y hace lo mismo, se itera el proceso consiguiendo las mejores características, hasta que se usen todas y luego las clasifica por orden de aportación.

- **Métodos integrados:** Presenta las cualidades de los métodos de filtro y de envoltura. Se implementa mediante algoritmos con sus propios métodos de selección de características integrados. Algunos de estos métodos son la selección LASSO y RI

Regresión Polinomial

Caso particular de la regresión lineal al añadir predictores adicionales obtenidos al elevar cada uno de los originales a una potencia, así se obtiene una relación no lineal (polinómica de orden distinto de cero o uno, por eso de las potencias) entre las variables independientes y la dependiente.

$$y = a_1x_1 + b \text{ // lineal}$$

$$y = a_1x_1 + a_2x_1^2 + b \text{ // polinómica}$$

A medida que aumentamos la complejidad, el número de características o parámetros aumenta considerablemente. A veces tiende a ajustarse drásticamente, demasiado. Los polinomios grandes pueden también acabar ocasionando errores grandes en la extrapolación, es decir, que haya habido un overfitting. Para evitar el uso de polinomios de alto grado se puede sustituir por muchos polinomios de bajo orden.

In [32]:

```
### Polynomial Regression (Práctica)

from sklearn import datasets, linear_model
import numpy as np
import matplotlib.pyplot as plt

boston = datasets.load_boston()
#print(boston)

print('boston_keys: ',boston.keys)

#print(boston.DESCR)
#Vemos hay 506 filas, 14 columnas o atributos y la 14 es el valor medio o target. Luego
nos explican todos los datos de las
#columnas.
#Nos dicen no hay ningún missing value por lo que no hay que realizar mucho preprocesamiento

#print(boston.data.shape)
#Nos da (506, 13), lo cual ya sabíamos

print('feature_names: ',boston.feature_names)

X_p = boston.data[:, np.newaxis, 5]

y_p = boston.target

plt.scatter(X_p, y_p)
plt.show()

from sklearn.model_selection import train_test_split

X_p_train, X_p_test, y_p_train, y_p_test = train_test_split(X_p, y_p, test_size=0.2)

# Ahora y antes de definir el algoritmo hay que decidir el grado del polinomio, y para
ello tenemos que importar cosas

from sklearn.preprocessing import PolynomialFeatures

# Definimos el grado del polinomio

poli_reg = PolynomialFeatures(degree = 6)
# Lo ideal es empezar con un número bajo e ir subiendo pero con cuidado a los sobreajustes u overfittings

# Se transforman las características existentes en características de mayor grado.
# Aquí se está haciendo lo de  $y = a_1x_1 + a_2x_1^2 + \dots$ 
X_p_train_poli = poli_reg.fit_transform(X_p_train)
X_p_test_poli = poli_reg.fit_transform(X_p_test)

# Definir el modelo
pr = linear_model.LinearRegression()

# Entrenar el modelo
pr.fit(X_p_train_poli, y_p_train)

# Realizamos una predicción con los valores reales
y_p_pred = pr.predict(X_p_test_poli)
```

Veamos los resultados frente al modelo

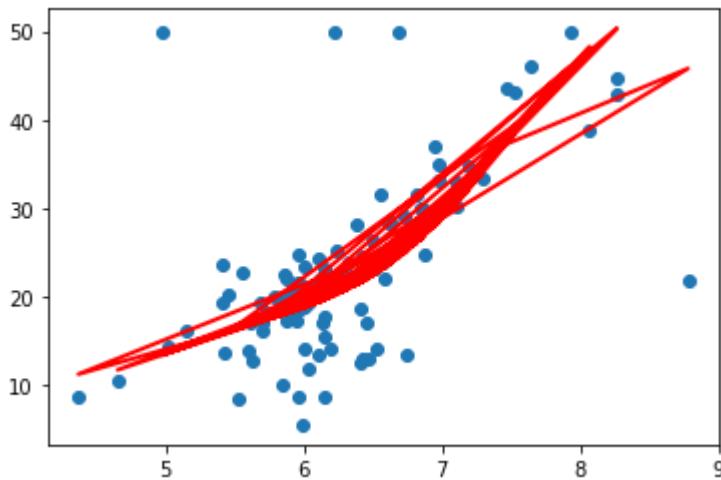
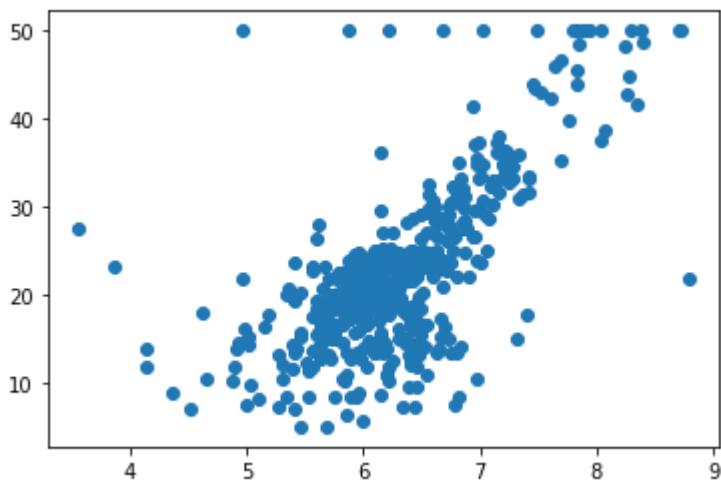
```
plt.scatter(X_p_test, y_p_test)
plt.plot(X_p_test, y_p_pred, color = 'red', linewidth = 2)
plt.show()
# Todos los datos fuera de la línea son errores, ahora vemos este modelo se ajusta muy bien
```

Vamos a calcular los parámetros y la precisión del modelo

```
a = pr.coef_
print('a:',a)
b = pr.intercept_
print('b:',b)
precision = pr.score(X_p_train_poli, y_p_train)
print('Precision:',precision)
# El primer valor de a siempre es cero, hay que ignorarlo, ya que es cosa del algoritmo
# Precisión no demasiado elevada ya que es c

# Si 2 parámetros tenemos Precision de 0.5510081564606899
# Si 3 parámetros tenemos Precision de 0.5372760023853598
# Si 4 parámetros tenemos Precision de 0.6256757881912909
# Si 5 parámetros tenemos Precision de 0.6006370790625328
# Si 6 parámetros tenemos Precision de 0.6088311696233033
# ...
```

```
boston_keys: <built-in method keys of Bunch object at 0x00000226619918B0>
feature_names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD'
'TAX' 'PTRATIO'
'B' 'LSTAT']
```



```
a: [ 0.0000000e+00 -3.98827733e+03  1.45988932e+03 -2.73729443e+02
 2.74065473e+01 -1.35841793e+00  2.48886402e-02]
b: 4401.013282255134
Precision: 0.6304232849777474
```

Vectores de Soporte de Regresión

Máquinas de Vectores de Soporte (MVS)

Las MVS se pueden aplicar a problemas de Clasificación y de Regresión(manteniendo todas las propiedades del caso de clasificación).

La idea es siempre minimizar el error o individualizar el hiperplano que maximiza el margen teniendo en cuenta que se considera parte del error.

Funciona muy bien para problemas lineales y no lineales.

Lo primero siempre es conseguir el mejor hiperplano que represente el comportamiento de estos datos. En caso de un problema lineal, ese hiperplano será una línea recta, mientras que si es no lineal, el hiperplano será mucho más complejo.

La fórmula para el hiperplano será la misma que para el lineal

$$y = wx + b$$

Lo siguiente es ver unas barras paralelas al hiperplano, w, que consigan cubrir el mayor número de datos posibles. A estas barras a distancia C se las conoce como vectores de apoyo o soporte. Esas barras no consiguen cubrir todos los puntos, y esos que quedan fuera son considerados los errores. Estos errores son los que se deben considerar para la fórmula del algoritmo, lo que se calcula es la epsilon (los errores por un lado, superiores al margen superior y los epsilon* como la distancia a los errores desde el margen inferior), que es la distancia (vertical si es lineal el problema) entre las barras y los puntos.

Así, la fórmula completa para el cálculo de este algoritmo utilizando datos lineales:

$$\min (0.5||w||^2) + C \sum_{i=1}^N (\text{epsilon} + \text{epsilon}^*)$$

el primer sumando es para maximizar el margen, el segundo para minimizar el error de entrenamiento.

w representa el vector o hiperplano

C es una constante que debe ser superior a cero, determina el equilibrio entre la regularidad de la función y la cuantía hasta la cual toleramos desviaciones mayores que las bandas de soporte.

epsilon y epsilon* son las variables que controlan el error cometido por la función de regresión al aproximar a las bandas

Si consideramos C demasiado grande (tendiendo a infinito) podremos ver estamos considerando el conjunto está perfectamente determinado por nuestro hiperplano predictor (overfitting), permitiendo epsilon cercanas a cero

Si consideramos C demasiado pequeño (tendiendo a cero), permitiendo epsilon elevados

Para ver la diferencia entre casos lineales y los no lineales se tiene que cambiar el kernel, pero el concepto es el mismo

El rendimiento del algoritmo de Vectores de Soporte de Regresión depende de una buena configuración de los parámetros C y los parámetros del kernel elegido.

Vectores de Soporte de Regresión

```
from sklearn.svm import SVR  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = SVR
```

Tenemos que tener en cuenta los valores que le pasamos al SVR, los importantes que habíamos visto eran C (por defecto es 1.0), luego tenemos epsilon (default 1.0) por lo general se suele dejar el default porque es complicado, y más si eres un noob de ML.

Otra opción que sí se modifica es la de kernel (tendremos que verla) por default tiene 'rbf', y otros posibles son 'linear', 'poly', 'sigmoid', 'precomputed' u otros

```
algoritmo.fit(x_train, y_train)  
  
algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [33]:

```
### Vectores de Soporte de Regresión

import numpy as np
import matplotlib.pyplot as plt

boston = datasets.load_boston()
#print(boston)

print('boston_keys: ',boston.keys)

#print(boston.DESCR)
#Vemos hay 506 filas, 14 columnas o atributos y la 14 es el valor medio o target. Luego
#nos explican todos los datos de las
#columnas.
#Nos dicen no hay ningún missing value por lo que no hay que realizar mucho preprocesamiento

#print(boston.data.shape)
#Nos da (506, 13), lo cual ya sabíamos

print('feature_names: ',boston.feature_names)

X_svr = boston.data[:, np.newaxis, 5]

y_svr = boston.target

plt.scatter(X_svr, y_svr)
plt.show()

from sklearn.model_selection import train_test_split

X_svr_train, X_svr_test, y_svr_train, y_svr_test = train_test_split(X_svr, y_svr, test_
size=0.2)

from sklearn.svm import SVR

#algoritmo = SVR(kernel='linear', C=1.0, epsilon=0.2)
algoritmo = SVR()

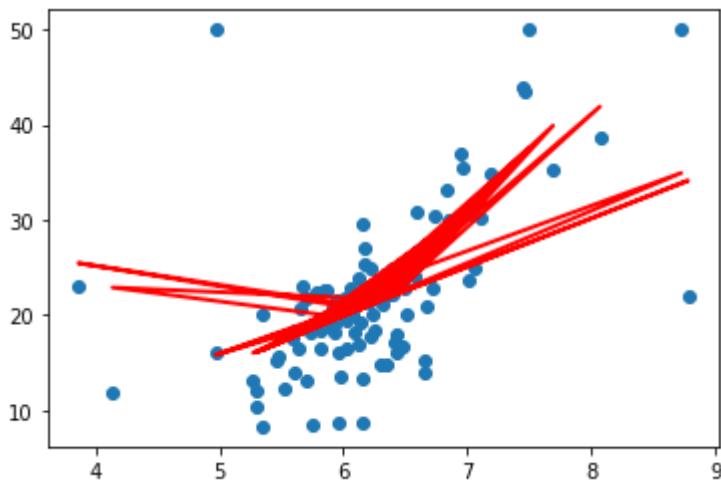
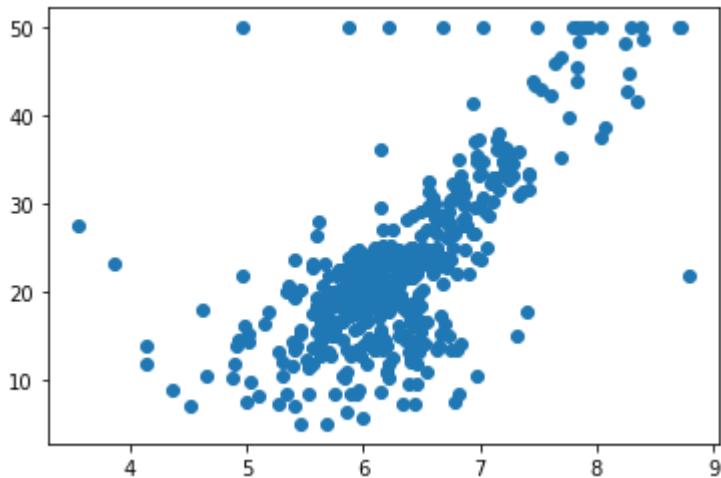
# Entrenar el modelo
algoritmo.fit(X_svr_train, y_svr_train)

# Realizamos una predicción con los valores reales
y_svr_pred = algoritmo.predict(X_svr_test)

# Veamos los resultados frente al modelo
plt.scatter(X_svr_test, y_svr_test)
plt.plot(X_svr_test, y_svr_pred, color = 'red', linewidth = 2)
plt.show()

precision = algoritmo.score(X_svr_test, y_svr_test)
print('Precision:',precision)
# Precision: 0.5404148744738217 con el SVR(kernel='linear', C=1.0, epsilon=0.2) el caso lineal
# Precision: 0.46392498391853343 con el SVR()
```

```
boston_keys: <built-in method keys of Bunch object at 0x00000226619911D0>
feature_names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD'
'TAX' 'PTRATIO'
'B' 'LSTAT']
```



Precision: 0.4859164202618478

Árboles de decisión de Regresión

Un árbol tiene muchas analogías, también en ML. Son una técnica de aprendizaje automático supervisado que predice valores de respuestas mediante el aprendizaje de reglas de regresión derivadas de características.

Se pueden utilizar tanto en regresión como en clasificación, ya que consiste en separar las muestras mediante regiones rectangulares simples.

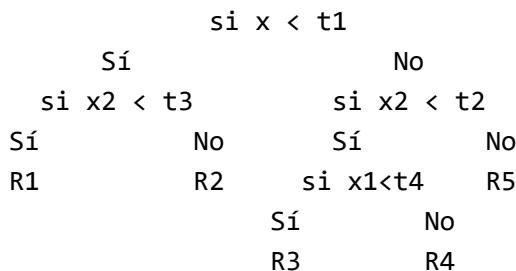
Para obtener una predicción para una observación particular se utiliza la media de las observaciones de entrenamiento dentro de la partición a la que pertenece dicha observación.

$$f(x) = \text{SUM}_{\{i=1}^N w_m * \Phi(x; v_m)$$

w_m es la respuesta media de la región particular R_m

v_m es la forma de dividirse de cada variable en un valor de umbral particular

Estas divisiones definen como el espacio de características en R^p en m regiones separadas o hiperbloques



Es decir tenemos condiciones y en base a eso generamos nuestro espacio dividido en rectángulos

Dadas las características p del espacio p -dimensional lo dividimos en m regiones mutuamente distantes que cubren todo el subconjunto de características y no se superponen. Estas características están dadas por esas regiones R_m . Cualquier nueva observación que caiga en alguno de esas regiones R_m podremos ver si tiene una respuesta (la variable independiente) dada por la media de las regiones de entrenamiento por la partición determinada por w_m , es decir, la media de ese subconjunto en el que cae. Sin embargo este proceso no define cómo crear la partición de forma algorítmica. Para ello lo que tenemos que hacer es usar una técnica conocida como división binaria recursiva.

Nuestro criterio para poder generar la partición y así ese algoritmo tiene que minimizar algún tipo de error. Por ello vamos a ver el mínimo de cuadrados residuales. Pero es demasiado costoso computacionalmente considerar todas las particiones posibles y encontrar la mejor, por lo que se usa la división binaria recursiva.

Un enfoque menos intensivo y más sofisticado. Comenzando el árbol por la parte superior lo divide en dos, y elige la partición que minimiza la suma de cuadrados residuales. Se va iterando este proceso y es computacionalmente factible y práctico para su uso, el problema es que es propenso al overfitting. Para evitar el overfitting lo que se puede hacer es detener el crecimiento antes de que sea demasiado grande o podar el árbol cuando ya haya acabado y sea demasiado grande. Ese tamaño se estima en función de la profundidad del árbol. Los datos disponibles para train el árbol formarán un conjunto de árboles de diferentes profundidades máximas en función del conjunto de capacitación y se validará con los test. La validación cruzada también se puede utilizar como parte de este enfoque.

La validación cruzada es que si tenemos 1, 2, 3, 4, 5 datos podemos coger 1, 2, 3, 4 como train y aplicarlo a 5 como test y luego probar con 1, 2, 3, 5, como train y 4 como test, es decir, ver las diferentes combinaciones posibles.

La poda del árbol implica probar un árbol original e ir recortando algunos nodos, quitarle profundidad y ver la comparación original-podado, en caso de que el podado sea el mejor, nos lo quedamos como nuevo árbol.

Este modelo no es demasiado bueno para la predicción (regresión) pero sí para la separación en clases (clasificación)

Árboles de decisión de Regresión

```
from sklearn.tree import DecisionTreeRegressor  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = DecisionTreeRegressor()
```

Tenemos que ver la configuración de este algoritmo criterion (default='mse'), es la media del error cuadrado para implementar la separación de los datos, otra posible es el error absoluto promedio MAE

Se puede utilizar también splitter, que es para ver la división en cada nodo, en este caso se cuenta con dos opciones 'best' (por defecto) y 'random', por lo general es mejor dejarlo como está

max_depth la profundidad del árbol, como default tiene 'None', pero hay que indicarle algo ya que el algoritmo hay que podarlo porque sino el algoritmo selecciona de forma automática los nodos hasta que las hojas estén puras o contengan menos datos, es decir, overfitting, por lo que debemos de modificarlo

```
algoritmo.fit(x_train, y_train)
```

```
algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [34]:

```
### Árboles de Decisión de Regresión

import numpy as np
import matplotlib.pyplot as plt

boston = datasets.load_boston()
#print(boston)

print('boston_keys: ',boston.keys)

#print(boston.DESCR)
#Vemos hay 506 filas, 14 columnas o atributos y la 14 es el valor medio o target. Luego nos explican todos los datos de las #columnas.
#Nos dicen no hay ningún missing value por lo que no hay que realizar mucho preprocesamiento

#print(boston.data.shape)
#Nos da (506, 13), lo cual ya sabíamos

print('feature_names: ',boston.feature_names)

X_adr = boston.data[:, np.newaxis, 5]

y_adr = boston.target

plt.scatter(X_adr, y_adr)
plt.show()

from sklearn.model_selection import train_test_split

X_adr_train, X_adr_test, y_adr_train, y_adr_test = train_test_split(X_adr, y_adr, test_size=0.2)

from sklearn.tree import DecisionTreeRegressor

# Definir el algoritmo e importante el max_depth para evitar el overfitting
algoritmo = DecisionTreeRegressor(max_depth = 4)

# Entrenar el modelo
algoritmo.fit(X_adr_train, y_adr_train)

# Realizamos una predicción con los valores reales
y_adr_pred = algoritmo.predict(X_adr_test)

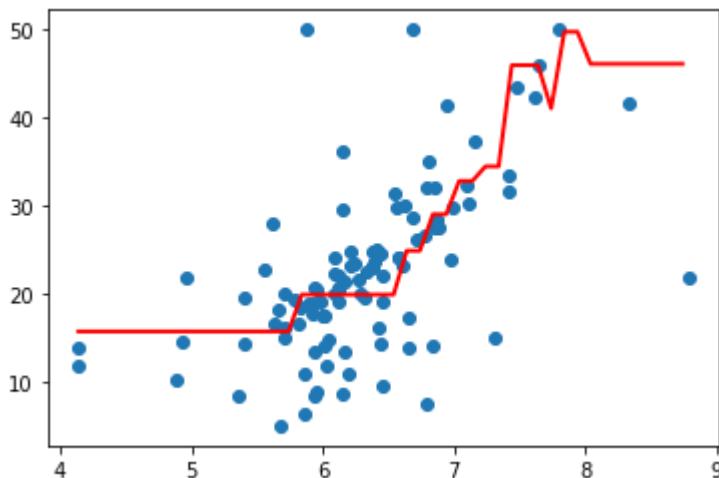
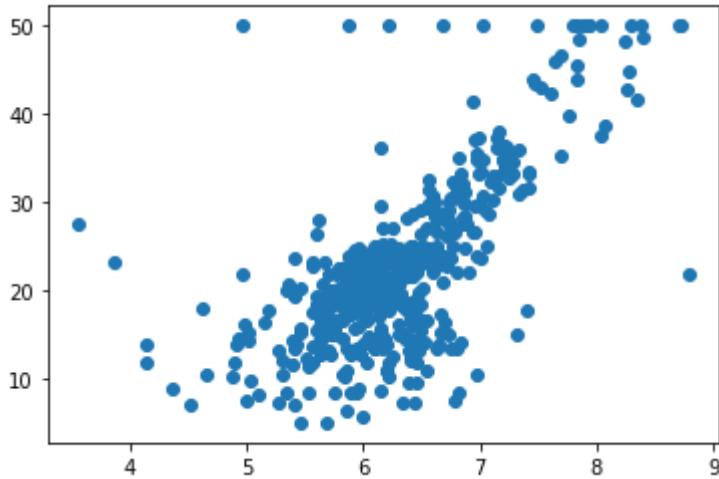
# Veamos los resultados frente al modelo
X_grid = np.arange(min(X_adr_test), max(X_adr_test), 0.1)
# Arange para espaciar el espacio de manera uniforme
X_grid = X_grid.reshape((len(X_grid), 1))
# Darle una nueva forma a la matriz de datos
plt.scatter(X_adr_test, y_adr_test)
plt.plot(X_grid, algoritmo.predict(X_grid), color = 'red', linewidth = 2)
plt.show()

# Modelo no es una Línea recta

#Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un #conjunto de datos con este algoritmo en particular
```

```
precision = algoritmo.score(X_adr_test, y_adr_test)
print('Precision:',precision)
## Precision: 0.4327209514538003 si max_depth = 4
## Precision: 0.49081651545392346 si max_depth = 5
## Precision: 0.3563025625656525 si max_depth = 6
```

```
boston_keys: <built-in method keys of Bunch object at 0x00000226619918B0>
feature_names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD'
'TAX' 'PTRATIO'
'B' 'LSTAT']
```



Precision: 0.37103818082459106

Método de ensamble de modelos (o métodos combinados)

Son métodos que intentan mejorar el rendimiento de algoritmos de ML a mejorar su precisión. Es decir, generar un nuevo algoritmo que resuelva mejor algunos problemas concretos de ML. Combinación de un conjunto de diferentes modelos para improvisar sobre la estabilidad y el poder predictivo del modelo.

Los modelos pueden ser (y son) diferentes por multitud de razones, entre ellas, diferencia en la población de datos, técnicas de modelado diferentes usadas, hipótesis diferentes.

La idea básica es la combinación de predicciones de modelos, promediar errores idiosincráticos y producir mejores predicciones generales.

Existen varios tipos de métodos de ensamble de modelos pero los más utilizados son

Agregación Bootstrap o Bagging: Dado un dataset se divide en subconjuntos de este dataset de forma aleatoria, se entrena los modelos de forma separada y se testan usando el subconjunto de prueba. La predicción final se combina en los proyectos de todos los submodelos.

Boosting o refuerzo: Es una fórmula de técnica de aprendizaje secuencial, el algoritmo funciona entrenando un modelo con todo el conjunto de entrenamiento y se actualiza el dataset teniendo en cuenta los valores de error residual y se le da más peso a las observaciones que el modelo anterior estimó previamente. Se itera este proceso ponderando las decisiones en función de su precisión y así finalmente se ve se da una estimación final.

Bagging vs. Boosting

-Ambos disminuyen la varianza de su estimación única ya que combinan varias estimaciones de diferentes modelos, obteniendo un modelo con mayor estabilidad.

-Si el problema es que el modelo único obtiene un rendimiento muy bajo, Boosting es mejor opción ya que optimiza ventajas y disminuye las ventajas del modelo único, ya que Bagging rara vez obtendrá un mejor sesgo.

-Si la dificultad del modelo único se da overfitting, entonces la mejor opción es Bagging.

Por la última razón, vemos suele ser más común el Bagging.

Bosques Aleatorios Regresión

Los Bosques Aleatorios son un algoritmo de ML fácil de usar y flexible que produce grandes resultados aún sin ajuste de parámetros en la mayor parte del tiempo. Es uno de los algoritmos más utilizados por su posible aplicación tanto en regresión como en clasificación.

Es un procedimiento de aprendizaje supervisado que crea un bosque de forma aleatoria, es decir, crea múltiples Árboles de Decisión y los combina de forma que se obtenga una mejor predicción, más precisa y estable. A más árboles, más preciso es el modelo y es más robusto.

Se genera la aleatoriedad adicional al modelo mientras crecen los árboles ya que al crecer los nodos ya que busca la mejor característica entre un subconjunto aleatorio de características, lo que da como resultado una amplia diversidad que generalmente resulta en un mejor resultado.

Por lo general suele considerar ese factor aleatorio pero se puede añadir más aleatoriedad aún usando umbrales aleatorios (en vez de los mejores) para cada función y como hacía cada Árbol de Decisión normal.

Árbol de Decisión vs Bosque Aleatorio

-Un Árbol de Decisión normal genera un conjunto de normas o reglas a esas características de entrenamiento que se usarán para las predicciones mientras que un Bosque Aleatorio selecciona al azar las observaciones y las características para construir varios Árboles de Decisión y luego promedia los resultados.

-Un Árbol de Decisión puede sufrir de overfitting, mientras que un Bosque Aleatorio evita el overfitting creando subconjuntos aleatorios de las características y árboles más pequeños usando dichas características. Posteriormente combina los sub-Árboles de Decisión. No funciona todas las veces y es lento, dependiendo ambas de cuántos árboles genere el bosque al azar.

Bosque Aleatorio

-Ventajas: Se considera un algoritmo muy útil y fácil, ya que los parámetros predeterminados no es muy alto y son fáciles de entender. El problema por lo general en un algoritmo de ML es el overfitting, pero en un Bosque Aleatorio podremos ver que con un número elevado de Árboles de Decisión tendremos que no habrá adaptación al modelo y por ello no habrá overfitting.

-Desventajas: Una gran cantidad de Árboles de decisión hace que el algoritmo sea lento e ineficiente, lo que lo hace un algoritmo malo para las predicciones a tiempo real, ya que son rápidos para entrenar pero lentos para hacer predicciones una vez entrenados. Además es una herramienta de modelado predictiva y no descriptiva.

Es un gran algoritmo para entrenar temprano y sencillo. Proporciona un buen indicador de la importancia que asigna a cada una de sus características.

Bosques Aleatorios de Regresión

```
from sklearn.ensemble import RandomForestRegressor  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = RandomForestRegressor()
```

n_estimators: integer (default: 10), el número de árboles, hace que sean más estables las predicciones pero también vuelve lento al cálculo

criterion: (default: 'mse') igual que en los Árboles de Decisión, utiliza la media del error cuadrado para implementar la separación de los datos en los árboles a construir

max_depth: (default: 'None'), los expande hasta que todas las hojas sean lo más sencillas posibles, como vimos en Árboles de Decisión

max_features: (default: 'auto'), el máximo de características que el Bosque Aleatorio debe probar en cada árbol individual.

```
algoritmo.fit(x_train, y_train)
```

```
algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [35]:

```
### Bosque Aleatorio de Regresión

import numpy as np
import matplotlib.pyplot as plt

boston = datasets.load_boston()
#print(boston)

print('boston_keys: ',boston.keys)

#print(boston.DESCR)
#Vemos hay 506 filas, 14 columnas o atributos y la 14 es el valor medio o target. Luego
nos explican todos los datos de las
#columnas.
#Nos dicen no hay ningún missing value por lo que no hay que realizar mucho preprocesamiento

#print(boston.data.shape)
#Nos da (506, 13), lo cual ya sabíamos

print('feature_names: ',boston.feature_names)

X_rf = boston.data[:, np.newaxis, 5]

y_rf = boston.target

plt.scatter(X_rf, y_rf)
plt.show()

from sklearn.model_selection import train_test_split

X_rf_train, X_rf_test, y_rf_train, y_rf_test = train_test_split(X_rf, y_rf, test_size=0.2)

from sklearn.ensemble import RandomForestRegressor

# Definimos el algoritmo
algoritmo = RandomForestRegressor(n_estimators = 300, max_depth = 8)
# Parámetros para poder ver el comportamiento del modelo

# Entrenamos el modelo
algoritmo.fit(X_rf_train, y_rf_train)

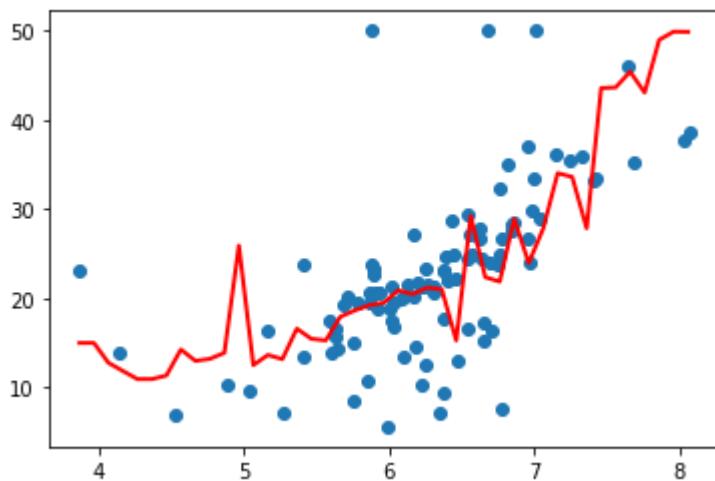
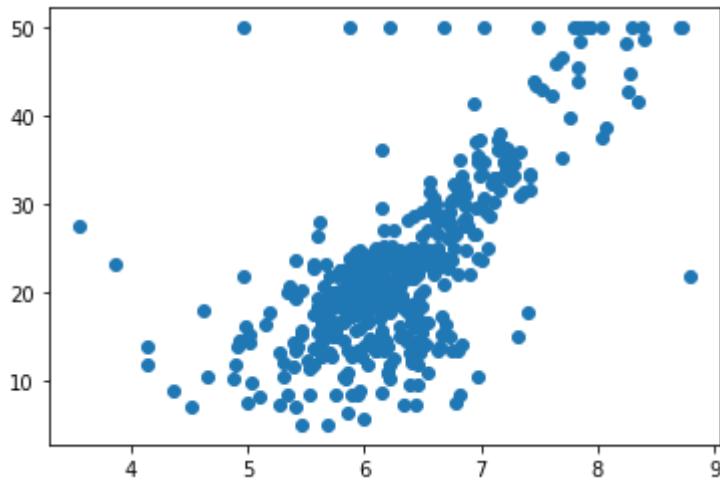
# Hacemos la predicción
y_rf_pred = algoritmo.predict(X_rf_test)

# Veamos los resultados frente al modelo
X_grid = np.arange(min(X_rf_test), max(X_rf_test), 0.1)
# Arange para espaciar el espacio de manera uniforme
X_grid = X_grid.reshape((len(X_grid), 1))
# Darle una nueva forma a la matriz de datos
plt.scatter(X_rf_test, y_rf_test)
plt.plot(X_grid, algoritmo.predict(X_grid), color = 'red', linewidth = 2)
plt.show()
# La línea se parece a la de los Árboles de Regresión

#Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para
a ver qué
#tan bien funcionan un conjunto de datos con este algoritmo en particular
```

```
precision = algoritmo.score(X_rf_test, y_rf_test)
print('Precision:',precision)
```

```
boston_keys: <built-in method keys of Bunch object at 0x0000022662A9BAE0>
feature_names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD'
'TAX' 'PTRATIO'
'B' 'LSTAT']
```



Precision: 0.3792688811621745

Ventajas y Desventajas de los Algoritmos de Regresión

Debemos tener en cuenta ningún algoritmo es el mejor en todos los problemas, cada uno se debe usar en un contexto dado, y todos tienen sus pros y cons. Vamos a ver algunos criterios por los cuales podríamos ver cuál es el mejor a elegir en cada caso. Recordemos son:

- Regresión lineal (Simple, Múltiple, Polinomial): Tiene ventajas como una única variable, fácil de entender y modelar cuando la relación no es compleja. Menos propenso al overfitting. Sin embargo tiene algunas desventajas como no poder adecuarse a relaciones no lineales sin modificar la entrada o relaciones complejas y sufren con valores atípicos. Suelen ser útiles para un primer vistazo inicial, si se tienen datos numéricos con muchas características
- Vectores de Soporte de Regresión: Se basa en la construcción de un hiperplano óptimo en forma de superficie de decisión. Hacen referencia a un subconjunto de las observaciones de train y poder hacer las descripciones o predicciones. Algunas de sus ventajas son poder modelar relaciones complejas no lineales y robustez al ruido (maximizar los márgenes C). Y presenta también algunas desventajas como la necesidad de seleccionar buenas funciones kernel y con una apropiada elección de parámetros, los parámetros son más difíciles de interpretar, requiere alta memoria y poder de preprocesamiento por lo que toma demasiado tiempo para entrenar. Suelen ser útiles para la clasificación de texto e imágenes y también para el reconocimiento de escritura a mano.
- Árboles de Decisión de Regresión: Dado un conjunto de datos se fabrican diagramas de construcciones lógicas que sirven para representar y categorizar condiciones, que ocurren de forma sucesiva, para la resolución de un problema. Las ventajas es que es muy fácil de implementar e interpretar, es robusto al ruido y valores perdidos, preciso y es excelente para entender relaciones complejas no lineales. Por lo general suelen alcanzar un rendimiento bastante alto. Sin embargo, tenemos varias desventajas, como la dificultad para interpretar árboles complejos, que a veces hay duplicación dentro del mismo subárbol, y en ocasiones no es usado por ser un algoritmo tan sencillo y poco preparado para datos complejos. Es un algoritmo útil para diagnóstico médico y análisis de riesgo creticio
- Bosques Aleatorios de Regresión: Combinación de Árboles de Decisión que depende de los valores aleatorios de un vector y es probado independientemente y con la misma distribución para todos los árboles. Algunas ventajas son que puede trabajar en paralelo, el overfitting no suele ocurrir, maneja automáticamente los missing values, no es necesario realizar ninguna transformación en los parámetros y es muy fácil de usar. Por el contrario las desventajas son la dificultad de interpretamiento, más débil en la regresión (era un buen clasificador recordemos) al estimar predicciones en los valores extremos de respuesta, y es parcialmente multiclasificadora hacia las clases más frecuentes. Este problema se suele usar en casi cualquier problema de ML y en bioinformática.

Matriz de confusión

Un modelo de clasificación de Machine Learning (ML) es aquel que predice una variable 'y' que es categórica. La matriz de confusión es una de las métricas más intuitivas y sencillas que se utiliza para encontrar la precisión y exactitud del modelo. Se utiliza para un problema cuando este presenta dos o más clases o variables independientes.

Si ponemos como ejemplo la predicción de una persona teniendo cáncer ó no (1 ó 0)

		Predicción	
		Positivo	Negativo
Real	Positivo	T.P.	F.N.
	Negativo	F.P.	T.N.

-T.P. (True Positive): Predicted 1, Real 1

-T.N. (True Negative): Predicted 0, Real 0

-F.P. (False Positive): Predicted 1, Real 0

-F.N. (False Negative): Predicted 0, Real 1

Falsos (F.P., F.N.) si se ha errado la predicción frente al valor real

Verdaderos(T.P., T.N.) si se ha acertado en la predicción en el valor real

En sí misma no es una medida de rendimiento pero casi todas las métricas se basan en los números que contienen

Evaluando el error en Modelos de Clasificación

Cómo calcular los errores en los modelos de clasificación tras clasificar los datos, hay que evaluar el modelo, para ello tenemos que ver diferentes métricas de rendimiento para evaluar estos modelos.

-Exactitud: El número de predicciones correctas frente al total de predicciones
 $\text{accuracy} = (\text{TP}+\text{TN})/(\text{TP}+\text{TN}+\text{FP}+\text{FN})$

-Precisión: Evaluamos las predicciones "positivas" $\text{precision} = (\text{TP}+\text{FP})/(\text{TP}+\text{TN}+\text{FP}+\text{FN})$

-Sensibilidad/Recall: Número de predicciones positivas correctas frente al número total de positivos $\text{recall} = \text{TP}/(\text{TP}+\text{FN})$

-Especificidad: Número de predicciones negativas correctas frente al total de negativos $\text{specificity} = \text{TN}/(\text{TN}+\text{FP})$

-Puntaje de F1: Como realmente no queremos calcular la precisión y sensibilidad cada vez que calculamos el modelo así que podemos combinarlas en una sola magnitud, como un criterio ponderado $\text{F1} = 2*\text{precision}*\text{accuracy}/(\text{precision}+\text{accuracy})$

Así esta puntuación tiene en cuenta tanto FP como FN

Curvas ROC y Áreas bajo la Curva AUC

En ML la medición del rendimiento es esencial. Tenemos en estos problemas la curva AUC ROC, ROC viene de las características de funcionamiento del receptor y AUC es el área bajo la curva.

La curva ROC es una característica de funcionamiento del receptor y nos dice que tan bien puedo distinguir un modelo entre dos clases. Un modelo bueno podrá medir bien y uno pobre no podrá distinguir bien.

Debemos incluir un umbral y todos los valores por encima de este umbral serán TP y los negativos por encima de este umbral serán FP, ya que se predicen incorrectamente como positivos. Los valores negativos por debajo de ese umbral serán TN y los positivos por debajo del umbral serán FN, ya que se predicen incorrectamente como negativos. Idea básica de que el modelo predice en base a un umbral de esta forma.

TN & FN UMBRAL FP & TP

Si recordamos

$$\text{recall} = \text{TP}/(\text{TP}+\text{FN})$$

$$\text{specifity} = \text{TN}/(\text{TN}+\text{FP})$$

Ahora vemos si disminuimos el umbral vamos a aumentar en FP y disminuir FN, por lo cual aumenta recall y disminuye specificity

Ahora vemos si aumentamos el umbral vamos a aumentar en FN y disminuir FP, por lo cual disminuye recall y aumenta specificity

Y se pinta (recall) vs (1-specificity), y eso es la curva ROC, eso es esa curva con forma de raíz cuadrada semicircular

¿Por qué 1-specificity? $1 - \text{specificity} = 1 - \text{TN}/(\text{TN}+\text{FP}) = \text{FP}/(\text{TN}+\text{FP})$

Es decir, mientras que specificity nos da TN, 1-specificity nos da FP, así vemos 1-specificity nos da la tasa de FP

AUC es el área bajo la curva ROC, y este puntaje nos da una idea de qué tan bien funciona el modelo, debe de ser un escalón, área bajo la curva (con respecto a la diagonal) es 1.0, esta situación ocurre cuando TN + FP & FN + TP = N & P, cuando están totalmente separadas. y el peor de los casos es la diagonal, cuando no se distingue nada de nada, es 0.5 esa diagonal.

Este método es bueno porque es invariable frente a la escala, mide qué tan bien se clasifican las predicciones en lugar de sus valores absolutos. Es invariable frente al umbral de clasificación, la calidad de las predicciones es independiente de ese umbral de clasificación.

Cómo implementar las Métricas de Evaluación

A medida que se entrene un modelo predictivo de clasificación queremos ver ese rendimiento asociado a cada modelo usado.

-La matriz de confusión

```
from sklearn.metrics import confusion_matrix  
  
matrix = confusion_matrix(y_test, y_pred)  
  
print(matrix)
```

-Accuracy

```
from sklearn.metrics import accuracy_score  
  
accuracy = accuracy_score(y_test, y_pred)  
  
print(accuracy)
```

-Precisión

```
from sklearn.metrics import precision_score  
  
precision = precision_score(y_test, y_pred)  
  
print(precision)
```

-Sensibilidad/Recall

```
from sklearn.metrics import recall_score  
  
recall = recall_score(y_test, y_pred)  
  
print(recall)
```

-Puntaje F1

```
from sklearn.metrics import f1_score  
  
f1 = f1_score(y_test, y_pred)  
  
print(f1)
```

-ROC-AUC

```
from sklearn.metrics import roc_auc_score  
  
roc_auc = roc_auc_score(y_test, y_pred)  
  
print(roc_auc)
```

Conjunto de Datos Desbalanceados

Detección de fraudes en banca, ofertas en tiempo real en el mercado o detección de intrusos en redes; todas ellas tienen en común que los datos usados en esta área suelen tener menos de un 1% de eventos raros o interesantes. Este conjunto de datos desbalanceados es malo para los que trabajamos con ML ya que la mayoría de métodos son malos para manejarlos.

Un grupo desbalanceado es aquel en la que una de las clases es muy muy muy predominante en cantidad de eventos frente a la otra. También hay un problema de métricas, si se tiene un problema de estos con tanta diferencia de número de eventos, nuestros algoritmos casi seguro indicarán como el más predominante o todas las predicciones para esa clase mayoritaria, lo cual no parece malo, ya que tendremos un accuracy de un 99% pero la clase minoritaria está siendo ignorada...

Por esto es que tenemos que tener en cuenta esa clase minoritaria. Tenemos varias soluciones:

-Recopilar más datos: para una perspectiva diferente y quizás más equilibrada de las clases

-Uso de las métricas de evaluación correctas: Accuracy mal. Precisión, Especificidad, Sensibilidad, Puntaje F1, ROC-AUC

-Remuestreo del conjunto de datos: Obtener diferentes conjuntos de datos, intentar igualar la proporción de eventos entre clases, para ello buscaremos aumentar las muestras de la clase minoritaria (añadir copias de la clase minoritaria); sobre muestreo o muestreo obsesivo, ó podemos reducir las muestras de la clase mayoritaria; submuestreo o muestreo aleatorio. Al eliminar aleatoriamente se le asigna a la clase minoritaria y se va generando un vacío de forma aleatoria.

Submuestreo: Las ventajas del submuestreo son que ayuda a reducir el tiempo de ejecución, pero como desventajas es que puede descartar información útil que podría ser utilizada por los bosques aleatorios para generar clases, también tendríamos estas muestras aleatorias estarán sesgadas y podrá ser un mal predictor o ya no predecir del todo bien la predicción y el clasificador comportarse mal en el testeo.

Sobre muestreo: Las ventajas del sobre muestreo aleatorio son que al replicar la clase minoritaria estamos aumentando el número de eventos por lo que no conduce a la pérdida de información. Sin embargo, algunas desventajas son que aumenta la probabilidad de un exceso de equipamiento en la clase minoritaria ya que repiten estos eventos y considera usar un sobre muestreo cuando no se tengan muchos datos (ya que es incrementar aún más ese tiempo de ejecución).

Aprendizaje Supervisado. Algoritmos de Clasificación

Regresión Logística

La regresión logística es un método de regresión común a muchos problemas de ML (~30% de los problemas, el otro 70% es de clasificación). La regresión logística es un método estadístico para predecir clases binarias o dicotómicas, dos clases posibles. Es uno de los algoritmos más simples y usados en cualquier problema para la clasificación, se usa en casi todos los problemas y puede ser la línea de base de cualquier problema de clasificación binaria.

Lleva el nombre de la función utilizada en el núcleo del método, o función sigmoide (sigmoid function), una especie de S suave, como un escalón, va de 0 a 1, si la salida es mayor que 0.5, se puede clasificar el resultado como 1 o 'yes', y si es menor que 0.5, se puede clasificar el resultado como 0 o 'no'.

Al igual que la función lineal era

$$y = a_1x_1 + a_2x_2 + \dots + a_nx_n + b$$

La función sigmoide viene definida como

$$p = 1/(1 + e^{-y})$$

Y si lo sustituimos vemos sería

$$p = 1/(1 + e^{-(a_1x_1 + a_2x_2 + \dots + a_nx_n + b)})$$

Podemos ver la comparativa de la función sigmoide con la lineal, regresión logística frente a regresión lineal

- Regresión Lineal: Salida continua, conocer el porcentaje o probabilidad de un suceso

- Regresión Logística: Salida discreta, saber una respuesta dicotómica

Tenemos más tipos de Regresión Logística

- Regresión Logística Lineal: variable dependiente u objetivo con dos resultados posibles (dicotómica)

- Regresión Logística Multinomial: variable objetivo con tres o más categorías nominales

- Regresión Logística Ordinal: variable objetivo con tres o más categorías numéricas

Un algoritmo simple para clasificación de clases binarias y multivariadas.

In [36]:

```
##### Regresión Logística

import numpy as np
import matplotlib.pyplot as plt
# Implementamos las datasets
from sklearn import datasets

dataset = datasets.load_breast_cancer()
#print(dataset)

#print('dataset_keys: ',dataset.keys())
#print(dataset.DESCR)
#569 instancias y 30 clases o características o atributos

# Tenemos la media, error estándar y el error de esos 30 atributos

# No hay missing values por lo cual ya es una parte del preprocesamiento que nos ahorra
mos

# Tenemos 212 malignos y 357 benignos, por lo cual vemos que tenemos los datos balanceados, no es un dataset desbalanceado

#print('feature_names: ',dataset.feature_names)

X_lr = dataset.data
# Todas las variables del dataset

y_lr = dataset.target

#plt.scatter(X_lr, y_lr)
#plt.show()

from sklearn.model_selection import train_test_split

X_lr_train, X_lr_test, y_lr_train, y_lr_test = train_test_split(X_lr, y_lr, test_size=0.2)

# Lo siguiente a hacer es escalar los datos, ya que las características son diferentes
en magnitudes, unidades y rango, por
# por lo que lo mejor es escalar dichas magnitudes

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_lr_train = scaler.fit_transform(X_lr_train)
X_lr_test = scaler.fit_transform(X_lr_test)

# Datos ya preparados para poder implementar el modelo

from sklearn.linear_model import LogisticRegression

# Definimos el modelo
lr = LogisticRegression()

# Entrenamos el modelo
lr.fit(X_lr_train, y_lr_train)

# Hacemos la predicción
y_lr_pred = lr.predict(X_lr_test)
```

```
# Vamos a calcular las métricas de evaluación para ver qué tal, ha ido
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score

matrix = confusion_matrix(y_lr_test, y_lr_pred)
accuracy = accuracy_score(y_lr_test, y_lr_pred)
precision = precision_score(y_lr_test, y_lr_pred)
recall = recall_score(y_lr_test, y_lr_pred)
f1 = f1_score(y_lr_test, y_lr_pred)
roc_auc = roc_auc_score(y_lr_test, y_lr_pred)

print('Matriz de confusión:\n', matrix)
print('accuracy: ', accuracy)
print('precision: ', precision)
print('recall: ', recall)
print('f1: ', f1)
print('Curva ROC-AUC: ', roc_auc)
```

Matriz de confusión:

```
[[39  2]
 [ 2 71]]
accuracy:  0.9649122807017544
precision:  0.9649122807017544
recall:  0.9726027397260274
f1:  0.9726027397260274
Curva ROC-AUC:  0.9619111259605746
```

k-vecinos más cercanos / k-nearest neighbors (KNN)

Uno de los algoritmos de clasificación de ML más básico y esencial, es de aprendizaje supervisado y presenta posibles aplicaciones en detección de patrones, minería web y detección de intrusos. Se puede usar también como algoritmo de regresión pero tiene una metodología diferente.

Es un algoritmo no paramétrico, es decir, no hace suposiciones sobre la forma funcional de datos, evitando modelar mal la distribución subyacente de los datos. Suponiendo que los datos son altamente no gaussianos, si se predice un modelo gaussiano va a dar una predicción muy pobre.

Nuestro algoritmo, con aprendizaje basado en distancia, implica que este algoritmo no aprende explícitamente de un modelo, en lugar de ello memoriza las instancias de formación que luego usará para la predicción. Si solo hace una consulta, el algoritmo predecirá en base a esas instancias memorizadas.

La fase de formación mínima de KNN tiene asociado un alto coste en memoria y en tiempo computacional, no durante el periodo de entrenamiento, sino en el periodo de test, ya que requiere de un agotamiento de todo el conjunto de datos. En la práctica esto no es deseable, ya que normalmente queremos respuestas rápidas.

Lo que hace este algoritmo es que si queremos hacer una predicción a una observación Z tenemos que observar cuáles serán sus k-vecinos más cercanos y una vez elegidos veremos tendremos que ver cuál es la clase predominante target de esos k-vecinos y será la que asociemos a Z. Para encontrar los KNN tenemos que calcular la distancia a un punto, la opción más común suele ser la distancia Euclídea, pero depende del caso tenemos la distancia Manhattan y la distancia Minkowski, y según el caso algunas serán más apropiadas que otras.

¿Cómo se define K? K es un hiperparámetro que se debe definir al implementar el modelo, cada conjunto de datos es diferente del resto por lo cual no hay un K fijo para todos los problemas de KNN y debemos buscar el óptimo para cada problema por individual. Una pequeña cantidad de vecinos será flexible y tendrán un bajo sesgo pero una alta varianza, mientras que una alta cantidad de vecinos tendrá un elevado sesgo pero baja varianza. Se recomienda elegir un número de K impar si el número de clases es par. También se puede hacer el modelo, ir cambiando K y ver el rendimiento para estos parámetros variando.

KNN K-Nearest Neighbors / K-vecinos más cercanos

```
from sklearn.neighbors import KNeighborsClassifier  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = KNeighborsClassifier()
```

n_neighbors el número de k vecinos, (default: 5), pero es muy recomendable cambiarlo

Para el cómo calcular la distancia tenemos dos parámetros, p y metric:

p (default: 2) si se quiere distancia Manhattan se tendrá que poner p=1, y si se quiere Minkowski p con un valor arbitrario

metric (default: "minkowski") realmente el de defecto es la distancia Euclídea

```
algoritmo.fit(x_train, y_train)  
  
y_pred = algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [37]:

```
##### KNN K-Nearest Neighbors / K-vecinos más cercanos

import numpy as np
import matplotlib.pyplot as plt
# Implementamos las datasets
from sklearn import datasets

dataset = datasets.load_breast_cancer()
#print(dataset)

#print('dataset_keys: ',dataset.keys())
#print(dataset.DESCR)
#569 instancias y 30 clases o características o atributos

# Tenemos la media, error estándar y el error de esos 30 atributos

# No hay missing values por lo cual ya es una parte del preprocesamiento que nos ahorra
#mos

# Tenemos 212 malignos y 357 benignos, por lo cual vemos que tenemos los datos balanceados,
#no es un dataset desbalanceado

#print('feature_names: ',dataset.feature_names)

X_knn = dataset.data
# Todas las variables del dataset

y_knn = dataset.target

#plt.scatter(X_lr, y_lr)
#plt.show()

from sklearn.model_selection import train_test_split

X_knn_train, X_knn_test, y_knn_train, y_knn_test = train_test_split(X_knn, y_knn, test_
size=0.2)

# Lo siguiente a hacer es escalar los datos, ya que las características son diferentes
#en magnitudes, unidades y rango, por
#por lo que lo mejor es escalar dichas magnitudes

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_knn_train = scaler.fit_transform(X_knn_train)
X_knn_test = scaler.fit_transform(X_knn_test)

# Datos ya preparados para poder implementar el modelo

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 5, metric = "minkowski", p = 2)
# Con los dos últimos parámetros realmente estamos eligiendo la distancia Euclídea
# Se recomienda cambiar los valores de n_neighbors

knn.fit(X_knn_train, y_knn_train)

y_knn_pred = knn.predict(X_knn_test)
```

```
# Vamos a calcular las métricas de evaluación para ver qué tal, ha ido
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score

matrix = confusion_matrix(y_knn_test, y_knn_pred)
accuracy = accuracy_score(y_knn_test, y_knn_pred)
precision = precision_score(y_knn_test, y_knn_pred)
recall = recall_score(y_knn_test, y_knn_pred)
f1 = f1_score(y_knn_test, y_knn_pred)
roc_auc = roc_auc_score(y_knn_test, y_knn_pred)

print('#Matriz de confusión:\n',matrix)
print('#accuracy: ', accuracy)
print('#precision: ', precision)
print('#recall: ', recall)
print('#f1: ', f1)
print('#Curva ROC-AUC: ',roc_auc)

#n_neighbors=5, metric = "minkowski", p = 2

#Matriz de confusión:
# [[35  2]
# [ 4 73]]
#accuracy:  0.9473684210526315
#precision:  0.9473684210526315
#recall:  0.9733333333333334
#f1:  0.9605263157894737
#Curva ROC-AUC:  0.946998946998947

#n_neighbors=10, metric = "minkowski", p = 2

#Matriz de confusión:
# [[41  5]
# [ 0 68]]
#accuracy:  0.956140350877193
#precision:  0.956140350877193
#recall:  0.9315068493150684
#f1:  0.9645390070921985
#Curva ROC-AUC:  0.9456521739130435

#n_neighbors=100, metric = "minkowski", p = 2

#Matriz de confusión:
# [[34  9]
# [ 0 71]]
#accuracy:  0.9210526315789473
#precision:  0.9210526315789473
#recall:  0.8875
#f1:  0.9403973509933775
#Curva ROC-AUC:  0.8953488372093023

#n_neighbors=2, metric = "minkowski", p = 2

#Matriz de confusión:
```

```
# [[47  3]
# [ 2 62]]
#accuracy:  0.956140350877193
#precision:  0.956140350877193
#recall:  0.9538461538461539
#f1:  0.9612403100775193
#Curva ROC-AUC:  0.954375
```

#Matriz de confusión:

```
[[41  4]
[ 2 67]]
#accuracy:  0.9473684210526315
#precision:  0.9473684210526315
#recall:  0.9436619718309859
#f1:  0.9571428571428571
#Curva ROC-AUC:  0.9410628019323671
```

Máquinas de Vectores de Soporte Clasificación (Support Vector Machine, SVM)

Ofrecen una precisión en clasificación muy alta frente a otros clasificadores como la regresión logística o los árboles de decisión. Es conocido por su truco de kernel, por lo cual puede resolver problemas no lineales. Tiene aplicación en problemas de detección de rostros, reconocimiento de intrusos, clasificación de correos electrónicos y noticias en páginas web...

El SVM, por las siglas en inglés, construye un hiperplano de d-dimensiones para separar las diferentes clases, y para minimizar un error va iterando el proceso de construcción de este hiperplano hasta que finalmente lo encuentra el hiperplano marginal óptimo.

Los vectores de soporte son los puntos de datos más cercanos al hiperplano, estos puntos definirán mejor la línea de separación calculando los márgenes, esos puntos serán los más importantes para el clasificador. El hiperplano es una superficie d-dimensional que va a separar los puntos que tengan pertenencia a clases diferentes. El margen es el espacio entre las dos líneas de separación y los puntos más cercanos de la clase. A mayor sea el margen, mejor.

El objetivo es diferenciar el conjunto de observaciones de la manera más eficiente posible, es decir, elegir el hiperplano con el mayor margen posible en el espacio de hiperplanos posibles. Para elegir el mejor de los hiperplanos tenemos que ver todos los hiperplanos que mejor separan las clases y luego de ellos seleccionar el mejor o que menor error de clasificación tenga. No siempre son hiperplanos lineales, ya que se usan los kerneles para transformar el espacio de entrada en un espacio dimensional superior y entonces poder trazar un hiperplano que sí que separe de forma correcta las clases. Es decir, un kernel transforma un espacio de datos en la forma requerida.

Ventajas: Ofrecen buena precisión, y realizan predicciones más rápidas en comparación al algoritmo de Naive Bayes. Utilizan menos memoria ya que utilizan un subconjunto del total de datos para hacer el entrenamiento. Es un algoritmo que funciona bien con un claro margen de separación y con un espacio dimensional elevado.

Desventajas: Las SVM no son buenas con grandes cantidades de datos ya que se necesita gran tiempo de computación (en comparación con el algoritmo de Naive Bayes), funciona mal con clases superpuestas y es sensible (dependiente) del tipo de kernel utilizado

Kernel o núcleo, es un método para el análisis de patrones muy usado en ML aunque su principal aplicación es en las SVM. La tarea general es encontrar los tipos de relaciones entre datos de un conjunto. Los kernel deben su nombre al uso de funciones de núcleo, que les permiten operar en un espacio de características implícito y de alta dimensión sin tener que calcular las coordenadas en ese espacio sino calculando los productos internos entre las imágenes de todos los pares de datos en el espacio de características. Todo esto es mucho más barato computacionalmente en tiempo que calculando todas las coordenadas.

Se busca mapear más dimensiones, calcular el hiperplano y hacer la clasificación. Sin embargo, a mayor sea esa dimensión en la que vayamos a mapear, vamos a ver se vuelve más caros computacionalmente. Para esto es que tenemos el truco del Kernel, ya que nos permite operar en el espacio dimensional de la característica original sin aumentar el número de dimensiones de las coordenadas (más bien, calcular las coordenadas en ese espacio dimensionalmente superior). Por ello vemos nos permite una forma más eficiente y menos costosa de hacer la transformación a estos datos. No se limita solo el concepto de kernel a SVM, cualquier método que esté haciendo productos de x^*y de alguna forma veremos nos viene bien para trabajar con mayor número dimensional pero que realmente sea el mismo.

Tipos de Kernel:

-Kernel Lineal: el producto normal entre dos observaciones dadas $K(x, x_i) = \text{SUM}(x * x_i)$

-Kernel Polinomial: forma más generalizada del Kernel Lineal, no solo observa las características dadas por las muestras de entrada para determinar su similitud sino que también las combinaciones de estas, con n características y d grados de 1 polinomio obtenemos n^d características expandidas $K(x, x_i) = 1 + \text{SUM}(x * x_i)^d$

-Kernel RBF (de base radial o radio gaussiano): puede mapear un espacio de entrada en un espacio dimensional infinito, el parámetro gamma va de 0 a 1, cuando es 1 será overfitting, por lo que se suele elegir $0.1 K(x, x_i) = \exp(-\gamma \sum(x - x_i)^2)$

Este SVM con sus Kernel suena perfecto pero al mapear los datos a una dimensión superior hay peligro de overfitting, y hay que saber hacer una buena elección del Kernel y de los parámetros.

Máquinas de Vectores de Soporte Clasificación (Support Vector Machine, SVM)

```
from sklearn.svm import SVC  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = SVC()
```

kernel (default: "rbf"), también tenemos otras opciones como "linear", "poly", "sigmoid" o "precomputed" entre otras

C (default: 1.0), un parámetro de regularización o penalización, representa clasificación errónea o término de error, le indica cuánto error es aceptable, así se ve el criterio de compromiso entre el límite de decisión y el límite de clasificación errónea, a mayor sea C mayor es el margen

```
algoritmo.fit(x_train, y_train)  
  
y_pred = algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [38]:

```
##### Máquinas de Vectores de Soporte Clasificación (Support Vector Machine, SVM)

import numpy as np
import matplotlib.pyplot as plt
# Implementamos las datasets
from sklearn import datasets

dataset = datasets.load_breast_cancer()
#print(dataset)

#print('dataset_keys: ',dataset.keys())
#print(dataset.DESCR)
#569 instancias y 30 clases o características o atributos

# Tenemos la media, error estándar y el error de esos 30 atributos

# No hay missing values por lo cual ya es una parte del preprocesamiento que nos ahorra
mos

# Tenemos 212 malignos y 357 benignos, por lo cual vemos que tenemos los datos balanceados, no es un dataset desbalanceado

#print('feature_names: ',dataset.feature_names)

X_svd = dataset.data
# Todas las variables del dataset

y_svd = dataset.target

#plt.scatter(X_lr, y_lr)
#plt.show()

from sklearn.model_selection import train_test_split

X_svd_train, X_svd_test, y_svd_train, y_svd_test = train_test_split(X_svd, y_svd, test_
size=0.2)

# Lo siguiente a hacer es escalar los datos, ya que las características son diferentes
en magnitudes, unidades y rango, por
# por lo que lo mejor es escalar dichas magnitudes

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_svd_train = scaler.fit_transform(X_svd_train)
X_svd_test = scaler.fit_transform(X_svd_test)

# Datos ya preparados para poder implementar el modelo

from sklearn.svm import SVC

svc = SVC(kernel = 'linear')

svc.fit(X_svd_train, y_svd_train)

y_svd_pred = svc.predict(X_svd_test)

# Vamos a calcular las métricas de evaluación para ver qué tal, ha ido
from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score

matrix = confusion_matrix(y_svd_test, y_svd_pred)
accuracy = accuracy_score(y_svd_test, y_svd_pred)
precision = precision_score(y_svd_test, y_svd_pred)
recall = recall_score(y_svd_test, y_svd_pred)
f1 = f1_score(y_svd_test, y_svd_pred)
roc_auc = roc_auc_score(y_svd_test, y_svd_pred)

print('#Matriz de confusión:\n ',matrix)
print('#accuracy: ', accuracy)
print('#precision: ', precision)
print('#recall: ', recall)
print('#f1: ', f1)
print('#Curva ROC-AUC: ',roc_auc)
```

```
#Matriz de confusión:
[[38  0]
 [ 3 73]]
#accuracy:  0.9736842105263158
#precision:  0.9736842105263158
#recall:  1.0
#f1:  0.9798657718120806
#Curva ROC-AUC:  0.9802631578947368
```

Naive Bayes

Uno de los algoritmos más simples y muy potente. Podemos ver se basa en el teorema de Bayes asumiéndon la independencia entre predictores. Es fácil de implementar y muy útil para conjuntos de datos muy grandes. Asume que el efecto de una característica al global en una clase es independiente del resto de características. El término de Naive se le da por esto mismo, por asumir esa independencia que no vamos a tener siempre realmente.

$$P(h|D) = P(D|h)P(h)/P(D)$$

$P(h)$ como la probabilidad de que la hipótesis h sea cierta, independientemente de los datos

$P(D)$ como la probabilidad de que la hipótesis D sea cierta, independientemente de los datos

$P(D|h)$ como la probabilidad de la hipótesis D dada h como cierta

$P(h|D)$ como la probabilidad de la hipótesis h dada D como cierta

En caso de que se tenga una sola característica el clasificador de Naive Bayes calcula la probabilidad de un evento en los siguientes pasos:

1º. Calcular la probabilidad previa para las etiquetas de clase dadas

2º. Determinar la probabilidad con cada atributo para cada clase

3º. Poner esos valores en el Teorema de Bayes y calcular esa Probabilidad condicional final

4º. Ver cuál de esas clases tiene una probabilidad más alta

Algunas de sus ventajas:

Es que es fácil y rápido predecir la clase de conjuntos de prueba. También funciona bien en la predicción multiclas

Cuando se mantiene una suposición de independencia, un clasificador Naive Bayes funciona mejor que otros modelos. Por ejemplo, como la regresión logística, y además necesita menos datos de entrenamiento

Funciona bien en el caso de variables de entrada categóricas comparadas con las variables numéricas

Algunas de sus desventajas:

Si aparece una variable categórica en el conjunto test que no se observó en el conjunto train, el modelo asignará una probabilidad 0 por lo que no se podrá hacer una predicción. Esto se conoce como frecuencia cero, y se puede evitar usando la técnica de alisamiento

Otra desventaja es el asumir el que sean independientes, ya que en la realidad casi nunca ocurre

Aún con todas estas desventajas es de los más potentes y usados en ML a día de hoy

Naive Bayes

```
from sklearn.naive_bayes import GaussianNB  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = GaussianNB()
```

Por ser tan básico se ve solo requiere de 2 parámetros, pero no se va a entrar en detalles porque ni siquiera se suelen usar

var_smoothing (optional, default: '1e-9)

priors (array-like, shape) n_classes

```
algoritmo.fit(x_train, y_train)  
  
y_pred = algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [39]:

```
##### Naive Bayes (NB)

import numpy as np
import matplotlib.pyplot as plt
# Implementamos las datasets
from sklearn import datasets

dataset = datasets.load_breast_cancer()
#print(dataset)

#print('dataset_keys: ',dataset.keys())
#print(dataset.DESCR)
#569 instancias y 30 clases o características o atributos

# Tenemos la media, error estándar y el error de esos 30 atributos

# No hay missing values por lo cual ya es una parte del preprocesamiento que nos ahorra
mos

# Tenemos 212 malignos y 357 benignos, por lo cual vemos que tenemos los datos balanceados, no es un dataset desbalanceado

#print('feature_names: ',dataset.feature_names)

X_nb = dataset.data
# Todas las variables del dataset

y_nb = dataset.target

#plt.scatter(X_lr, y_lr)
#plt.show()

from sklearn.model_selection import train_test_split

X_nb_train, X_nb_test, y_nb_train, y_nb_test = train_test_split(X_nb, y_nb, test_size=0.2)

# Lo siguiente a hacer es escalar los datos, ya que las características son diferentes
en magnitudes, unidades y rango, por
# por lo que lo mejor es escalar dichas magnitudes

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_nb_train = scaler.fit_transform(X_nb_train)
X_nb_test = scaler.fit_transform(X_nb_test)

# Datos ya preparados para poder implementar el modelo

from sklearn.naive_bayes import GaussianNB

nb = GaussianNB()

nb.fit(X_nb_train, y_nb_train)

y_nb_pred = nb.predict(X_nb_test)

# Vamos a calcular las métricas de evaluación para ver qué tal, ha ido
from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score

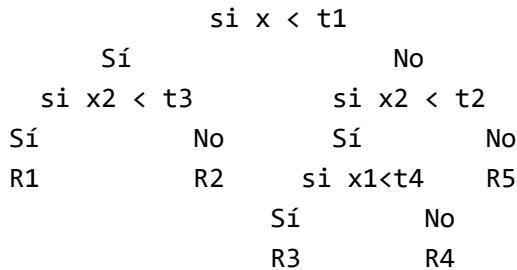
matrix = confusion_matrix(y_nb_test, y_nb_pred)
accuracy = accuracy_score(y_nb_test, y_nb_pred)
precision = precision_score(y_nb_test, y_nb_pred)
recall = recall_score(y_nb_test, y_nb_pred)
f1 = f1_score(y_nb_test, y_nb_pred)
roc_auc = roc_auc_score(y_nb_test, y_nb_pred)

print('#Matriz de confusión:\n ',matrix)
print('#accuracy: ', accuracy)
print('#precision: ', precision)
print('#recall: ', recall)
print('#f1: ', f1)
print('#Curva ROC-AUC: ',roc_auc)
```

```
#Matriz de confusión:
[[43  4]
 [ 4 63]]
#accuracy:  0.9298245614035088
#precision:  0.9298245614035088
#recall:  0.9402985074626866
#f1:  0.9402985074626865
#Curva ROC-AUC:  0.9275960622419815
```

Árboles de Decisión. Clasificación

Los Árboles de Decisión de ML son uno de los algoritmos más usados en ML ya que es fácil de ver lo que está sucediendo. Recordemos las estructuras se basaban en Condiciones y Ramas, la idea era la siguiente: Seleccionar el mejor atributo usando una selección de atributos y ver aquellos que tengan las mejores características, hacer de esos atributos nodos de decisión y así construir el árbol de decisión. Sería algo como lo ya visto antes, pero de forma gráfica y más clara:



Y así se puede construir el árbol de forma recursiva hasta que una de las siguientes condiciones coincida y venga todas las variables por ello pertenezcan a ese mismo atributo y no se pueda seguir dividiendo.

La medida de selección de datos es una heurística, para seleccionar el criterio de selección de división que separa los datos de la mejor forma posible, también son reglas de partición, reglas de ruptura para un nodo dado. Esta medida proporciona un rango a cada característica, explicando el conjunto de datos dado, el atributo de mejor puntuación será elegido como el atributo de división. En el caso de un atributo de valor continuo también es necesario hacer puntos de división por las ramas.

Las medidas de selección más populares son la ganancia de información, la relación de ganancia y el índice de Gini.

Ganancia de información: Cuando usamos un nodo en un Árbol de Decisión, para partitionar las instancias de formación, vemos la entropía cambia. La ganancia de información da medida de este cambio de información/entropía que es la medida de la incertidumbre de una variable aleatoria, caracterizando la impureza de una colección arbitraria de ejemplos. A mayor entropía, mayor información. Para construir un árbol de información tendremos que comenzar usando todas las instancias de información asociadas al nodo raíz, utilizar la ganancia de información para elegir qué atributo etiquetar cada nodo con cuál y construir cada subárbol en el subconjunto de instancias de capacitación que se clasificarían en ese camino en el árbol. Nunca se debe tener dos veces de raíz a hoja el mismo atributo.

Índice Gini: Es una métrica para medir la frecuencia con la que un elemento elegido al azar sería clasificado incorrectamente. Esto indica que se debe elegir un índice de Gini bajo.

Ventajas de Árboles de Decisión:

Los Árboles de Decisión son fáciles de interpretar y visualizar, y pueden capturar patrones no lineales.

Requiere menos preprocesamiento por parte del usuario. No se requiere por ejemplo de normalizar las columnas.

Se puede utilizar como ingeniería de características como para la predicción de variables perdidos, adecuada para la selección de variables.

No tiene suposiciones sobre la forma de la distribución debido a la naturaleza no paramétrica del árbol.

Desventajas de Árboles de Decisión:

Datos sensibles al ruido, pueden sobredimensionar esos datos ruidosos.

Una pequeña variación en los datos puede llevar a un Árbol de Decisión diferente.

Árboles de Decisión. Clasificación

```
from sklearn.tree import DecisionTreeClassifier  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = DecisionTreeClassifier()
```

criterion (default: 'gini'), se define la medida de selección para dividir los datos de la manera mejor posible, otras opciones son 'entropy'

split (default: 'best'), es para la división en cada nodo, tenemos dos opciones 'best' y 'random', es mejor 'best'

max_depth(default=None), para ver la máxima profundidad, por defecto va a ir hasta el final, clasificando todos los posibles y es lo más seguro que obtengamos un overfitting y eso no lo queremos, tenemos que ir variando el valor y buscar el que mejor se ajuste

```
algoritmo.fit(x_train, y_train)  
  
y_pred = algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [40]:

```
##### Árboles de Decisión Clasificación

import numpy as np
import matplotlib.pyplot as plt
# Implementamos las datasets
from sklearn import datasets

dataset = datasets.load_breast_cancer()
#print(dataset)

#print('dataset_keys: ',dataset.keys())
#print(dataset.DESCR)
#569 instancias y 30 clases o características o atributos

# Tenemos la media, error estándar y el error de esos 30 atributos

# No hay missing values por lo cual ya es una parte del preprocesamiento que nos ahorra
mos

# Tenemos 212 malignos y 357 benignos, por lo cual vemos que tenemos los datos balanceados, no es un dataset desbalanceado

#print('feature_names: ',dataset.feature_names)

X_dtc = dataset.data
# Todas las variables del dataset

y_dtc = dataset.target

#plt.scatter(X_lr, y_lr)
#plt.show()

from sklearn.model_selection import train_test_split

X_dtc_train, X_dtc_test, y_dtc_train, y_dtc_test = train_test_split(X_dtc, y_dtc, test_
size=0.2)

# Lo siguiente a hacer es escalar los datos, ya que las características son diferentes
en magnitudes, unidades y rango, por
# por lo que lo mejor es escalar dichas magnitudes

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_dtc_train = scaler.fit_transform(X_dtc_train)
X_dtc_test = scaler.fit_transform(X_dtc_test)

# Datos ya preparados para poder implementar el modelo

from sklearn.tree import DecisionTreeClassifier

dtc = DecisionTreeClassifier(criterion = 'gini')

dtc.fit(X_dtc_train, y_dtc_train)

y_dtc_pred = dtc.predict(X_dtc_test)

# Vamos a calcular las métricas de evaluación para ver qué tal, ha ido
from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score

matrix = confusion_matrix(y_dtc_test, y_dtc_pred)
accuracy = accuracy_score(y_dtc_test, y_dtc_pred)
precision = precision_score(y_dtc_test, y_dtc_pred)
recall = recall_score(y_dtc_test, y_dtc_pred)
f1 = f1_score(y_dtc_test, y_dtc_pred)
roc_auc = roc_auc_score(y_dtc_test, y_dtc_pred)

print('#Matriz de confusión:\n ',matrix)
print('#accuracy: ', accuracy)
print('#precision: ', precision)
print('#recall: ', recall)
print('#f1: ', f1)
print('#Curva ROC-AUC: ',roc_auc)

#DecisionTreeClassifier(criterion = 'entropy', max_depth = 5)

#Matriz de confusión:
# [[43  8]
# [ 1 62]]
#accuracy:  0.9210526315789473
#precision:  0.9210526315789473
#recall:  0.8857142857142857
#f1:  0.9323308270676691
#Curva ROC-AUC:  0.9136321195144725

#DecisionTreeClassifier(criterion = 'entropy')

#Matriz de confusión:
# [[38  5]
# [ 4 67]]
#accuracy:  0.9210526315789473
#precision:  0.9210526315789473
#recall:  0.9305555555555556
#f1:  0.9370629370629372
#Curva ROC-AUC:  0.9136914510317721

#DecisionTreeClassifier(criterion = 'gini')

#Matriz de confusión:
# [[35  4]
# [ 3 72]]
#accuracy:  0.9385964912280702
#precision:  0.9385964912280702
#recall:  0.9473684210526315
#f1:  0.9536423841059603
#Curva ROC-AUC:  0.9287179487179487
```

```
#Matriz de confusión:  
[[38  2]  
 [ 4 70]]  
#accuracy: 0.9473684210526315  
#precision: 0.9473684210526315  
#recall: 0.9722222222222222  
#f1: 0.9589041095890412  
#Curva ROC-AUC: 0.947972972972973
```

Bosques Aleatorios Clasificación

Los Bosques Aleatorios es un algoritmo de aprendizaje supervisado, es usado tanto en regresión como en clasificación. Es el algoritmo más flexible y fácil de usar. Es un conjunto de Árboles de Decisión, se dice a más se tenga, más robusto va a ser el modelo. Se crean Árboles de Decisión a partir de muestras del conjunto seleccionadas al azar y se obtiene predicciones de cada árbol y selecciona la mejor solución mediante votaciones. Selecciona un indicador bastante bueno de la característica.

Técnicamente es un método de conjuntos basado en el enfoque de dividir y conquistar en un conjunto de Árboles de Decisión generados usando un indicador de atributos como la ganancia de información, la relación de ganancia o el índice de Gini. Realmente vemos cada árbol depende de una muestra aleatoria independiente, en cada árbol por individual se elige la clase más popular al votar. Es más simple y más potente que otros algoritmos de clasificación.

Funciona de la siguiente forma:

Construir un Árbol de Decisión para cada muestra y obtener un resultado de predicción para cada Árbol

Realizar una predicción por cada resultado previsto

Seleccionar el resultado de la predicción con más votos como predicción final

Importancia de las características, poder ver las importancias de las características en cada nodo y podemos ver los árboles que utilizan estas características reducen la impureza en todos los árboles del bosque. Se calcula la puntuación para todos los árboles y se normaliza su peso en función de la importancia, y así tenemos todas normalizadas de 0 a 1 y así se puede reducir dimensiones, ya que así se reducirán costes o posibles fallos.

Los Bosques Aleatorios evitan el overfitting creando Árboles de Decisión de subconjuntos de datos. Los Árboles de decisión por otro lado son más baratos de coste temporal y más sencillos de interpretar ya que nos dan reglas.

Ventajas de los Bosques Aleatorios:

Son más robustos por no tener ese overfitting ya que toma el promedio de todas las predicciones por lo que anula los sesgos.

Pueden lidiar también con los missing values ya que pueden usar los valores medios para reemplazar esas variables continuas o para categóricos calculando los promedios ponderados por proximidad de los valores faltantes.

Desventajas de los Bosques Aleatorios:

Son lentos en hacer predicciones ya que cada vez que hace una predicción, este es el promedio de las predicciones de todos los Árboles de Decisión.

Modelo más difícil de interpretar que un Árbol de Decisión ya que en ellos solo hay que seguir la ruta de dicho árbol.

Árboles de Decisión. Clasificación

```
from sklearn.ensemble import RandomForestClassifier  
  
x_train = variablesIndependientesEntrenamiento  
  
y_train = variablesDependientesEntrenamiento  
  
x_test = variablesIndependientesTest  
  
y_test = variablesDependientesTest  
  
algoritmo = RandomForestClassifier()
```

n_estimators el número de árboles que este construye. Por lo general un mayor número de Árboles garantiza un mayor rendimiento y más robusto pero también ralentiza el cálculo

max_features (default='auto') el número máximo de características (muestras o datos vaya) que un algoritmo utiliza para separar un nodo, otras opciones son None, 'auto', 'sqrt', 'log2'

criterion (default: 'gini'), se define la medida de selección para dividir los datos de la manera mejor posible, otras opciones son 'entropy'

split (default: 'best'), es para la división en cada nodo, tenemos dos opciones 'best' y 'random', es mejor 'best'

max_depth(default=None), para ver la máxima profundidad, por defecto va a ir hasta el final, clasificando todos los posibles y es lo más seguro que obtengamos un overfitting y eso no lo queremos, tenemos que ir variando el valor y buscar el que mejor se ajuste

```
algoritmo.fit(x_train, y_train)  
  
y_pred = algoritmo.predict(x_test)
```

Finalmente hay que verificar el rendimiento del algoritmo ya que es muy importante para ver qué tan

bien funcionan un conjunto de datos con este algoritmo en particular

```
precision = algoritmo.score(x_test, y_test)
```

In [41]:

```
##### Bosques Aleatorios Clasificación

import numpy as np
import matplotlib.pyplot as plt
# Implementamos las datasets
from sklearn import datasets

dataset = datasets.load_breast_cancer()
#print(dataset)

#print('dataset_keys: ',dataset.keys())
#print(dataset.DESCR)
#569 instancias y 30 clases o características o atributos

# Tenemos la media, error estándar y el error de esos 30 atributos

# No hay missing values por lo cual ya es una parte del preprocesamiento que nos ahorra
mos

# Tenemos 212 malignos y 357 benignos, por lo cual vemos que tenemos los datos balanceados, no es un dataset desbalanceado

#print('feature_names: ',dataset.feature_names)

X_rfc = dataset.data
# Todas las variables del dataset

y_rfc = dataset.target

#plt.scatter(X_lr, y_lr)
#plt.show()

from sklearn.model_selection import train_test_split

X_rfc_train, X_rfc_test, y_rfc_train, y_rfc_test = train_test_split(X_rfc, y_rfc, test_
size=0.2)

# Lo siguiente a hacer es escalar los datos, ya que las características son diferentes
en magnitudes, unidades y rango, por
# por lo que lo mejor es escalar dichas magnitudes

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_rfc_train = scaler.fit_transform(X_rfc_train)
X_rfc_test = scaler.fit_transform(X_rfc_test)

# Datos ya preparados para poder implementar el modelo

from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier(n_estimators = 10, criterion = 'gini')

rfc.fit(X_rfc_train, y_rfc_train)

y_rfc_pred = rfc.predict(X_rfc_test)

# Vamos a calcular las métricas de evaluación para ver qué tal, ha ido
from sklearn.metrics import confusion_matrix
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score

matrix = confusion_matrix(y_rfc_test, y_rfc_pred)
accuracy = accuracy_score(y_rfc_test, y_rfc_pred)
precision = precision_score(y_rfc_test, y_rfc_pred)
recall = recall_score(y_rfc_test, y_rfc_pred)
f1 = f1_score(y_rfc_test, y_rfc_pred)
roc_auc = roc_auc_score(y_rfc_test, y_rfc_pred)

print('#Matriz de confusión:\n ',matrix)
print('#accuracy: ', accuracy)
print('#precision: ', precision)
print('#recall: ', recall)
print('#f1: ', f1)
print('#Curva ROC-AUC: ',roc_auc)
```

```
#Matriz de confusión:
[[44  1]
 [ 2 67]]
#accuracy:  0.9736842105263158
#precision:  0.9736842105263158
#recall:  0.9852941176470589
#f1:  0.9781021897810219
#Curva ROC-AUC:  0.9743961352657003
```

Ventajas y Desventajas de los Algoritmos de Clasificación

Debemos tener en cuenta ningún algoritmo es el mejor en todos los problemas, cada uno se debe usar en un contexto dado, y todos tienen sus pros y cons. Vamos a ver algunos criterios por los cuales podríamos ver cuál es el mejor a elegir en cada caso. Recordemos son:

- Regresión Logística: El análisis de regresión es el más apropiado cuando tenemos una variable objetivo binaria. Como todos los análisis de regresión, es un análisis predictivo. Sus ventajas son que es fácil de entender, rara vez hay overfitting, y el uso de la regularización es útil a la hora de la selección de funciones, rápido de entrenar y fácil de entrenar debido a su versión estocástica. Desventajas son que tienes que trabajar duro para que se ajuste a valores no lineales, sufre con valores atípicos y en algunas ocasiones es útil para captar relaciones complejas entre variables. Son útiles para ordenar los resultados por probabilidad.

-KNN (K-Nearest Neighbors): Es un algoritmo muy simple, fácil de entender, versátil, basado en la similitud de características. Es un algoritmo no paramétrico, no hace suposiciones de la distribución de datos adyacentes, es decir, no necesita puntos de datos de train para la generación de modelos. Ventajas son que es simple, potente, entrenamiento rápido y puede manejar naturalmente problemas extremos multiclases. Desventajas son que es costoso y lentos para predecir nuevas instancias, se debe definir una función de distancia significativa y funciona mal en datos de alta dimensionalidad. Se usan en seguridad informática (detección de intrusos), sistemas de recomendación y problemas de corrección ortográficas.

- Vectores de Soporte de Clasificación: Se basa en la construcción de un hiperplano óptimo en forma de superficie de decisión de manera que los márgenes a los lados se amplíen al máximo, los vectores de soporte hacen referencia a un subconjunto de las observaciones de train y poder hacer las descripciones o predicciones. Algunas de sus ventajas son poder modelar relaciones complejas no lineales y robustez al ruido (maximizar los márgenes C). Y presenta también algunas desventajas como la necesidad de seleccionar buenas funciones kernel y con una apropiada elección de parámetros, los parámetros son más difíciles de interpretar, requiere alta memoria y poder de preprocessamiento por lo que toma demasiado tiempo para entrenar. Suelen ser útiles para la clasificación de texto e imágenes y también para el reconocimiento de escritura a mano.

- Naive Bayes: Es un modelo de ML que se usa para clasificación basándose en el Teorema de Bayes. Como ventajas vemos es fácil y rápido de implementar, no requiere demasiada memoria y se puede aprovechar para el aprendizaje en línea y es fácil de entender. Algunas de las desventajas es que falla al tener características raras o irrelevantes. Son útiles en el reconocimiento de rostros, análisis de

NEURAL NETWORKS / DEEP LEARNING

Son unos algoritmos que nos van a devolver datos y de tal forma que van a devolvernos un resultado que será aquel obtenido de hacer operaciones con ese input. El output puede ser desde una variable continua, un valor categórico, una probabilidad... Un Neural Network (NN) va a ser una forma de calcular lo que queremos. Toma su nombre por la forma de la estructura del esquema y su similitud a un cerebro.

Se trata de algoritmos muy flexibles y se pueden complicar muchos. Se puede explicar como basándonos en Linear Regression, donde tenemos los inputs como esas variables independientes como cada nodo de nuestra Red Neuronal, como se hacía antes, veíamos queríamos relacionarlos con una variable dependiente o target. De esta forma sabemos que nuestro output sería dicho target, y tendríamos que darle un cierto peso o importancia, (weight) a cada uno de esos inputs, y también van a estar relacionados por medio de un "bias" que va a representar la intersección entre ellos. El output es la combinación de cada input con sus weights y con el bias.

Lo siguiente que nos podemos preguntar es cómo estamos entrenando a nuestro NN para que nos clasifique, para ver cómo hace dicha predicción. Para cuantificar la calidad de la predicción, como era en el caso de Linear Regression teníamos la suma de errores cuadráticos o varianza y ahora, queremos ver en nuestro caso de NN tendremos una función de pérdida análoga (Loss Function) de tal forma que lo que hace nuestro NN es que recorre el dataset, e itera el espacio de posibilidades en búsqueda de una minimización de esta Loss Function.

Además, el output tiene asociada una función de activación que nos da cuenta de cómo de buena es la estimación en función del valor verdadero, que depende de cómo sea el tipo de predicción en la que nos estemos basando, si se trata de Linear Regression veremos no tiene esa activation function, mientras que Logistic Regression tendrá una sigmoide.

Obviamente se puede construir NN con modelos más complicados que los mencionados. En estos casos sencillos sabemos que tenemos una capa o layer de inputs y una layer de outputs, y con sus weights. Sin embargo se pueden tener múltiples capas, se puede ir de input (1^a capa) a otras capas y después a, finalmente, el output. Esto es el llamado Deep Learning, cuando se tienen "Multiple Layers Neural Networks". También hay que pensar en cómo están conectadas esas capas, eso son las "dense connections", todos los nodos de una capa conectados con cada uno de los nodos de la siguiente capa, aunque no tienen por qué estar conectados todos con todos. Como bien sabíamos, tenemos de una capa a la siguiente una serie de combinaciones matemáticas en esa capa previa y los nodos seleccionados de tal forma que el nodo origen o sub-output va a tener su propia activation function y tendremos que centrarnos en algunos de ellos. Sabemos también vamos a tener una gran variabilidad a la hora de obtener dichas funciones de activación y lo que querremos ver serán esas funciones según nuestro output deseado y por esta gran flexibilidad en número de capas, cuántas conexiones y cómo hacerlas... Por ello estos problemas no siempre serán sencillos ya que no hay buenos criterios o absolutos, sino reglas del pulgar o similares.

In [42]:

```
##### Neural Network (Basic Level: Linear Regression)

import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Para instalar tensorflow
#!pip3 install --upgrade tensorflow --ignore-installed

import tensorflow as tf
import tensorflow.keras as kb
from tensorflow.keras import backend

from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing

# Se va a realizar una NN para un caso sencillo en el que el input y el output se relacionen mediante Linear Regression

data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Wholesale customers data.csv')

data.head()

# Vamos a asumir que queremos trabajar con Fresh, Milk, Grocery, Frozen y Detergents_Paper para predecir Delicassen

X_data_names = ['Fresh', 'Milk', 'Grocery', 'Frozen', 'Detergents_Paper']
y_data_names = ['Delicassen']

X = data[X_data_names]
y = data[y_data_names]

# Convertimos a array tipo numpy para poder meterlo en el modelo
#X_np = X.values.astype('float32', copy = False)
#y_np = y.values.astype('float32', copy = False)

# Normalizamos los datos
scaled_X = preprocessing.Normalizer().fit_transform(X)#_np)
scaled_y = preprocessing.Normalizer().fit_transform(y)#_np)

# Se define el modelo en keras:

# 1. Estructura del modelo
model = kb.Sequential([
    kb.layers.Dense(5, input_shape = [5]),
    kb.layers.Dense(1)
])
# recordemos que 'kb' es tensorflow.keras
# 'Sequential()' es para decirle que cada capa va directamente a la siguiente
```

```

# 'kb.layers.Dense(5, input_shape = [5])' será La primera capa:
# 'Dense' es para que cada punto de la capa anterior vaya a la siguiente e interaccione todos los nodos de una con la otra
# Ese 5 es por el número de nodos en el input layer (Fresh, Milk, Grocery, Frozen y Detractives_Paper)
# input_shape = [5] como el número de nodos que estamos esperando y es solo necesario para el primer input

# 'kb.layers.Dense(1)' será La segunda capa, completamente análoga al caso anterior

# 2. Entrenamos el modelo
model.compile(loss = "mean_squared_error",
               optimizer = kb.optimizers.SGD())
# Compilar el modelo
# loss es la loss_function, se puede minimizar el error de mínimos cuadrados, el error de mínimos en la media, ...
# optimizer es el optimizador y ese SGD da cuenta de Stochastic Gradient Descent

# 3. Fit el modelo
model.fit(scaled_X, scaled_y, epochs = 100)
# epochs es las veces que busca e itera en los datos, por lo general, a mayor epochs, mejor, pero para modelos sencillos no
# hace falta

# Se puede ver este loss: va decreciendo con cada epoch hasta que se llega al final y entrenamos el modelo
# Así pues tenemos ya un NN funcional y podemos ver lo podemos usar como hacíamos en regresión para hacer predicciones
# Además el Linear Regression está implementado por la forma en la que hemos conectado las capas entre sí

# Vamos a ver cómo predice nuestro modelo

# X_n = scaled_X.to_numpy()
# en caso de seguir con un tipo pandas dataframe

y_pred = model.predict(scaled_X).flatten()
# 'model.predict(X_n)' esto únicamente nos daría un array
# con 'flatten()', nos da una única lista

#true_pred = pd.DataFrame[{'true': y, 'pred': y_pred}]

```

```
Epoch 1/100
14/14 [=====] - 0s 2ms/step - loss: 0.5844
Epoch 2/100
14/14 [=====] - 0s 2ms/step - loss: 0.1798
Epoch 3/100
14/14 [=====] - 0s 2ms/step - loss: 0.1233
Epoch 4/100
14/14 [=====] - 0s 1ms/step - loss: 0.0960
Epoch 5/100
14/14 [=====] - 0s 687us/step - loss: 0.0768
Epoch 6/100
14/14 [=====] - 0s 998us/step - loss: 0.0624
Epoch 7/100
14/14 [=====] - 0s 571us/step - loss: 0.0510
Epoch 8/100
14/14 [=====] - 0s 572us/step - loss: 0.0421
Epoch 9/100
14/14 [=====] - 0s 1ms/step - loss: 0.0349
Epoch 10/100
14/14 [=====] - 0s 1ms/step - loss: 0.0293
Epoch 11/100
14/14 [=====] - 0s 1ms/step - loss: 0.0246
Epoch 12/100
14/14 [=====] - 0s 571us/step - loss: 0.0209
Epoch 13/100
14/14 [=====] - 0s 1ms/step - loss: 0.0179
Epoch 14/100
14/14 [=====] - 0s 571us/step - loss: 0.0154
Epoch 15/100
14/14 [=====] - 0s 923us/step - loss: 0.0133
Epoch 16/100
14/14 [=====] - 0s 571us/step - loss: 0.0116
Epoch 17/100
14/14 [=====] - 0s 0s/step - loss: 0.0102
Epoch 18/100
14/14 [=====] - 0s 940us/step - loss: 0.0090
Epoch 19/100
14/14 [=====] - 0s 1ms/step - loss: 0.0080
Epoch 20/100
14/14 [=====] - 0s 1ms/step - loss: 0.0072
Epoch 21/100
14/14 [=====] - 0s 895us/step - loss: 0.0065
Epoch 22/100
14/14 [=====] - 0s 1ms/step - loss: 0.0059
Epoch 23/100
14/14 [=====] - 0s 1ms/step - loss: 0.0054
Epoch 24/100
14/14 [=====] - 0s 804us/step - loss: 0.0050
Epoch 25/100
14/14 [=====] - 0s 1ms/step - loss: 0.0046
Epoch 26/100
14/14 [=====] - 0s 1ms/step - loss: 0.0043
Epoch 27/100
14/14 [=====] - 0s 572us/step - loss: 0.0040
Epoch 28/100
14/14 [=====] - 0s 1ms/step - loss: 0.0038
Epoch 29/100
14/14 [=====] - 0s 571us/step - loss: 0.0036
Epoch 30/100
14/14 [=====] - 0s 572us/step - loss: 0.0034
Epoch 31/100
```

```
14/14 [=====] - 0s 1ms/step - loss: 0.0032
Epoch 32/100
14/14 [=====] - 0s 572us/step - loss: 0.0031
Epoch 33/100
14/14 [=====] - 0s 572us/step - loss: 0.0030
Epoch 34/100
14/14 [=====] - 0s 572us/step - loss: 0.0029
Epoch 35/100
14/14 [=====] - 0s 819us/step - loss: 0.0028
Epoch 36/100
14/14 [=====] - 0s 571us/step - loss: 0.0027
Epoch 37/100
14/14 [=====] - 0s 1ms/step - loss: 0.0026
Epoch 38/100
14/14 [=====] - 0s 572us/step - loss: 0.0025
Epoch 39/100
14/14 [=====] - 0s 659us/step - loss: 0.0025
Epoch 40/100
14/14 [=====] - 0s 571us/step - loss: 0.0024
Epoch 41/100
14/14 [=====] - 0s 571us/step - loss: 0.0024
Epoch 42/100
14/14 [=====] - 0s 1ms/step - loss: 0.0023
Epoch 43/100
14/14 [=====] - 0s 431us/step - loss: 0.0022
Epoch 44/100
14/14 [=====] - 0s 0s/step - loss: 0.0022
Epoch 45/100
14/14 [=====] - 0s 617us/step - loss: 0.0022
Epoch 46/100
14/14 [=====] - 0s 571us/step - loss: 0.0021
Epoch 47/100
14/14 [=====] - 0s 639us/step - loss: 0.0021
Epoch 48/100
14/14 [=====] - 0s 572us/step - loss: 0.0020
Epoch 49/100
14/14 [=====] - 0s 1ms/step - loss: 0.0020
Epoch 50/100
14/14 [=====] - 0s 571us/step - loss: 0.0020
Epoch 51/100
14/14 [=====] - 0s 572us/step - loss: 0.0019
Epoch 52/100
14/14 [=====] - 0s 1ms/step - loss: 0.0019
Epoch 53/100
14/14 [=====] - 0s 635us/step - loss: 0.0019
Epoch 54/100
14/14 [=====] - 0s 2ms/step - loss: 0.0018
Epoch 55/100
14/14 [=====] - 0s 1ms/step - loss: 0.0018
Epoch 56/100
14/14 [=====] - 0s 572us/step - loss: 0.0018
Epoch 57/100
14/14 [=====] - 0s 571us/step - loss: 0.0017
Epoch 58/100
14/14 [=====] - 0s 1ms/step - loss: 0.0017
Epoch 59/100
14/14 [=====] - 0s 634us/step - loss: 0.0017
Epoch 60/100
14/14 [=====] - 0s 571us/step - loss: 0.0017
Epoch 61/100
14/14 [=====] - 0s 571us/step - loss: 0.0016
```

```
Epoch 62/100
14/14 [=====] - 0s 1ms/step - loss: 0.0016
Epoch 63/100
14/14 [=====] - 0s 1ms/step - loss: 0.0016
Epoch 64/100
14/14 [=====] - 0s 1ms/step - loss: 0.0016
Epoch 65/100
14/14 [=====] - 0s 1ms/step - loss: 0.0016
Epoch 66/100
14/14 [=====] - 0s 572us/step - loss: 0.0015
Epoch 67/100
14/14 [=====] - 0s 572us/step - loss: 0.0015
Epoch 68/100
14/14 [=====] - 0s 572us/step - loss: 0.0015
Epoch 69/100
14/14 [=====] - 0s 572us/step - loss: 0.0015
Epoch 70/100
14/14 [=====] - 0s 253us/step - loss: 0.0015
Epoch 71/100
14/14 [=====] - 0s 620us/step - loss: 0.0014
Epoch 72/100
14/14 [=====] - 0s 1ms/step - loss: 0.0014
Epoch 73/100
14/14 [=====] - 0s 1ms/step - loss: 0.0014
Epoch 74/100
14/14 [=====] - 0s 1ms/step - loss: 0.0014
Epoch 75/100
14/14 [=====] - 0s 572us/step - loss: 0.0014
Epoch 76/100
14/14 [=====] - 0s 895us/step - loss: 0.0014
Epoch 77/100
14/14 [=====] - 0s 638us/step - loss: 0.0013
Epoch 78/100
14/14 [=====] - 0s 288us/step - loss: 0.0013
Epoch 79/100
14/14 [=====] - 0s 571us/step - loss: 0.0013
Epoch 80/100
14/14 [=====] - 0s 1ms/step - loss: 0.0013
Epoch 81/100
14/14 [=====] - 0s 572us/step - loss: 0.0013
Epoch 82/100
14/14 [=====] - 0s 1ms/step - loss: 0.0013
Epoch 83/100
14/14 [=====] - 0s 467us/step - loss: 0.0013
Epoch 84/100
14/14 [=====] - 0s 571us/step - loss: 0.0012
Epoch 85/100
14/14 [=====] - 0s 572us/step - loss: 0.0012
Epoch 86/100
14/14 [=====] - 0s 571us/step - loss: 0.0012
Epoch 87/100
14/14 [=====] - 0s 1ms/step - loss: 0.0012
Epoch 88/100
14/14 [=====] - 0s 571us/step - loss: 0.0012
Epoch 89/100
14/14 [=====] - 0s 906us/step - loss: 0.0012
Epoch 90/100
14/14 [=====] - 0s 633us/step - loss: 0.0012
Epoch 91/100
14/14 [=====] - 0s 572us/step - loss: 0.0012
Epoch 92/100
```

```
14/14 [=====] - 0s 1ms/step - loss: 0.0012
Epoch 93/100
14/14 [=====] - 0s 1ms/step - loss: 0.0011
Epoch 94/100
14/14 [=====] - 0s 1ms/step - loss: 0.0011
Epoch 95/100
14/14 [=====] - 0s 572us/step - loss: 0.0011
Epoch 96/100
14/14 [=====] - 0s 1ms/step - loss: 0.0011
Epoch 97/100
14/14 [=====] - 0s 1ms/step - loss: 0.0011
Epoch 98/100
14/14 [=====] - 0s 571us/step - loss: 0.0011
Epoch 99/100
14/14 [=====] - 0s 572us/step - loss: 0.0011
Epoch 100/100
14/14 [=====] - 0s 572us/step - loss: 0.0011
```

Como hemos podido ver hay una serie de conceptos importantes, como bien pueden ser las funciones de pérdida o las funciones optimizadoras. Algunos conceptos importantes de entender son los siguientes:

Gradient Descent

Una técnica de Machine Learning de optimización que busca encontrar el set óptimo de valores para unos parámetros para un problema dado, es decir, vamos a estar trabajando con esta función de coste o de error del sistema de aprendizaje o modelo. Así, lo que realmente estaremos viendo será el balance de cómo de cercanos serán los resultados predichos por este modelo frente a los resultados reales en función de un determinado conjunto de parámetros.

Inicialmente se elige aleatoriamente una combinación de ese espacio de parámetros, se medirá el error que esa medida inicial produce en nuestro sistema, e ir variando dicha combinación de parámetros de forma ligera buscando reducir ese error. Se sabe que en términos de Inteligencia Artificial o Machine Learning no es de gran importancia los mínimos relativos o locales, por lo que este método de ir buscando ese set óptimo de parámetros no tendrá problema. La forma de ver cómo se sabe cuándo se está yendo hacia abajo, es decir, se puede evaluar por medio del gradiente, que da cuenta de la pendiente de esa curva de error, es decir, por medio de esas derivadas parciales.

Autodiff

Es una técnica para acelerar ese proceso ya que si toca hacer esas derivadas en todo el espacio de posibles parámetros es demasiado costoso computacionalmente. Todo esto está implementado ya en tensorflow por lo que no hay problema.

Softmax

Cuando tenemos el output de un NN lo que sabemos es que tenemos ese output y una serie de weights. Softmax convierte cada uno de esos pesos en probabilidades, por ejemplo en un problema de clasificación, probabilidades de pertenencia a cada una de las clases posibles y entonces elegir aquella que mayor probabilidad tenga y así poder hacer una clasificación.

Linear Threshold Unit (LTU) and Perceptron

LTU: Inputs con unos determinados pesos y el output va a ser una función escalón ya que tendremos pesos positivos y negativos y serán de tal forma que serán los que nos den esa función escalón, y por lo tanto podremos ver que al ser esa función escalón será como un interruptor.

El perceptrón realmente será una layer de LTU's conectados de tal forma que podamos unir los pesos para intentar reproducir el comportamiento o output deseado, y estaremos buscando recompensar a aquellos que mejor clasifiquen, podremos ver aquí dónde se introduce el concepto del bias. El bias es un valor constante de tal forma que también podremos cambiar buscando optimizarla.

El siguiente nivel de dificultad sería el perceptrón multicapa o con capas escondidas añadidas, ya que están en un punto medio que no es ni el input ni el output, esto, es Deep Learning. Estos son complejos de entrenar y la dificultad como se ha mencionado surge en optimizar cuáles, cuántas y cómo hacer estas conexiones ya que esta gran multitud de posibilidades en las conexiones va a hacer que surja toda esta dificultad.

Modern Deep Neural Network (Multi-Layer Perceptron, MLP)

Reemplazar esas funciones de activación que habíamos asumido como constantes por funciones escalón por otras funciones (relu), ya que a veces funciones con otras formas o pendientes que harán converger el problema de forma más rápida a nuestra NN y veremos aplicaremos softmax al output y finalmente podremos usar algún criterio de entrenamiento como bien podría ser ese gradiente descendente u alguna combinación. Además, podremos usar también Autodiff para hacer ese entrenamiento más eficiente.

Para adaptarse a diferentes formas complejas tendremos que aumentar el número de hidden layers o de inputs o de nodos por cada layer.

¿Cómo se entrena un Multi-Layer Perceptron (MLP)? Se hace por medio de una técnica llamada Back-Propagation, que realmente es gradiente descendente con el truco para optimizar que es con ese truco de modo reverso (el autodiff mencionado). Por cada iteración de entrenamiento, calculamos el error en el output para los weights de cada una de las conexiones y aquí es donde aparece ese Back-Propagation, calcular la suma de esos errores y buscar el gradiente en dirección descendente de tal forma que se busque un cambio en los parámetros de los pesos para esa siguiente iteración y así se pueda obtener un mejor modelo que en la iteración anterior ya que se ha reducido dicho error en el output. Habrá que iterar dicho proceso hasta que encontremos el valor mínimo en este nuestro mapa de espacio de configuraciones de posibles combinaciones de parámetros. Iteraciones serán realmente las épocas o epochs.

Activation Functions (aka Rectifier)

La Activation Function es aquella que determina el output dada la suma (teniendo en cuenta los pesos) de esos inputs, por ser el output veremos será realmente donde tendremos que buscar también el gradient descent pero tendremos un problema en las funciones escalón, ya que estas no tienen pendiente. Sin embargo, vamos a tener alternativas como bien van a ser

- Logistic function
- Hyperbolic tangent function (curvy)
- Exponential Linear Unit (ELU) (aún más curvy)
- Rectified Linear Unit (ReLU) (números negativos con valor nulo y en los positivos una línea recta comenzando en el origen con un ángulo de 45º) ReLU va a ser una muy buena opción y muy común por ser muy rápida de implementar y computacionalmente barata, buena opción si nos preocupa el tiempo o recursos de ejecución y la convergencia.

Si no nos importa tanto la eficiencia vamos a ver tenemos variantes como son "Leaky ReLU", que en vez de ser nula en los números negativos tendrá un valor de pendiente pequeño, todo esto con base matemática detrás en la que no hace falta entrar pero obviamente está relacionado con la pendiente y los gradientes. Otra opción similar es "Noisy ReLU". Sin embargo, también obtendremos entrenamientos rápidos con ELU.

Optimization Functions

Hay más opciones y más rápidas que Gradient Descent para el entrenamiento y convergencia del modelo. Algunas de estas funciones son variaciones del propio Gradient Descent y podemos ver algunas de ellas:

- Momentum Optimization: la idea es descender de manera menos abrupta cuando la pendiente es descendiente pero está cerca de ser plana, es decir, cuando nos estamos aproximando al mínimo.
- Nesterov Accelerated Gradient: un retoque en Momentum Optimization, que busca más allá del gradiente, es decir, la información justo más adelante para poder implementarlo de forma previa.
- RMSProp: se basa en un ratio de aprendizaje adaptativo que nos ayuda, como el de Nesterov Accelerated Gradient, a poder apuntar en la dirección correcta hacia el mínimo.
- ADAM: ADAptive Moment Estimation, que realmente es una combinación de Momentum Optimization y RMSProp, combinando lo mejor de los dos mundos, por lo que es una opción muy común actualmente y además es muy fácil de usar.

Avoiding overfitting

Con millones de pesos/conexiones entre neuronas a ir modificando vamos a ver va a ser altamente probable que ocurra un overfitting, de esta forma veremos formas de evitarlo:

- Early Stopping: a medida que vemos cuando la performance empieza a droppearse, es que está aprendiendo demasiado, lo cual es overfitting y no lo queremos
- Regularization terms: se pueden añadir a la Cost Function en la fase de training y esto va a ser como el bias
- Dropout: una técnica que sorprende, ya que se va a ignorar un 50% de las neuronas y vamos a ver entonces se va a forzar a esas neuronas a "trabajar el doble", es decir, va a trabajar más de lo que deberían. La más usada.

Tuneando la Topología

Otra forma de mejorar los resultados de nuestra NN va a ser prueba y error con el número de neuronas y el número de capas que tenemos. Sin embargo vamos a tener formas más metodológicas o una estrategia es reducir el tamaño de nuestro NN en las hidden layers ó se puede evaluar un NN mayor eligiendo un mayor número de hidden layers. Pero como es tan complejo, se hace por prueba y error.

Por lo general no pasa nada, es perfectamente común tener en nuestro NN más neuronas de las que realmente necesitamos. Por lo general, más layers y menos neuronas van a llevarnos a una convergencia más rápida que aquella con muchas neuronas y menos hidden layers.

In [43]:

```
##### Neural Network (Basic Level: Handwritting)

#para instalar
#!pip3 install keras

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

(mnist_train_images, mnist_train_labels), (mnist_test_images, mnist_test_labels) = mnist.load_data()
# 60.000 muestras de train y 10.000 de test y son imágenes de caligrafía y querremos distinguir un dígito (0-9)

# cada una de las muestras es una imagen de 28x28 greyscale píxeles pero lo vamos a querer ver como un array 1D de 784 píxeles
train_images = mnist_train_images.reshape(60000, 784) #60.000 train
test_images = mnist_test_images.reshape(10000, 784) #10.000 test

# Para que sean datos más mejores para las librerías (inicialmente el data es 8-bit, lo pasamos a 32)
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

# Para normalizar (ya que antes teníamos cambio de 8 a 32, si se divide por 255 se está escalando de 0 a 1)
train_images = train_images/255
test_images = test_images/255

# Convertimos las labels a one_hot format 0-10 values
train_labels = keras.utils.to_categorical(mnist_train_labels, 10)
test_labels = keras.utils.to_categorical(mnist_test_labels, 10)

import matplotlib.pyplot as plt

def display_sample(num):
    image = train_images[num].reshape([28,28])
    plt.imshow(image)
    plt.show()

#display_sample(1234)

model = Sequential()

model.add(Dense(512, activation = 'relu', input_shape = (784,) ))
# 512 neuronas, modo relu y el input shape es la dimensión de cada neurona o nodo en el input ; 1ª capa

model.add(Dense(10, activation = 'softmax'))
# 10 como resultado de esas posibles clases a clasificar (0-9 dígitos, son 10)

# Para chequear que todo va bien en el modelo tal y como lo queremos
model.summary()

# Compilar el modelo
model.compile( loss = 'categorical_crossentropy',
              optimizer = RMSprop(), # otras opciones de optimizer son Adagrad, SVG, Ad
```

```
am, Adamax, y Nadam
    metrics = ['accuracy']
)

# Entrenamos el modelo
fit_model = model.fit(train_images, train_labels,
                      batch_size = 100,
                      epochs = 10,
                      verbose = 2,
                      validation_data = (test_images, test_labels)
)
# Podemos ver es lento pero aún así la accuracy es elevada incluso en la epoch 1

score = model.evaluate(test_images, test_labels, verbose = 0)
print('Test Loss:', score[0])
print('Test Accuracy:', score[1])
#Test Loss: 0.06836040318012238
#Test Accuracy: 0.9829999804496765
#Nada mal pero se puede mejorar
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
<hr/>		
dense_2 (Dense)	(None, 512)	401920
<hr/>		
dense_3 (Dense)	(None, 10)	5130
<hr/>		
Total params: 407,050		
Trainable params: 407,050		
Non-trainable params: 0		

Epoch 1/10

600/600 - 5s - loss: 0.2398 - accuracy: 0.9302 - val_loss: 0.1203 - val_accuracy: 0.9640

Epoch 2/10

600/600 - 5s - loss: 0.0976 - accuracy: 0.9710 - val_loss: 0.0817 - val_accuracy: 0.9746

Epoch 3/10

600/600 - 4s - loss: 0.0658 - accuracy: 0.9801 - val_loss: 0.0774 - val_accuracy: 0.9779

Epoch 4/10

600/600 - 5s - loss: 0.0472 - accuracy: 0.9858 - val_loss: 0.0738 - val_accuracy: 0.9791

Epoch 5/10

600/600 - 5s - loss: 0.0353 - accuracy: 0.9898 - val_loss: 0.0691 - val_accuracy: 0.9787

Epoch 6/10

600/600 - 5s - loss: 0.0275 - accuracy: 0.9917 - val_loss: 0.0794 - val_accuracy: 0.9778

Epoch 7/10

600/600 - 5s - loss: 0.0215 - accuracy: 0.9936 - val_loss: 0.0737 - val_accuracy: 0.9793

Epoch 8/10

600/600 - 4s - loss: 0.0166 - accuracy: 0.9952 - val_loss: 0.0684 - val_accuracy: 0.9817

Epoch 9/10

600/600 - 4s - loss: 0.0136 - accuracy: 0.9957 - val_loss: 0.0735 - val_accuracy: 0.9824

Epoch 10/10

600/600 - 5s - loss: 0.0101 - accuracy: 0.9973 - val_loss: 0.0741 - val_accuracy: 0.9822

Test Loss: 0.07412002235651016

Test Accuracy: 0.982200026512146

In [44]:

```
# Vamos a ver los fallos
for x in range(30):
    # rango elegido para que solo aparezca el 1º
    test_image = test_images[x,:].reshape(1,784)
    predicted_cat = model.predict(test_image).argmax()
    label = test_labels[x].argmax()
    if(predicted_cat != label):
        plt.title('Predicted %d but it was a %d' % (predicted_cat, label))
        plt.imshow(test_image.reshape([28, 28]))
        plt.show()
# Podemos ver es un fallo bastante comprensible
```

In [45]:

```
# Vamos a volver atrás y ver si se puede mejorar estos resultados

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

(mnist_train_images, mnist_train_labels), (mnist_test_images, mnist_test_labels) = mnist.load_data()

train_images = mnist_train_images.reshape(60000, 784) #60.000 train
test_images = mnist_test_images.reshape(10000, 784) #10.000 test

train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

train_images = train_images/255
test_images = test_images/255

train_labels = keras.utils.to_categorical(mnist_train_labels, 10)
test_labels = keras.utils.to_categorical(mnist_test_labels, 10)

# Comenzamos, para ello podemos probar:
# diferentes optimizadores que nos sean RMSprop
# diferentes topologías (muchas de ellas, si ya están resueltos los problemas, están por algún lado ya hechas)

model = Sequential()

model.add(Dense(512, activation = 'relu', input_shape = (784,)))

# Se añaden Dropout Layers para prevenir overfitting
model.add(Dropout(0.2))

model.add(Dense(512, activation = 'relu'))

model.add(Dropout(0.2))

model.add(Dense(10, activation = 'softmax'))

model.summary()

model.compile( loss = 'categorical_crossentropy',
               optimizer = RMSprop(), # otras opciones de optimizer son Adagrad, SVG, Adam, Adamax, y Nadam
               metrics = [ 'accuracy' ]
             )

fit_model = model.fit(train_images, train_labels,
                      batch_size = 100,
                      epochs = 10,
                      verbose = 2,
                      validation_data = (test_images, test_labels)
                    )

score = model.evaluate(test_images, test_labels, verbose = 0)
print('Test Loss:', score[0])
```

```

print('Test Accuracy:', score[1])
# Ahora Test Loss: 0.12062180042266846
# Ahora Test Accuracy: 0.9794999957084656

# Antes Test Loss: 0.06836040318012238
# Antes Test Accuracy: 0.9829999804496765

```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
<hr/>		
dense_4 (Dense)	(None, 512)	401920
dropout (Dropout)	(None, 512)	0
dense_5 (Dense)	(None, 512)	262656
dropout_1 (Dropout)	(None, 512)	0
dense_6 (Dense)	(None, 10)	5130
<hr/>		
Total params: 669,706		
Trainable params: 669,706		
Non-trainable params: 0		

Epoch 1/10
 600/600 - 8s - loss: 0.2374 - accuracy: 0.9267 - val_loss: 0.0923 - val_accuracy: 0.9706

Epoch 2/10
 600/600 - 8s - loss: 0.1036 - accuracy: 0.9684 - val_loss: 0.0804 - val_accuracy: 0.9773

Epoch 3/10
 600/600 - 8s - loss: 0.0767 - accuracy: 0.9772 - val_loss: 0.0761 - val_accuracy: 0.9791

Epoch 4/10
 600/600 - 8s - loss: 0.0624 - accuracy: 0.9822 - val_loss: 0.0757 - val_accuracy: 0.9815

Epoch 5/10
 600/600 - 7s - loss: 0.0533 - accuracy: 0.9846 - val_loss: 0.0750 - val_accuracy: 0.9815

Epoch 6/10
 600/600 - 7s - loss: 0.0448 - accuracy: 0.9865 - val_loss: 0.0842 - val_accuracy: 0.9806

Epoch 7/10
 600/600 - 8s - loss: 0.0409 - accuracy: 0.9879 - val_loss: 0.0906 - val_accuracy: 0.9819

Epoch 8/10
 600/600 - 8s - loss: 0.0391 - accuracy: 0.9890 - val_loss: 0.0928 - val_accuracy: 0.9807

Epoch 9/10
 600/600 - 7s - loss: 0.0336 - accuracy: 0.9907 - val_loss: 0.0969 - val_accuracy: 0.9808

Epoch 10/10
 600/600 - 7s - loss: 0.0316 - accuracy: 0.9916 - val_loss: 0.1016 - val_accuracy: 0.9835

Test Loss: 0.10157257318496704

Test Accuracy: 0.9835000038146973

Podemos ver esto era un problema de clasificación multiclas ya que para el resultado a clasificar tenemos varias opciones: números de 0 a 9.

```
model = Sequential()
model.add(Dense(64, activation = 'relu', input_dim = 20 ))
```

64 neuronas

```
model.add(Dropout(0.5))
```

prevenir overfitting

```
model.add(Dense(64, activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation = 'softmax'))
sgd = SGD(lr = 0.01, decay = 1e-6, momentum = 0.9, nesterov = True)
model.compile(loss = 'categorical_crossentropy',
optimizer = sgd, # otras opciones de optimizer son Adagrad, SVG, Adam, Adamax, y Nadam metrics =
['accuracy'])
```

Binary Classification

Podemos ver ahora el caso de querer hacer una clasificación binaria, es decir, distinguir nuestro output en una clasificación en dos clases diferentes (male or female, democrat or republican, cat or dog...)

```
model = Sequential()
model.add(Dense(64, activation = 'relu', input_dim = 20 ))
model.add(Dropout(0.5))
model.add(Dense(64, activation = 'relu'))
model.add(Dropout(0.5))
```

Lo recomendado para el caso binario es usar una 'sigmoid' en vez de 'softmax', ya que no se requiere de la complejidad de softmax si se quiere evaluar si 0 ó 1

```
model.add(Dense(10, activation = 'sigmoid'))
sgd = SGD(lr = 0.01, decay = 1e-6, momentum = 0.9, nesterov = True)
```

Para la clasificación binaria podemos ver vamos a cambiar tanto la función loss, como el optimizador,

son los recomendados para resolver este problema concreto

```
model.compile( loss = 'binary_crossentropy',# antes era cross_entropy optimizer = rmsprop, # otras opciones de optimizer son Adagrad, SVG, Adam, Adamax, y Nadam metrics = ['accuracy'] )
```

Integrando keras con scikit_learn

```
from keras.wrappers.scikit_learn import KerasClassifier  
  
from sklearn.model_selection import cross_val_score  
  
def create_model():  
  
    model = Sequential()  
  
    #input layer  
    model.add(Dense(6, activation = 'relu', input_dim = 4, kernel_initializer = 'normal'))  
  
    # hidden layer  
    model.add(Dense(4, activation = 'relu', kernel_initializer = 'normal'))  
  
    #output layer  
    model.add(Dense(1, activation = 'sigmoid', kernel_initializer = 'normal'))  
  
    model.compile(loss = 'binary_entropy', optimizer = 'rmsprop', metrics = ['accuracy'])  
  
    return model  
  
  
binary_estimator = KerasClassifier(build_fn = create_model(), nb_epoch = 100, verbose = 0)  
  
cv_scores = cross_val_score(binary_estimator, features, labels, cv = 10)  
  
print(cv_scores.mean())
```

In [46]:

```
##### ##### Neural Network (Basic level: Predict political parties with keras)

# Predecir el partido de los congresistas basado en sus votos anteriores

import pandas as pd
from sklearn.model_selection import cross_val_score

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop
from keras.wrappers.scikit_learn import KerasClassifier

features = ['party','handicapped-infants', 'water-project-cost-sharing', 'adoption-of-the-budget-resolution',
            'physician-fee-freeze', 'el-salvador-aid', 'religious-groups-in-schools',
'anti-satellite-test-ban',
            'aid-to-nicaraguan-contras', 'mx-missle', 'immigration', 'synfuels-corporation-cutback',
            'education-spending', 'superfund-right-to-sue', 'crime', 'duty-free-exports'
,
            'export-administration-act-south-africa']

voting_data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\DeepLearning\house-votes-84.data.txt',
na_values = ['?'], names = features)
voting_data.head()

# Hay que limpiar la base de datos y tenemos que poner los datos en formato apto para k eras

# Podemos ver en la primera de las columnas es el target o lo que vamos a querer predecir

voting_data.describe()
# Y vemos hay muchísimos missing values que debemos eliminar y por ello entonces vamos a tener que eliminar esas filas,
# ya que no queremos que nuestras medidas sean afectadas por el bias
voting_data.dropna(inplace = True)
voting_data.describe()
# Nos quedamos con 232 políticos, no es lo ideal pero bueno

# El siguiente paso es que sabemos que a keras no le gustan las variables categóricas, le vamos a pasar variables numéricas

voting_data.replace(('y','n') , (1,0), inplace = True)
voting_data.replace(('democrat','republican'), (1,0), inplace = True)
voting_data

# Ahora que todo es numérico tenemos que convertirlo a numpy_array para que así se lo podamos dar a keras

# El mismo que 'features' pero sin party
features_classes = ['handicapped-infants', 'water-project-cost-sharing', 'adoption-of-the-budget-resolution',
            'physician-fee-freeze', 'el-salvador-aid', 'religious-groups-in-schools',
'anti-satellite-test-ban',
            'aid-to-nicaraguan-contras', 'mx-missle', 'immigration', 'synfuels-corporat
```

```
ion-cutback',
    'education-spending', 'superfund-right-to-sue', 'crime', 'duty-free-exports'
,
    'export-administration-act-south-africa']
#print(len(features_classes)): 16
X = voting_data[features].values
y = voting_data['party'].values

def create_model():
    model = Sequential()
    #Lo normal es empezar con un gran número de neuronas e ir reduciendo
    model.add(Dense(32, activation = 'relu', input_dim = 17, kernel_initializer = 'normal'))
    #hidden Layer con 16 neuronas
    model.add(Dense(16, activation = 'relu', kernel_initializer = 'normal'))
    model.add(Dense(1, activation = 'sigmoid', kernel_initializer = 'normal'))
    model.compile(loss = 'binary_crossentropy', optimizer = 'rmsprop', metrics = ['accuracy'])
    return model

#binary_estimator = KerasClassifier(build_fn = create_model, nb_epoch = 100, verbose = 0)

#cv_scores = cross_val_score(binary_estimator, X, y, cv = 10)

#print('cv_scores.mean():',cv_scores.mean())

#model = create_model()

model = Sequential()
#Lo normal es empezar con un gran número de neuronas e ir reduciendo
model.add(Dense(32, activation = 'relu', input_dim = 17, kernel_initializer = 'normal'))
#hidden Layer con 16 neuronas
model.add(Dense(16, activation = 'relu', kernel_initializer = 'normal'))
model.add(Dense(1, activation = 'sigmoid', kernel_initializer = 'normal'))
model.compile(loss = 'binary_crossentropy', optimizer = 'rmsprop', metrics = ['accuracy'])

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)

fit_model = model.fit(X_train, y_train,
                      batch_size = 100,
                      epochs = 10,
                      verbose = 2,
                      validation_data = (X_test, y_test)
                     )

score = model.evaluate(X_test, y_test, verbose = 0)
print('Test Loss:', score[0])
print('Test Accuracy:', score[1])
```

```
Epoch 1/10
3/3 - 1s - loss: 0.6928 - accuracy: 0.4904 - val_loss: 0.6900 - val_accuracy: 0.5000
Epoch 2/10
3/3 - 0s - loss: 0.6885 - accuracy: 0.5385 - val_loss: 0.6871 - val_accuracy: 0.7917
Epoch 3/10
3/3 - 0s - loss: 0.6849 - accuracy: 0.8750 - val_loss: 0.6825 - val_accuracy: 0.8333
Epoch 4/10
3/3 - 0s - loss: 0.6792 - accuracy: 0.8942 - val_loss: 0.6766 - val_accuracy: 0.8333
Epoch 5/10
3/3 - 0s - loss: 0.6720 - accuracy: 0.9231 - val_loss: 0.6694 - val_accuracy: 0.8333
Epoch 6/10
3/3 - 0s - loss: 0.6634 - accuracy: 0.9327 - val_loss: 0.6619 - val_accuracy: 0.8333
Epoch 7/10
3/3 - 0s - loss: 0.6547 - accuracy: 0.9471 - val_loss: 0.6528 - val_accuracy: 0.8333
Epoch 8/10
3/3 - 0s - loss: 0.6434 - accuracy: 0.9423 - val_loss: 0.6439 - val_accuracy: 0.7917
Epoch 9/10
3/3 - 0s - loss: 0.6319 - accuracy: 0.8942 - val_loss: 0.6327 - val_accuracy: 0.8333
Epoch 10/10
3/3 - 0s - loss: 0.6189 - accuracy: 0.9375 - val_loss: 0.6218 - val_accuracy: 0.8333
Test Loss: 0.6217913031578064
Test Accuracy: 0.8333333134651184
```

Convolutional Neural Networks (CNN)

Una cosa ya vista son los perceptrones y ahora podemos pasar a ver problemas para NN's más complicados como ocurre con los CNN. Se suelen utilizar para el análisis de imágenes, ya que se trata de buscar anomalías en los datos o cosas que uno no se esperaría encontrar. Es decir, nos vale para encontrar patrones y características en los datos aun cuando no sabemos por dónde buscarlos, de ello se encarga CNN ya que los encuentra, sin importar dónde estén, por nosotros.

¿Cómo funcionan las CNN's? Funcionan con convolución, es decir, dividir las imágenes en chunks (llamados convolutions), o cachitos pequeños y luego unirlos o hacer un ensamblado y así formar la imagen grande, y así podremos ver realmente tenemos varias complejidades o capas de características, esto es Deep Learning, y en cada una de esas layers vamos a ver van a ser para identificar características como líneas horizontales, líneas verticales, líneas a diferentes ángulos... los llamaremos filters. Y poniendo filtros por encima de ellos podremos ver si somos capaces de distinguir objetos o formas basados en los patrones o formas. Y si encima está en color habrá que multiplicarlo por 3 (Red, Green y Blue lights, los colores principales).

Este reconocimiento de patrones se basa en el mismo concepto que nuestro cerebro, por diferencias de contrastes, en edges o extremos o bordes. También sabemos que por diferencias de formas o colores es que podremos ver este contraste hasta llegar a alguna layer que sea capaz de, con ayuda de las anteriores, de reconocer de qué se trata lo que se está visualizando.

Tendremos que tener en cuenta que si se va a trabajar con imágenes vamos a multiplicar los píxeles de la horizontal por los píxeles de la vertical y multiplicarlo también por el número de canales de color (en caso de blanco y negro, es 1, y si es en color habrá que multiplicar ese rectángulo de píxeles por 3 debido a los colores primarios). Tendremos un tipo de capa llamada "Conv2D" que nos será útil para separar esa imagen en chunks y poder procesarla por partes. También podemos usar CNN para más que imágenes, como con "Conv1D" para análisis de textos o datos 1D o podemos usar "Conv3D" para volúmenes.

Otra layer especializada en keras es "MaxPooling2D" (También tenemos sus variantes 1D y 3D como antes), y la idea es reducir el tamaño del data de tal forma que sería como "encoger las imágenes". Para poder optimizar esto, además de esa reducción de tamaño, habremos de usar "Dropout".

Finalmente tendremos que tener una layer que haga "flatten" para convertir esta imagen 2D en un array 1D y así tener un perceptrón multicapa.

Así, podemos ver realmente que una CNN seguirá casi siempre el siguiente esquema

Conv2D -> MaxPooling2D -> Dropout -> Flatten -> Dense -> Dropout -> Softmax

CNN son muy pesadas, ya que llevan un gran uso de la CPU, GPU, y memoria. Todo esto realmente va a estar regulado por hiperparámetros, además de los típicos parámetros que controlamos como son la topología, optimizador, función de pérdida o activación, o tipo o tamaño de kernel, el área que se convoluciona, cuántas capas y unidades tenemos, el pooling...

La dificultad de estos hiperparámetros está resuelta por una serie de arquitecturas o algoritmos de CNN de tal forma que veremos, por lo cual la dificultad del problema vendrá realmente dada por la elección de esta arquitectura óptima para nuestro problema. Tenemos las siguientes arquitecturas:

- LeNet-5: muy adecuada para reconocimiento del handwritting.
- AlexNet: para clasificación de imágenes, más profunda que LeNet
- GoogleLeNet: más profunda aún que AlexNet (más layers) y tiene un mejor resultado ya que introduce un concepto denominado 'inception modules', que son grupos de layers convolucionales lo cual es muy útil y funciona en una gran mayoría de casos.
- ResNet: (Residual Network), el más sofisticado hoy en día, aún más deep y se b

In [47]:

```
##### CNN (Convolutional Neural Network) (Basic Level: Predict political parties with k
eras)

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from keras.optimizers import RMSprop

(mnist_train_images, mnist_train_labels), (mnist_test_images, mnist_test_labels) = mnis
t.load_data()
# 60.000 muestras de train y 10.000 de test y son imágenes de caligrafía y querremos di
stinguir un dígito (0-9)

# Como ahora tenemos una CNN en vez de una NN vamos a poder trabajar con una imagen 2D,
no como antes que había que pasar a 1D
# En vez de eso, se convierte en un array de chunks que va a ser el número de píxeles d
el ancho (width, W) por el # de píxeles
# del alt (L, length) o de la imagen y además multiplicados por la escala de color (c),
en este caso es grey así que tendremos
# L*W*c = L*W

from keras import backend as K

if K.image_data_format() == 'channels_first':
    train_images = mnist_train_images.reshape(mnist_train_images.shape[0], 1, 28, 28)
    test_images = mnist_test_images.reshape(mnist_test_images.shape[0], 1, 28, 28)
    input_shape = (1, 28, 28)
else:
    train_images = mnist_train_images.reshape(mnist_train_images.shape[0], 28, 28, 1)
    test_images = mnist_test_images.reshape(mnist_test_images.shape[0], 28, 28, 1)
    input_shape = (28, 28, 1)

train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

train_images = train_images/255
test_images = test_images/255

train_labels = keras.utils.to_categorical(mnist_train_labels, 10)
test_labels = keras.utils.to_categorical(mnist_test_labels, 10)

def display_sample(num):
    image = train_images[num].reshape([28,28])
    plt.imshow(image)
    plt.show()

display_sample(1234)

# Conv2D -> MaxPooling2D -> Dropout -> Flatten -> Dense -> Dropout -> Softmax

model = Sequential()

# el input_shape antes era esas (784,), ahora es (28,28,1) ó (1,28,28)
model.add(Conv2D(32, kernel_size = (3,3), activation = 'relu', input_shape = input_shap
e))

# 64 neuronas (3x3) kernels
model.add(Conv2D(64, (3,3), activation = 'relu'))
```

```
# Vamos a reducir tomando el mayor del kernel (2x2)
model.add(MaxPooling2D(pool_size = (2,2)))

# Evitar overfitting
model.add(Dropout(0.25))

# Hacer flatten para convertir a 1D y pasárselo a la layer final
model.add(Flatten())

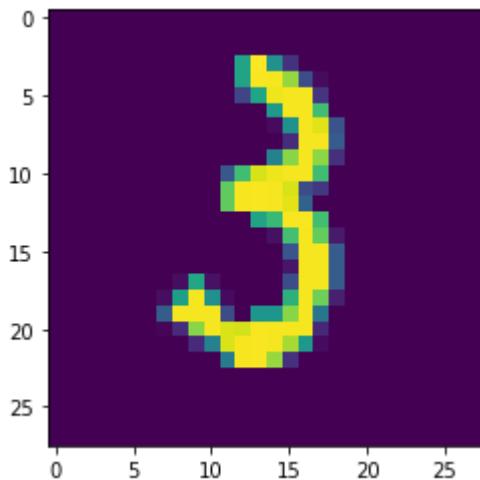
# Una hidden Layer para aprender
model.add(Dense(128, activation = 'relu'))

# Evitar overfitting again
model.add(Dropout(0.5))

# Categorización final de 0 a 9 con Softmax
model.add(Dense(10, activation = 'softmax'))

model.summary()

model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
```



Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
conv2d_1 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d (MaxPooling2D)	(None, 12, 12, 64)	0
dropout_2 (Dropout)	(None, 12, 12, 64)	0
flatten (Flatten)	(None, 9216)	0
dense_10 (Dense)	(None, 128)	1179776
dropout_3 (Dropout)	(None, 128)	0
dense_11 (Dense)	(None, 10)	1290
<hr/>		
Total params: 1,199,882		
Trainable params: 1,199,882		
Non-trainable params: 0		

In [48]:

```
#### ojo no ejecutar que tarda una 20 mins cuidado!!!!  
  
def run_model():  
    history = model.fit(train_images, train_labels,  
                         batch_size = 32,  
                         epochs = 10,  
                         verbose = 2,  
                         validation_data = (test_images, test_labels)  
    )  
    score = model.evaluate(test_images, test_labels, verbose = 0)  
    print('Test Loss:', score[0])  
    print('Test Accuracy:', score[1])  
  
#run_model()  
  
# De resultado  
#Epoch 1/10  
#1875/1875 - 100s - Loss: 0.2042 - accuracy: 0.9381 - val_loss: 0.0496 - val_accuracy:  
0.9836  
#Epoch 2/10  
#1875/1875 - 101s - Loss: 0.0854 - accuracy: 0.9745 - val_loss: 0.0372 - val_accuracy:  
0.9872  
#Epoch 3/10  
#1875/1875 - 104s - Loss: 0.0659 - accuracy: 0.9803 - val_loss: 0.0320 - val_accuracy:  
0.9903  
#Epoch 4/10  
#1875/1875 - 108s - Loss: 0.0540 - accuracy: 0.9837 - val_loss: 0.0303 - val_accuracy:  
0.9904  
#Epoch 5/10  
#1875/1875 - 108s - Loss: 0.0451 - accuracy: 0.9859 - val_loss: 0.0305 - val_accuracy:  
0.9905  
#Epoch 6/10  
#1875/1875 - 112s - Loss: 0.0387 - accuracy: 0.9879 - val_loss: 0.0302 - val_accuracy:  
0.9911  
#Epoch 7/10  
#1875/1875 - 116s - Loss: 0.0343 - accuracy: 0.9885 - val_loss: 0.0273 - val_accuracy:  
0.9923  
#Epoch 8/10  
#1875/1875 - 117s - Loss: 0.0301 - accuracy: 0.9905 - val_loss: 0.0321 - val_accuracy:  
0.9905  
#Epoch 9/10  
#1875/1875 - 119s - Loss: 0.0280 - accuracy: 0.9905 - val_loss: 0.0295 - val_accuracy:  
0.9919  
#Epoch 10/10  
#1875/1875 - 116s - Loss: 0.0257 - accuracy: 0.9917 - val_loss: 0.0312 - val_accuracy:  
0.9907  
#Test Loss: 0.031180117279291153  
#Test Accuracy: 0.9907000064849854
```

Recurrent Neural Networks (RNN)

Son para secuencias de datos, bien podrían ser una secuencia de datos en tiempo. Podremos ver además estas NN van a venir dadas de tal forma que su output de un nodo concreto regresa a ese mismo nodo o neurona como el nuevo input, ahí la recurrencia. Inicialmente puede que tuviésemos una función escalón como función de activación, pero a medida que van pasando las iteraciones, esa suma de funciones escalón va a converger en una forma más suavizada similar a la expresión de una tangente hiperbólica.

De esta forma recursiva es por lo que podemos ver la información del pasado modifica la actual, es decir, el pasado va a contribuir al futuro (en términos del input, es ver como este tiene memoria). Si se prefiere ver de forma "secuencial", cada una de estas veces que de nuestra layer sale el output hacia sí misma como nuevo input, unas N veces, sería lo mismo visualizar N layers iguales en serie, cada una de ellas recibiendo un único input y enviándolo a la siguiente. Como es evidente, el pasado más reciente influirá de mayor forma en estas nuevas neuronas o nodos.

Podemos tener una capa de RNN's que funciona tal que, les aplicamos un mismo input a todas las neuronas, y sumamos los outputs de las neuronas que haya y ese va a ser el nuevo input para todas ellas, e iterando el proceso mencionado tenemos esta capa de RNN's.

Pueden ser tan complicadas o más incluso que las CNN's, ya que ahora vamos a necesitar 'backpropagation' no solo en la RNN propia sino también en el tiempo y un standpoint práctico va a venir bien ya que a medida que iteramos, como hemos visto antes, cada vez que se itera es el equivalente a otra layer, y si acabamos teniendo muchas el coste posterior para poder realizar el fit va a ser innecesariamente grande, de tal forma que vamos a ver tenemos que añadir un tiempo máximo de train a nuestra RNN para que no sea demasiado Deep, a veces se limita esta backpropagation a un número limitado de pasos o iteraciones que vamos a llamar 'truncated backpropagation'.

Como hemos mencionado, la información más reciente va a tener mayor peso que la antigua, y de esta forma veremos esto puede ser un problema avanzada la serie temporal si la información al principio era realmente importante. Una solución a este problema es un tipo de estructura o célula llamada LSTM Cell (Long Short-Term Memory Cell), que va a guardar por separado la información reciente por un lado, y por otro la información asociada a términos a gran escala o alcance temporal (más antiguo). Otra opción es una GRU Cell, que nos va a servir como una LSTM pero simplificada y obtiene unos resultados bastante parecidos.

Las RNN también son muy difíciles de entrenar ya que van a ser muy sensibles a las topologías, y vamos a ver la elección de hiperparámetros va a ser realmente difícil. Por tratarse de algo tan sensible se pueden cometer fallos y por tratarse de series temporales podremos ver a medida que se avanza en la red ese error se va arrastrando de tal forma que el problema no converja o no sea al final un buen clasificador.

In [49]:

```
##### RNN (Recurrent Neural Network) (Basic Level: Sentiment Analysis from movie reviews)

import keras
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding
from keras.layers import LSTM
from keras.datasets import imdb

# Vamos a analizar cadenas R, reviews, y vamos a intentar predecir si es positiva o negativa
# Se va a usar esa LSTM para hacer counter a ese efecto de que perdamos información de los primeros valores

(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words = 20000)
# Vamos a limitarnos a las 20.000 palabras más importantes
# y = 1 si les gustó /// y = 0 si no les gustó

X_train[0]
# Y vemos tenemos estas son las palabras más populares y ya están encoded, por lo que son números del 0 al 20.000 y ya está preprocessado y nos ahorramos convertir de categórico a numérico.

y_train[0]

# Para poder hacer backpropagation tenemos que fijar un límite, ya que sino puede tener un coste computacional de días incluso
X_train = sequence.pad_sequences(X_train, maxlen = 80)
X_test = sequence.pad_sequences(X_test, maxlen = 80)

#Generamos el modelo RNN

model = Sequential()

# Un poco de preprocessamiento para conseguir estos vectores en esta Embedding Layer:
# Se va a convertir esos datos en esos 80 que hemos truncado y se convierten en dense vectors de tamaño fijo
# Así que se va a crear un dense vector de 20.000 palabras y eso a embudar o comprimir en un 128 neuronas
# Realmente se va a convertir dichos datos en apropiados para nuestro modelo
model.add(Embedding(20000, 128))

# Toda la RNN aparece en la siguiente Línea de código, con las 128 neuronas
# el dropout se puede pasar directamente como argumento como se ve del 20%
model.add(LSTM(128, dropout = 0.2, recurrent_dropout = 0.2))

# sigmoid por ser binario
model.add(Dense(1, activation = 'sigmoid'))

# Esquema del modelo
model.summary()

# Compilamos el modelo
model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
<hr/>		
embedding (Embedding)	(None, None, 128)	2560000
lstm (LSTM)	(None, 128)	131584
dense_12 (Dense)	(None, 1)	129
<hr/>		
Total params:	2,691,713	
Trainable params:	2,691,713	
Non-trainable params:	0	

In [50]:

```
#### ojo no ejecutar que tarda una 40 mins cuidado!!!!  
  
def run_model():  
    history = model.fit(X_train, y_train,  
                         batch_size = 32,  
                         epochs = 15,  
                         verbose = 2,  
                         validation_data = (X_test, y_test)  
    )  
    score = model.evaluate(X_test, y_test, verbose = 0)  
    print('Test Loss:', score[0])  
    print('Test Accuracy:', score[1])  
  
#run_model()  
  
# No se van a obtener grandes resultados pero es porque solo estamos mirando las 80 primeras palabras  
  
#Epoch 1/15  
#782/782 - 145s - loss: 0.4319 - accuracy: 0.7936 - val_loss: 0.3787 - val_accuracy: 0.8356  
#Epoch 2/15  
#782/782 - 143s - loss: 0.2548 - accuracy: 0.9007 - val_loss: 0.3667 - val_accuracy: 0.8398  
#Epoch 3/15  
#782/782 - 156s - loss: 0.1639 - accuracy: 0.9391 - val_loss: 0.4844 - val_accuracy: 0.8261  
#Epoch 4/15  
#782/782 - 162s - loss: 0.1062 - accuracy: 0.9605 - val_loss: 0.5784 - val_accuracy: 0.8251  
#Epoch 5/15  
#782/782 - 154s - loss: 0.0673 - accuracy: 0.9764 - val_loss: 0.6045 - val_accuracy: 0.8209  
#Epoch 6/15  
#782/782 - 150s - loss: 0.0478 - accuracy: 0.9843 - val_loss: 0.6704 - val_accuracy: 0.8193  
#Epoch 7/15  
#782/782 - 141s - loss: 0.0408 - accuracy: 0.9869 - val_loss: 0.9328 - val_accuracy: 0.8186  
#Epoch 8/15  
#782/782 - 141s - loss: 0.0292 - accuracy: 0.9904 - val_loss: 0.7958 - val_accuracy: 0.8150  
#Epoch 9/15  
#782/782 - 149s - loss: 0.0199 - accuracy: 0.9940 - val_loss: 0.9216 - val_accuracy: 0.8170  
#Epoch 10/15  
#782/782 - 150s - loss: 0.0138 - accuracy: 0.9960 - val_loss: 0.9949 - val_accuracy: 0.8118  
#Epoch 11/15  
#782/782 - 145s - loss: 0.0171 - accuracy: 0.9942 - val_loss: 0.9469 - val_accuracy: 0.8057  
#Epoch 12/15  
#782/782 - 155s - loss: 0.0175 - accuracy: 0.9945 - val_loss: 0.8837 - val_accuracy: 0.8025  
#Epoch 13/15  
#782/782 - 154s - loss: 0.0129 - accuracy: 0.9961 - val_loss: 0.9860 - val_accuracy: 0.8183  
#Epoch 14/15  
#782/782 - 160s - loss: 0.0085 - accuracy: 0.9975 - val_loss: 1.0555 - val_accuracy: 0.
```

```
8137
```

```
#Epoch 15/15
```

```
#782/782 - 151s - loss: 0.0087 - accuracy: 0.9973 - val_loss: 1.0907 - val_accuracy: 0.  
8134
```

```
#Test Loss: 1.090665340423584
```

```
#Test Accuracy: 0.8133999705314636
```

In [51]:

```
### PROYECTO FINAL DE NN's (Predicción de Cancer de mama según mamografías)

import pandas as pd
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

import keras
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Embedding, Dropout, Conv2D, MaxPooling2D, Flatten
from keras.layers import LSTM
from keras.datasets import imdb
from keras.optimizers import RMSprop
from keras.wrappers.scikit_learn import KerasClassifier

# Dataset de 961 instancias (filas) de masas detectadas en mamografías, 6 columnas

# 1. BI-RAIDS assesment: 1-5 (ordinal)
# 2. Age: patient's age in years (integer)
# 3. Shape: mass shape; round=1, oval=2, lobular=3, irregular=4 (nominal)
# 4. Margin: mass margin: circumscribed=1 microlobulated=2 obscured=3 ill-defined=4 spiculated=5 (nominal)
# 5. Density: mass density high=1 iso=2 low=3 fat-containing=4 (ordinal)
# 6. Severity: benign=0 or malignant=1 (binomial)

# Como bien sabemos los valores nominales no son los mejores y siempre va a ser mejor tratar con data ordinal o numérico
# no obstante, al menos están ordenados de tal forma que están 'medianamente ordenados' de menos a más irregular

# Etiquetas a los datos
features = ['BI-RAIDS assesment', 'Age', 'Shape', 'Margin', 'Density', 'Severity']

# Importamos estos datos
data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\DeepLearning\mammographic_masses.data.txt', na_values = ['?'], names = features)
data.head()

# Vamos a observar nuestros datos
data.describe()
# Podemos ver en count que hay missing values

# Y vemos hay muchísimos missing values que debemos eliminar y por ello entonces vamos a tener que eliminar esas filas,
# ya que no queremos que nuestras medidas sean afectadas por el bias
data.dropna(inplace = True)
data.describe()
# Nos quedamos con 830 de las 961 instancias ideales, no es lo ideal pero bueno

# Ahora que todo es numérico tenemos que convertirlo a numpy_array para que así se lo podamos dar a keras
# El mismo que 'features' pero sin Severity que es el target
features_classes = ['BI-RAIDS assesment', 'Age', 'Shape', 'Margin', 'Density']
X = data[features_classes].values
y = data['Severity'].values

# Siempre normalizar
```

```
scaler = preprocessing.StandardScaler()
X = scaler.fit_transform(X)

def create_model_nn_binary():

    model = Sequential()

    # Lo normal es empezar con un gran número de neuronas e ir reduciendo
    model.add(Dense(6, activation = 'relu', input_dim = 5, kernel_initializer = 'normal'))

    # hidden Layer con 16 neuronas
    model.add(Dense(16, activation = 'relu', kernel_initializer = 'normal'))
    # sin esto, solo se llega a un 72%

    model.add(Dense(1, activation = 'sigmoid', kernel_initializer = 'normal'))

    model.compile(loss = 'binary_crossentropy', optimizer = 'rmsprop', metrics = [ 'accuracy' ])
    return model

#def create_model_cnn():
#def create_model_rnn():

model = create_model_nn_binary()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)

fit_model = model.fit(X_train, y_train,
                      batch_size = 100,
                      epochs = 10,
                      verbose = 2,
                      validation_data = (X_test, y_test)
                     )

score = model.evaluate(X_test, y_test, verbose = 0)
print('Test Loss:', score[0])
print('Test Accuracy:', score[1])
```

```
Epoch 1/10
8/8 - 0s - loss: 0.6930 - accuracy: 0.5515 - val_loss: 0.6927 - val_accuracy: 0.4699
Epoch 2/10
8/8 - 0s - loss: 0.6922 - accuracy: 0.5248 - val_loss: 0.6917 - val_accuracy: 0.5422
Epoch 3/10
8/8 - 0s - loss: 0.6910 - accuracy: 0.6854 - val_loss: 0.6899 - val_accuracy: 0.7590
Epoch 4/10
8/8 - 0s - loss: 0.6890 - accuracy: 0.8032 - val_loss: 0.6873 - val_accuracy: 0.8193
Epoch 5/10
8/8 - 0s - loss: 0.6863 - accuracy: 0.8153 - val_loss: 0.6838 - val_accuracy: 0.8193
Epoch 6/10
8/8 - 0s - loss: 0.6829 - accuracy: 0.8139 - val_loss: 0.6796 - val_accuracy: 0.8675
Epoch 7/10
8/8 - 0s - loss: 0.6790 - accuracy: 0.8072 - val_loss: 0.6749 - val_accuracy: 0.8675
Epoch 8/10
8/8 - 0s - loss: 0.6745 - accuracy: 0.8126 - val_loss: 0.6693 - val_accuracy: 0.8675
Epoch 9/10
8/8 - 0s - loss: 0.6691 - accuracy: 0.8086 - val_loss: 0.6623 - val_accuracy: 0.8675
Epoch 10/10
8/8 - 0s - loss: 0.6626 - accuracy: 0.8032 - val_loss: 0.6543 - val_accuracy: 0.8675
Test Loss: 0.6542593836784363
Test Accuracy: 0.8674699068069458
```

In [109]:

```
##### EJERCICIOS ANÁLISIS DE IMÁGENES

##### CNN (Reconocimiento de caras; distinción hombre-mujer)

from keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
from keras.models import Sequential, load_model, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, GlobalAveragePooling2D
from keras import backend as K
from keras.callbacks import ModelCheckpoint
from keras import regularizers
from keras import optimizers
from keras import models
from glob import glob
#!pip install opencv-python
import cv2
import os
from skimage.io import imread
from skimage import transform
from cv2 import resize

import matplotlib.pyplot as plt
import numpy as np

from IPython.display import SVG, display, clear_output
from keras.utils.vis_utils import model_to_dot

from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc

faces_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\caras'
faces_dir_output = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\caras\output'

batch_size = 16

#!pip install split-folders tqdm
# import splitfolders
# splitfolders.ratio(faces_dir, output = faces_dir_output, seed=1337, ratio=(.8, 0.1, 0.1));

# Antes teníamos una carpeta llamada 'caras', en ella 'famosos' y 'famosas'
# con esto hemos creado una carpeta extra y dentro de 'caras' ahora tenemos 'famosos',
# 'famosas' y 'output'
# dentro de output tenemos ahora las carpetas de 'train', 'test' y 'val', dentro de las cuales 'famosos' y 'famosas'

train_data_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\caras\output\train'
validation_data_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\caras\output\val'
test_data_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\caras\output\test'

batch_size = 16

# data augmentation:
```

```
train_datagen = ImageDataGenerator(  
    #rescale=1. / 255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    fill_mode='nearest',  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True)  
  
preprocess_input = lambda x:x/255.  
  
# dimensiones a las que vamos a llevar las imágenes  
img_width, img_height = 150, 150  
  
normed_dims = (img_height, img_width)  
  
train_datagen = ImageDataGenerator(  
    dtype='float32',  
    #rescale=1. / 255,  
    preprocessing_function = preprocess_input,  
    rotation_range=10,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    fill_mode='nearest',  
    shear_range=0.1,  
    zoom_range=0.1,  
    horizontal_flip=True)  
  
val_datagen = ImageDataGenerator(dtype='float32',  
                                 #rescale=1. / 255,  
                                 preprocessing_function = preprocess_input)  
  
test_datagen = ImageDataGenerator(dtype='float32',  
                                 #rescale=1. / 255,  
                                 preprocessing_function = preprocess_input)  
  
train_generator = train_datagen.flow_from_directory(  
    train_data_dir,  
    target_size=normed_dims,  
    batch_size=batch_size,  
    shuffle=True,  
    class_mode='binary')  
  
validation_generator = val_datagen.flow_from_directory(  
    validation_data_dir,  
    target_size=normed_dims,  
    batch_size=batch_size,  
    shuffle=False,  
    class_mode='binary')  
  
test_generator = test_datagen.flow_from_directory(  
    test_data_dir,  
    target_size=normed_dims,  
    batch_size=batch_size,  
    shuffle=False,  
    class_mode='binary')
```

```
# Model
model = Sequential()

model.add(Conv2D(32, (3,3), activation = 'relu', input_shape = normed_dims+(3,)))
model.add(MaxPooling2D((2,2)))

model.add(Conv2D(32, (3,3), activation = 'relu'))

model.add(MaxPooling2D((2,2)))

model.add(Conv2D(64, (3,3), activation = 'relu'))

model.add(MaxPooling2D((2,2)))

model.add(Flatten())

# aqui empieza la red neuronal 'standard'
model.add(Dropout(0.5))

model.add(Dense(128, activation = 'relu'))

model.add(Dropout(0.5))

model.add(Dense(1, activation = 'sigmoid'))

model.compile(loss = 'binary_crossentropy', optimizer=optimizers.RMSprop(),metrics=['acc'])
    #optimizer=optimizers.RMSprop(lr=1e-4),
    #optimizer=optimizers.RMSprop(),
    #metrics=['acc'])

model.summary()
```

Found 1722 images belonging to 2 classes.

Found 215 images belonging to 2 classes.

Found 216 images belonging to 2 classes.

Model: "sequential_15"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_26 (Conv2D)	(None, 148, 148, 32)	896
<hr/>		
max_pooling2d_25 (MaxPooling)	(None, 74, 74, 32)	0
<hr/>		
conv2d_27 (Conv2D)	(None, 72, 72, 32)	9248
<hr/>		
max_pooling2d_26 (MaxPooling)	(None, 36, 36, 32)	0
<hr/>		
conv2d_28 (Conv2D)	(None, 34, 34, 64)	18496
<hr/>		
max_pooling2d_27 (MaxPooling)	(None, 17, 17, 64)	0
<hr/>		
flatten_9 (Flatten)	(None, 18496)	0
<hr/>		
dropout_20 (Dropout)	(None, 18496)	0
<hr/>		
dense_32 (Dense)	(None, 128)	2367616
<hr/>		
dropout_21 (Dropout)	(None, 128)	0
<hr/>		
dense_33 (Dense)	(None, 1)	129
<hr/>		

Total params: 2,396,385

Trainable params: 2,396,385

Non-trainable params: 0

In [112]:

```
#### ojo no ejecutar que tarda una 10 mins cuidado!!!!
```

```
modelpath="model_current_best_parte3.h5"

checkpoint = ModelCheckpoint(modelpath, monitor='val_acc', verbose=1,
                             save_best_only=True,
                             mode='max') # graba sólo los que mejoran en validación
callbacks_list = [checkpoint]

def run_model():
    history = model.fit_generator(generator = train_generator,
                                   steps_per_epoch=train_generator.n // batch_size,
                                   epochs=15,
                                   callbacks=callbacks_list,
                                   verbose=1,
                                   shuffle = False,
                                   validation_data=validation_generator,
                                   validation_steps=validation_generator.n // batch_size
    )

#run_model()

#Epoch 1/15
#107/107 [=====] - ETA: 0s - loss: 0.6515 - acc: 0.6157
#Epoch 00001: val_acc improved from -inf to 0.75481, saving model to model_current_best_parte3.h5
#107/107 [=====] - 50s 471ms/step - loss: 0.6515 - acc: 0.6157
- val_loss: 0.5286 - val_acc: 0.7548
#Epoch 2/15
#107/107 [=====] - ETA: 0s - loss: 0.5883 - acc: 0.6933
#Epoch 00002: val_acc improved from 0.75481 to 0.78846, saving model to model_current_best_parte3.h5
#107/107 [=====] - 37s 342ms/step - loss: 0.5883 - acc: 0.6933
- val_loss: 0.5039 - val_acc: 0.7885
#Epoch 3/15
#107/107 [=====] - ETA: 0s - loss: 0.5340 - acc: 0.7453
#Epoch 00003: val_acc did not improve from 0.78846
#107/107 [=====] - 36s 337ms/step - loss: 0.5340 - acc: 0.7453
- val_loss: 0.4969 - val_acc: 0.7788
#Epoch 4/15
#107/107 [=====] - ETA: 0s - loss: 0.5270 - acc: 0.7477
#Epoch 00004: val_acc did not improve from 0.78846
#107/107 [=====] - 38s 354ms/step - loss: 0.5270 - acc: 0.7477
- val_loss: 0.5627 - val_acc: 0.7404
#Epoch 5/15
#107/107 [=====] - ETA: 0s - loss: 0.5218 - acc: 0.7500
#Epoch 00005: val_acc did not improve from 0.78846
#107/107 [=====] - 35s 325ms/step - loss: 0.5218 - acc: 0.7500
- val_loss: 0.4952 - val_acc: 0.7452
#Epoch 6/15
#107/107 [=====] - ETA: 0s - loss: 0.4950 - acc: 0.7646
#Epoch 00006: val_acc improved from 0.78846 to 0.80288, saving model to model_current_best_parte3.h5
#107/107 [=====] - 35s 324ms/step - loss: 0.4950 - acc: 0.7646
- val_loss: 0.4983 - val_acc: 0.8029
#Epoch 7/15
#107/107 [=====] - ETA: 0s - loss: 0.4929 - acc: 0.7605
#Epoch 00007: val_acc did not improve from 0.80288
#107/107 [=====] - 34s 318ms/step - loss: 0.4929 - acc: 0.7605
```

```
- val_loss: 0.4594 - val_acc: 0.7692
#Epoch 8/15
#107/107 [=====] - ETA: 0s - loss: 0.5023 - acc: 0.7629
#Epoch 00008: val_acc did not improve from 0.80288
#107/107 [=====] - 34s 313ms/step - loss: 0.5023 - acc: 0.7629
- val_loss: 0.4606 - val_acc: 0.7788
#Epoch 9/15
#107/107 [=====] - ETA: 0s - loss: 0.4798 - acc: 0.7821
#Epoch 00009: val_acc did not improve from 0.80288
#107/107 [=====] - 36s 332ms/step - loss: 0.4798 - acc: 0.7821
- val_loss: 0.4438 - val_acc: 0.7885
#Epoch 10/15
#107/107 [=====] - ETA: 0s - loss: 0.4773 - acc: 0.7739
#Epoch 00010: val_acc did not improve from 0.80288
#107/107 [=====] - 36s 334ms/step - loss: 0.4773 - acc: 0.7739
- val_loss: 0.4323 - val_acc: 0.8029
#Epoch 11/15
#107/107 [=====] - ETA: 0s - loss: 0.4521 - acc: 0.7868
#Epoch 00011: val_acc improved from 0.80288 to 0.82212, saving model to model_current_best_parte3.h5
#107/107 [=====] - 37s 344ms/step - loss: 0.4521 - acc: 0.7868
- val_loss: 0.4097 - val_acc: 0.8221
#Epoch 12/15
#107/107 [=====] - ETA: 0s - loss: 0.4737 - acc: 0.7821
#Epoch 00012: val_acc did not improve from 0.82212
#107/107 [=====] - 36s 334ms/step - loss: 0.4737 - acc: 0.7821
- val_loss: 0.6449 - val_acc: 0.7596
#Epoch 13/15
#107/107 [=====] - ETA: 0s - loss: 0.4627 - acc: 0.7897
#Epoch 00013: val_acc did not improve from 0.82212
#107/107 [=====] - 36s 335ms/step - loss: 0.4627 - acc: 0.7897
- val_loss: 0.4149 - val_acc: 0.7981
#Epoch 14/15
#107/107 [=====] - ETA: 0s - loss: 0.4372 - acc: 0.8014
#Epoch 00014: val_acc did not improve from 0.82212
#107/107 [=====] - 38s 358ms/step - loss: 0.4372 - acc: 0.8014
- val_loss: 0.4171 - val_acc: 0.8077
#Epoch 15/15
#107/107 [=====] - ETA: 0s - loss: 0.4691 - acc: 0.7775
#Epoch 00015: val_acc did not improve from 0.82212
#107/107 [=====] - 37s 346ms/step - loss: 0.4691 - acc: 0.7775
- val_loss: 0.3932 - val_acc: 0.7933
```

Bueno, se llega a una val_acc de aprox 80%

```
Epoch 1/15
107/107 [=====] - ETA: 0s - loss: 0.4572 - acc: 0.7944
Epoch 00001: val_acc improved from -inf to 0.77404, saving model to model_current_best_parte3.h5
107/107 [=====] - 35s 326ms/step - loss: 0.4572 - acc: 0.7944 - val_loss: 0.4984 - val_acc: 0.7740
Epoch 2/15
107/107 [=====] - ETA: 0s - loss: 0.4566 - acc: 0.7950
Epoch 00002: val_acc improved from 0.77404 to 0.82212, saving model to model_current_best_parte3.h5
107/107 [=====] - 35s 323ms/step - loss: 0.4566 - acc: 0.7950 - val_loss: 0.3738 - val_acc: 0.8221
Epoch 3/15
107/107 [=====] - ETA: 0s - loss: 0.4331 - acc: 0.8067
Epoch 00003: val_acc did not improve from 0.82212
107/107 [=====] - 35s 325ms/step - loss: 0.4331 - acc: 0.8067 - val_loss: 0.4091 - val_acc: 0.8173
Epoch 4/15
107/107 [=====] - ETA: 0s - loss: 0.4332 - acc: 0.8067
Epoch 00004: val_acc did not improve from 0.82212
107/107 [=====] - 35s 325ms/step - loss: 0.4332 - acc: 0.8067 - val_loss: 0.4085 - val_acc: 0.8173
Epoch 5/15
107/107 [=====] - ETA: 0s - loss: 0.4412 - acc: 0.8002
Epoch 00005: val_acc improved from 0.82212 to 0.83173, saving model to model_current_best_parte3.h5
107/107 [=====] - 37s 342ms/step - loss: 0.4412 - acc: 0.8002 - val_loss: 0.3981 - val_acc: 0.8317
Epoch 6/15
107/107 [=====] - ETA: 0s - loss: 0.4207 - acc: 0.8096
Epoch 00006: val_acc did not improve from 0.83173
107/107 [=====] - 36s 335ms/step - loss: 0.4207 - acc: 0.8096 - val_loss: 0.3815 - val_acc: 0.8173
Epoch 7/15
107/107 [=====] - ETA: 0s - loss: 0.4188 - acc: 0.8218
Epoch 00007: val_acc improved from 0.83173 to 0.83654, saving model to model_current_best_parte3.h5
107/107 [=====] - 35s 328ms/step - loss: 0.4188 - acc: 0.8218 - val_loss: 0.3599 - val_acc: 0.8365
Epoch 8/15
107/107 [=====] - ETA: 0s - loss: 0.4294 - acc: 0.8090
Epoch 00008: val_acc did not improve from 0.83654
107/107 [=====] - 35s 329ms/step - loss: 0.4294 - acc: 0.8090 - val_loss: 0.3954 - val_acc: 0.8221
Epoch 9/15
107/107 [=====] - ETA: 0s - loss: 0.4250 - acc: 0.7956
Epoch 00009: val_acc did not improve from 0.83654
107/107 [=====] - 37s 342ms/step - loss: 0.4250 - acc: 0.7956 - val_loss: 0.4657 - val_acc: 0.7788
Epoch 10/15
107/107 [=====] - ETA: 0s - loss: 0.3978 - acc: 0.8236
```

```
Epoch 00010: val_acc did not improve from 0.83654
107/107 [=====] - 38s 353ms/step - loss: 0.3978 - 
acc: 0.8236 - val_loss: 0.4184 - val_acc: 0.8173
Epoch 11/15
107/107 [=====] - ETA: 0s - loss: 0.4031 - acc:
0.8172
Epoch 00011: val_acc did not improve from 0.83654
107/107 [=====] - 40s 372ms/step - loss: 0.4031 - 
acc: 0.8172 - val_loss: 0.3862 - val_acc: 0.8173
Epoch 12/15
107/107 [=====] - ETA: 0s - loss: 0.3977 - acc:
0.8224
Epoch 00012: val_acc did not improve from 0.83654
107/107 [=====] - 40s 377ms/step - loss: 0.3977 - 
acc: 0.8224 - val_loss: 0.3597 - val_acc: 0.8317
Epoch 13/15
107/107 [=====] - ETA: 0s - loss: 0.4013 - acc:
0.8283
Epoch 00013: val_acc did not improve from 0.83654
107/107 [=====] - 40s 376ms/step - loss: 0.4013 - 
acc: 0.8283 - val_loss: 0.3703 - val_acc: 0.8365
Epoch 14/15
107/107 [=====] - ETA: 0s - loss: 0.3930 - acc:
0.8183
Epoch 00014: val_acc did not improve from 0.83654
107/107 [=====] - 38s 352ms/step - loss: 0.3930 - 
acc: 0.8183 - val_loss: 0.4087 - val_acc: 0.8173
Epoch 15/15
107/107 [=====] - ETA: 0s - loss: 0.3948 - acc:
0.8277
Epoch 00015: val_acc did not improve from 0.83654
107/107 [=====] - 36s 337ms/step - loss: 0.3948 - 
acc: 0.8277 - val_loss: 0.3683 - val_acc: 0.8365
```

In [133]:

```
##### EJERCICIOS ANÁLISIS DE IMÁGENES

##### CNN (Reconocimiento de RX; distinción covid-pulmonia-normal)

from keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
from keras.models import Sequential, load_model, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, GlobalAveragePooling2D
from keras import backend as K
from keras.callbacks import ModelCheckpoint
from keras import regularizers
from keras import optimizers
from keras import models
from glob import glob
#!pip install opencv-python
import cv2
import os
from skimage.io import imread
from skimage import transform
from cv2 import resize

import matplotlib.pyplot as plt
import numpy as np

from IPython.display import SVG, display, clear_output
from keras.utils.vis_utils import model_to_dot

from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc

rx_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Rx Covid vs Pulmonia vs Normal'
rx_dir_output = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Rx Covid vs Pulmonia vs Normal\output'

batch_size = 16
#Rx Covid vs Pulmonia vs Normal

#!pip install split-folders tqdm
#import splitfolders
#splitfolders.ratio(rx_dir, output = rx_dir_output, seed=1337, ratio=(.8, 0.1,0.1));

# Antes teníamos una carpeta llamada 'Rx Covid vs Pulmonia vs Normal', en ella 'covid', 'pulmonia' y 'normal'
# con esto hemos creado una carpeta extra y dentro de 'caras' ahora tenemos 'covid', 'pulmonia', 'normal' y 'output'
# dentro de output tenemos ahora las carpetas de 'train', 'test' y 'val', dentro de las cuales 'covid', 'pulmonia', 'normal'

train_data_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Rx Covid vs Pulmonia vs Normal\output\train'
validation_data_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Rx Covid vs Pulmonia vs Normal\output\val'
test_data_dir = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Rx Covid vs Pulmonia vs Normal\output\test'

batch_size = 16

# data augmentation:
```

```
train_datagen = ImageDataGenerator(  
    #rescale=1. / 255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    fill_mode='nearest',  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True)  
  
preprocess_input = lambda x:x/255.  
  
# dimensiones a las que vamos a Llevar las imágenes  
img_width, img_height = 150, 150  
  
normed_dims = (img_height, img_width)  
  
train_datagen = ImageDataGenerator(  
    dtype='float32',  
    #rescale=1. / 255,  
    preprocessing_function = preprocess_input,  
    rotation_range=10,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    fill_mode='nearest',  
    shear_range=0.1,  
    zoom_range=0.1,  
    horizontal_flip=True)  
  
val_datagen = ImageDataGenerator(dtype='float32',  
                                 #rescale=1. / 255,  
                                 preprocessing_function = preprocess_input)  
  
test_datagen = ImageDataGenerator(dtype='float32',  
                                 #rescale=1. / 255,  
                                 preprocessing_function = preprocess_input)  
  
train_generator = train_datagen.flow_from_directory(  
    train_data_dir,  
    target_size=normed_dims,  
    batch_size=batch_size,  
    shuffle=True,  
    class_mode='categorical')  
  
validation_generator = val_datagen.flow_from_directory(  
    validation_data_dir,  
    target_size=normed_dims,  
    batch_size=batch_size,  
    shuffle=False,  
    class_mode='categorical')  
  
test_generator = test_datagen.flow_from_directory(  
    test_data_dir,  
    target_size=normed_dims,  
    batch_size=batch_size,  
    shuffle=False,  
    class_mode='categorical')  
  
print('input_shape:',normed_dims+(3,))  
# Model
```

```
model = Sequential()

model.add(Conv2D(32, (3,3), activation = 'relu', input_shape = normed_dims+(3,) ))

model.add(MaxPooling2D((2,2)))

model.add(Conv2D(32, (3,3), activation = 'relu'))

model.add(MaxPooling2D((2,2)))

model.add(Conv2D(64, (3,3), activation = 'relu'))

model.add(MaxPooling2D((2,2)))

model.add(Flatten())

# aqui empieza la red neuronal 'standard'
model.add(Dropout(0.5))

model.add(Dense(128, activation = 'relu'))

model.add(Dropout(0.5))

model.add(Dense(3, activation = 'softmax'))

model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics=['accuracy'])

#optimizer=optimizers.RMSprop(lr=1e-4),
#optimizer=optimizers.RMSprop(),
#metrics=['acc'])

model.summary()
```

Found 2334 images belonging to 3 classes.
Found 290 images belonging to 3 classes.
Found 295 images belonging to 3 classes.
input_shape: (150, 150, 3)
Model: "sequential_35"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_62 (Conv2D)	(None, 148, 148, 32)	896
<hr/>		
max_pooling2d_61 (MaxPooling)	(None, 74, 74, 32)	0
<hr/>		
conv2d_63 (Conv2D)	(None, 72, 72, 32)	9248
<hr/>		
max_pooling2d_62 (MaxPooling)	(None, 36, 36, 32)	0
<hr/>		
conv2d_64 (Conv2D)	(None, 34, 34, 64)	18496
<hr/>		
max_pooling2d_63 (MaxPooling)	(None, 17, 17, 64)	0
<hr/>		
flatten_21 (Flatten)	(None, 18496)	0
<hr/>		
dropout_44 (Dropout)	(None, 18496)	0
<hr/>		
dense_58 (Dense)	(None, 128)	2367616
<hr/>		
dropout_45 (Dropout)	(None, 128)	0
<hr/>		
dense_59 (Dense)	(None, 3)	387
<hr/>		
Total params: 2,396,643		
Trainable params: 2,396,643		
Non-trainable params: 0		

In [134]:

```
#### ojo no ejecutar que tarda una X mins cuidado!!!!
### 1.31-
modelpath="model_current_best_parte3.h5"

checkpoint = ModelCheckpoint(modelpath, monitor='val_acc', verbose=1,
                             save_best_only=True,
                             mode='max') # graba sólo Los que mejoran en validación
callbacks_list = [checkpoint]

def run_model():
    history = model.fit_generator(generator = train_generator,
                                   steps_per_epoch=train_generator.n // batch_size,
                                   epochs=5,
                                   callbacks=callbacks_list,
                                   verbose=1,
                                   shuffle = False,
                                   validation_data=validation_generator,
                                   validation_steps=validation_generator.n // batch_size
                               )

run_model()
```

Epoch 1/5
145/145 [=====] - ETA: 0s - loss: 0.7755 - accuracy: 0.6530WARNING:tensorflow:Can save best model only with val_acc available, skipping.
145/145 [=====] - 101s 693ms/step - loss: 0.7755 - accuracy: 0.6530 - val_loss: 0.7398 - val_accuracy: 0.6632
Epoch 2/5
145/145 [=====] - ETA: 0s - loss: 0.7492 - accuracy: 0.6582WARNING:tensorflow:Can save best model only with val_acc available, skipping.
145/145 [=====] - 94s 650ms/step - loss: 0.7492 - accuracy: 0.6582 - val_loss: 0.7187 - val_accuracy: 0.6562
Epoch 3/5
145/145 [=====] - ETA: 0s - loss: 0.7384 - accuracy: 0.6621WARNING:tensorflow:Can save best model only with val_acc available, skipping.
145/145 [=====] - 94s 651ms/step - loss: 0.7384 - accuracy: 0.6621 - val_loss: 0.7089 - val_accuracy: 0.6632
Epoch 4/5
145/145 [=====] - ETA: 0s - loss: 0.7328 - accuracy: 0.6647WARNING:tensorflow:Can save best model only with val_acc available, skipping.
145/145 [=====] - 100s 689ms/step - loss: 0.7328 - accuracy: 0.6647 - val_loss: 0.7027 - val_accuracy: 0.6701
Epoch 5/5
145/145 [=====] - ETA: 0s - loss: 0.7223 - accuracy: 0.6698WARNING:tensorflow:Can save best model only with val_acc available, skipping.
145/145 [=====] - 97s 669ms/step - loss: 0.7223 - accuracy: 0.6698 - val_loss: 0.6916 - val_accuracy: 0.6632

In [17]:

```
## Curso Udemy: Artificial Neural Networks (ANN) with keras in Python and R
```

Artificial Neural Networks (ANN)

La habilidad de las NN para crear sus propias reglas dado un conjunto de datos, y una vez con esas reglas poder hacer distinciones y separar en diferentes clases, esa habilidad de generar reglas, las hace muy versátiles y por ello muy útiles.

Vamos a ver en este curso acerca de las diferentes escalas de las NN como bien son el perceptrón o célula única (single-cell), del multi-nivel perceptrón (multiple cells), de backward-propagation y forward-propagation, del gradiente estocástico descendiente y de su implementación en Python.

Perceptrón: Vamos a ver su analogía con la biología y la neurona como unidad del sistema nervioso, un perceptrón va a ser una neurona artificial (AN) de tal forma que vamos a ver va a actuar como una neurona sigmoide, en una representación vectorial sería una función que recibiría un número "m" dado de inputs y produce un único output, todos ellos, los inputs y el output, son binarios. Hay varias formas en las que el conjunto de inputs nos pueden dar el output deseado, como que su suma ponderada supere un un dado threshold value o valor de umbral, y en caso de superarlo que sea '1' o no superarlo '0' para el output.

Hay que ver esa suma está ponderada tal que si los inputs son $\{x_1, x_2, \dots, x_m\}$ y tienen pesos (weights) asociados $\{w_1, w_2, \dots, w_m\}$ y entonces la magnitud podría ser

$$S = \sum_{i=1}^m (x_i * w_i)$$

if $S > \text{threshold} \Rightarrow \text{output} = 1$

else $\text{output} = 0$

Así podríamos recordar que esos pesos $\{w_1, w_2, \dots, w_m\}$ van a ser muy significantes en el problema ya que son la importancia de cada input, ya que estos inputs $\{x_1, x_2, \dots, x_m\}$ solo son magnitudes binarias acerca de si el input en cuestión cumple o no una característica, es decir, será tal que $x_i = \{0, 1\}$

Así, veremos cada objeto o dato tendrá una serie de posibles m inputs o características, y una vez evaluadas, ese objeto individual tendrá una magnitud S asociada, por lo cual podremos ver que será importante también cuál es el valor del umbral.

Ahora se puede extender el concepto de perceptrón de binario, que suena muy limitado, y realmente veremos se puede trabajar con valores continuos, tanto para el input como el output, y no tendremos porqué trabajar con esos valores binarios que nos limitan tanto. Así, podremos trabajar con valores reales y el resto de valores funcionaran de la misma forma (los pesos y el umbral).

Otra forma de ver estas operaciones anteriores como despejar el umbral de tal forma que

$$S = \sum_{i=1}^m (x_i * w_i) - \text{umbral}$$

$$\text{output} = 1 \text{ if } S > \text{threshold} ; \sum_{i=1}^m (x_i * w_i) > \text{umbral} ; \sum_{i=1}^m (x_i * w_i) - \text{umbral} > 0 ;$$

Y podemos ver un cambio en la notación: "-umbral = b", donde la letra "b" que es una constante se denota así por el "bias" o sesgo

$$\text{output} = 1 \text{ if } \sum_{i=1}^m (x_i * w_i) + b > 0$$

Y análogamente

$$\text{output} = 0 \text{ if } \sum_{i=1}^m (x_i * w_i) + b < 0$$

La representación de la función Output va a ser una función escalón, es decir, es 0 hasta que S supera un determinado valor en cuyo caso pasa a valer 1, es decir, esta es la función de activación.

Se va a llamar función de activación porque va a ser la que nos va a determinar cómo es el output, es una función en la que se va a satisfacer una condición o no, si supera un umbral/threshold.

Hay muchos otros tipos de funciones de activación, uno de los más populares es la "sigmoid function" que es una versión amortiguada de la función escalón. Es una especie de curva que "converge" en el 0 y 1 pero en vez de una transición en un solo punto, presenta ese cambio de una forma más suave. Funcionan también en la regresión logística.

La función sigmoide funciona mejor que la función escalón porque es menos sensible a la variación individual de una única observación. Al estar clasificando erróneamente algo, para arreglar esto un modelo lo que va a necesitar es nuevos valores de el bias y de los weights, y aquí es donde entra el problema, ya que cambios pequeños en los weights o en el bias van a acarrear cambios muy bruscos en esa función escalón, mientras que en la función sigmoide estos cambios son más graduales, por lo que esos cambios van a ser más fáciles de controlar.

Una AN individual con una función de activación sigmoide va a ser llamada "sigmoid neuron" o "logistic neuron".

La función sigmoide es

$$f(z) = 1/(1 + \exp(-z))$$

Y el output va a ser

$$\text{output} = 1/(1 + \exp(-\sum_{i=1}^m (x_i * w_i) - b))$$

Como bien sabemos es una la inversa de una función exponencial, por lo que tomará valores de

$$\text{output} = [0, 1]$$

y de la forma mencionada previamente. de tal forma que podemos ver el output será un valor en el continuo, por lo cual hemos roto esa barrera binaria.

In [29]:

```
##### Single-Perceptron Model

import numpy as np
import pandas as pd
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
# Un dataset con 3 tipos diferentes de flores (setosa, versicolour y virginica), longitudes del sépalo y del pétalo, en un # array de 150x4 tipo numpy array

iris = load_iris()

iris # Nos da la información de dichas características, las 4 columnas son: # 'Sepal Length', 'Sepal Width', 'Petal Length' y 'Petal Width'

iris.target # Nos da el resultado, el tipo de flores clasificado como '0', '1' ó '2 (setosa, versicolour ó virginica)

# Vamos a querer ver el problema como binario, es decir, no 3 características sino solo 2, y vamos a ver si es setosa o no
# (es decir, si no es setosa el default es que será versicolour ó virginica)

y = (iris.target == 0)
# Con esto vamos a modelar los 0's como True (es decir setosa como True), y los 1's y 2's como False (versicolour ó virginica
# como false)

y = (iris.target == 0).astype(np.int) # Para modelar los True como '1's y los False como '0's, y así no variables categóricas

X = iris.data[:,(2,3)] # En caso de no querer usar las 4 características sino solo 'Petal Length' y 'Petal Width'

perceptron = Perceptron(random_state = 42)
# Para recibir siempre el mismo resultado

# Perceptron() Es una función que simula un perceptrón y va a tener diferentes posibles parámetros

# 'penalty': constante que multiplica al término de regularización
# 'alpha': término de regularización y tiene como default 0.0001
# Y el resto de hiperparámetros que dejaremos como sus respectivos default

# Implementamos el modelo
perceptron.fit(X, y)

y_predicted = perceptron.predict(X)

y_predicted

accuracy_score(y, y_predicted)
print('Accuracy of the model:', accuracy_score(y, y_predicted))
# Y vemos es 1.0 por lo cual tenemos una accuracy del 100% y que hemos encontrado todos los casos perfectamente

# perceptron.coef_ # Como se ha realizado un ajuste de regresión Lineal lo que podemos ver aquí son los parámetros de esa recta
```

```
# perceptron.intercept_ # El corte con el origen
```

Accuracy of the model: 1.0

Out[29]:

```
array([4.])
```

Cómo funcionan las ANN. Gradient Descent Y CrossEntropy Error Function

Ahora vamos a "apilar" estas neuronas o células individuales de tal forma que vamos a formar los Networks, $AN + N = ANN$ y podremos tener finalmente nuestros Artificial Neural Networks. Hay dos formas de apilarlas, en paralelo o en secuencial.

El "parallel stacking" consiste en un mismo conjunto de inputs que vamos a pasar a diferentes neuronas y cada una nos va a dar un múltiples outputs diferentes dado un único conjunto de inputs.

El "sequential stacking" consiste en añadir una siguiente capa de neuronas, es decir, tendremos un conjunto de inputs y lo enviaremos a M diferentes neuronas, obteniendo así un conjunto de M outputs, hasta aquí todo esto es solo "parallel stacking", el "sequential stacking" aparece al colocar una segunda tanda M' de neuronas, cuyos inputs van a ser los outputs de las anteriores M neuronas, obteniendo así M' outputs nuevos. Una vez iterado el proceso podremos continuar añadiendo capas de neuronas hasta llegar a un output final.

Una pregunta que cabe hacerse ahora mismo es ¿por qué no podemos coger ese único conjunto de inputs y hacerlo pasar por una única neurona, perceptrón simple, y obtener directamente el único output? Es decir, sin necesidad de otras neuronas o de capas de neuronas. Y la respuesta es que, por ejemplo, en un problema de separación lineal se podrá encargar un único y perceptrón simple, pero cuando la situación es más complicada, como es casi siempre en la vida real, y necesitamos clasificar de forma no lineal, o Clústers..., necesitaremos más de un separador lineal, es decir, acabaremos teniendo una combinación de separadores lineales.

Como notación, podremos ver que este conjunto de inputs es la "input layer", el output es la "output layer" y las capas intermedias son las "hidden layers". El output layer será el último perceptrón que nos devolverá ese único output final.

Además, en el caso descrito podemos ver todos los outputs pasan a ser inputs de la siguiente layer, de tal forma que avanza en una única dirección, y diremos las ANN de este tipo serán "Feed Forward Network" mientras que si además, el output de una neurona de una capa pasa como input a todas las neuronas de la siguiente capa, se dirá la ANN será "Fully Connected Network". Si además un output volviese como input de esa misma neurona sería un "Cyclic Network".

Deep Learning es las Hidden Layers ya que es avanzar en secuencial lo que se puede ver como avanzar en profundidad. A mayor sea el número de capas, un network más profundo, y relaciones más complejas entre input y output.

¿Cómo funcionan exactamente las Neural Networks? Un poco de recapitulación, es que usaremos las neuronas individuales y estas van a ser neuronas sigmoides ya que van a actuar de forma más suave y por ello más manejable. Recordemos la fórmula era

$$\text{output} = \frac{1}{1 + \exp(-\sum_{i=1}^m (x_i * w_i) - b)}$$

Donde x_i es el conjunto de inputs, w_i los pesos asociados y b el bias. El problema del funcionamiento reside en encontrar los valores de los pesos y el bias para que el output calculado se parezca lo máximo posible al output predicho.

Para un caso sencillo de 2 neuronas en la para el input de 2 variables, y la neurona final, tendremos 2 pesos y 1 bias para cada una de las neuronas de entrada del input, y para la neurona final tendremos esos dos outputs por lo cual necesitaremos 2 pesos y 1 bias. Entonces vemos en total tenemos 6 pesos y 3 bias, es decir, 9 variables a determinar. ¿Cómo determinamos estos valores? Con Gradient Descent, GD, que es una

función de optimización que opera para buscar el mínimo de una función, de esta forma sabemos que hay otras funciones de optimización pero esta es mejor computacionalmente hablando que cualquier otras técnicas, ya que entrenará el modelo hasta una convergencia más rápida.

La forma en la que GD funciona es asignando un valor de w_i y b_i aleatorios inicialmente, calcular el output final asociado a este conjunto de parámetros, y finalmente vemos la función de error I asociado a esta predicción, (todo esto esto lo vamos a ver es hacia adelante, en secuencial y única dirección, Forward Propagation) ahora, cambiaremos los valores de w_i y b_i con intenciones de encontrar otros que minimicen esa función de error, es decir, volvemos atrás, realizamos un cambio en los parámetros y recalculamos la función output y luego la función de error nueva para estos nuevos parámetros (ahora hemos vuelto atrás, por lo que esto es Backward Propagation). Estamos buscando así cuáles de estos w_i y/o b_i son los que tienen mayor impacto y vamos a buscar variarlos para minimizar la función de error.

Con GD estamos buscando ese mínimo de esa función de tal forma que finalmente podemos avanzar hacia él, no buscando conocer la función del problema, lo cual tomaría demasiado tiempo, sino intentaríamos buscar una región pequeña en la que estemos y centrarnos en ella y buscar la dirección en la que avanzamos hacia abajo, es decir, nos vamos a estar basando en la pendiente de la función, que no es otra cosa que el gradiente y en dirección descendiente.

Ahora veremos qué es esta función de error I de la que estamos hablando de minimizar, una función que podemos utilizar es CrossEntropy Error Function:

$$I(y, y') = -y \log(y') - (1 - y) \log(1 - y')$$

y es el valor real del output e y' es el valor del output predicho, el por qué utilizamos esta función es porque se trata de una función sin mínimos locales, lo cual es imprescindible en un problema de mínimos, que es encontrar el mínimo global y no un mínimo relativo ya que no será la solución correcta. Como bien sabemos output varía entre 0 y 1 por ser una sigmoide, y queremos encontrar la función que nos permita ver

$$y' = \text{output} \rightarrow 1$$

Es decir, que tienda a uno, maximizar la solución. Así veremos variaremos los pesos y bias ligeramente (Dw_i y Db_i ; denotando los diferenciales de peso y bias respectivamente) hasta minimizar la función de error

$$w_i' = w_i - \alpha Dw_i$$

$$b_i' = b_i - \alpha Db_i$$

Donde α será el ratio de aprendizaje, y va a determinar el número de pasos que debemos de realizar hasta el mínimo en esa dirección hacia el lowest point. Si α es grande tenemos que tomar muchos pasos en dirección del fondo, la ventaja es que podemos avanzar más rápido, pero el problema es que nos podemos pasar del mínimo. En caso de estar demasiado cerca y tomar demasiados pasos, nos va a alejar del fondo. Es decir, α grande ayuda a converger rápido pero tiene problemas a la hora de converger.

Así pues, encontrar Dw_i y Db_i lo vamos a conseguir con Backward Propagation

Sabemos en un caso sencillo de un perceptrón como neurona sigmoide y un input de dos características (x_1, x_2):

$$y' \text{ output} = 1/(1 + \exp(-\sum_{j=1}^m (x_j w_j) - b)) = 1/(1 + \exp(-x_1 w_1 - x_2 w_2 - b))$$

$$I(y, y') = -y \log(y') - (1 - y) \log(1 - y')$$

Y sabemos

$$y = 1$$

Que a su vez implica

$$l(y=1,y') = -1/\log(y') - (1-1)\log(1-y') = \log(y')$$

Por lo cual procederíamos a calcular $l(y,y')$, y aquí es cuando entra en juega el Backpropagation, lo que queremos es minimizar la función de error, lo cual significa, derivada para ver este problema de mínimos

$$\frac{dl(y,y')}{dy} = \frac{d(\log(y'))}{dy} = -1/y' = -1/\text{output} = -(1 + \exp(-x_1w_1 - x_2w_2 - b))$$

Y si queremos hablar de la pendiente de activación y de su error va a ser recordar la función sigmoide como

$$y' = \text{output} = 1/(1 + \exp(-z))$$

y ver entonces

$$\frac{d(y')}{dz} = \exp(-z)/(1 + \exp(-z))^2$$

Donde sabemos que z es tal que

$$z = x_1w_1 + x_2w_2 + b$$

Y finalmente podremos ver las derivadas respecto de w_1 , w_2 y b

$$\frac{dz}{dw_1} = x_1$$

$$\frac{dz}{dw_2} = x_2$$

$$\frac{dz}{db} = 1$$

Y con todo esto podemos construir el impacto de cada una de estas variables en la función de error l por medio de la regla de la cadena

$$\frac{dl}{dw_1} = \frac{dl}{dy'} \frac{dy'}{dz} \frac{dz}{dw_1} = -(1 + \exp(-x_1w_1 - x_2w_2 - b)) \exp(-z)/(1 + \exp(-z))^2 x_1$$

$$\frac{dl}{dw_2} = \frac{dl}{dy'} \frac{dy'}{dz} \frac{dz}{dw_2} = -(1 + \exp(-x_1w_1 - x_2w_2 - b)) \exp(-z)/(1 + \exp(-z))^2 x_2$$

$$\frac{dl}{db} = \frac{dl}{dy'} \frac{dy'}{dz} \frac{dz}{db} = -(1 + \exp(-x_1w_1 - x_2w_2 - b)) \exp(-z)/(1 + \exp(-z))^2 1$$

Y así ver la variación de l en función de los pesos y el bias

Finalmente podremos ver

$$D^*w_1 = \frac{dl}{dw_1}$$

$$D^*w_2 = \frac{dl}{dw_2}$$

$$D^*b = \frac{dl}{db}$$

Y podremos sustituir esto en las expresiones anteriores

$$w_1' = w_1 - \alpha Dw_1 = w_1 - \alpha(-(1 + \exp(-x_1w_1 - x_2w_2 - b)) \exp(-z)/(1 + \exp(-z))^2 * x_1)$$

$$w_2' = w_2 - \alpha Dw_2 = w_2 - \alpha(-(1 + \exp(-x_1w_1 - x_2w_2 - b)) \exp(-z)/(1 + \exp(-z))^2 * x_2)$$

$$b' = b - \alpha Db = b - \alpha(-(1 + \exp(-x_1w_1 - x_2w_2 - b)) \exp(-z)/(1 + \exp(-z))^2 * 1)$$

Y de esta forma vamos a actualizar esos valores de los pesos y del bias, donde hemos usado la función sigmoide y la función CrossEntropy Error Function, de tal forma que estamos haciendo uso del GD para minimizar así esa función de error y poder actualizar esos valores de los pesos y bias de forma correcta.

Possibles preguntas sobre Activation Functions

Otra pregunta que podríamos hacernos es, ¿Por qué utilizamos funciones de activación? Es el que nos da el Output de una neurona, es decir, sin una función de activación lo único que tendríamos sería una suma ponderada de los Input pesados con los Pesos, tal que

$$z = \text{Sum}_{i=1 \text{ to } m} (x_i * w_i)$$

Donde tendríamos m Inputs x_i con sus respectivos pesos w_i pero no tendríamos una función donde evaluar z (recordemos que en el caso de la función activación sigmoide, z actúa como la magnitud a sustituir y por ello acabaríamos viendo $\text{Output}(z)$ es decir, como función de esta magnitud z). Para un problema de regresión, solo usar z podría ser aceptable, pero para un problema de clasificación no es suficiente, ya que queremos un $\text{Output} = [0, 1]$ o de tipo binario. Además, si solo trabajásemos con z en lugar del Output veríamos que solo podríamos establecer relaciones lineales con las neuronas, de tal forma que el resultado final del Input y el Output final, por más hidden layers que hubiese o neuronas en cada una de esas hidden layers, sería una relación lineal entre el Input y el Output final del ANN.

Por estos dos motivos es por los que usamos las funciones de activación, por un lado porque así podemos imponer condiciones de frontera en estos problemas de clasificación al Output, tanto en clasificación como en regresión. El segundo motivo es que podemos así añadir no linealidad, y poder ver patrones no lineales más complejos.

También otra pregunta que podríamos hacernos sería por qué hay varios tipos de funciones de activación, antes hemos mencionado la función escalón, y la función sigmoide. Otra función de activación muy similar sería la tangente hiperbólica $\tanh(x)$ ó $\text{tgh}(x)$ de tal forma que tiene diferentes límites de convergencia, las funciones escalón y sigmoide iban de 0 a 1, mientras que la tangente hiperbólica va de -1 a 1 siendo simétrica y pasando por el origen. Por tener esta centralidad en el origen es que vemos va a tener una mayor eficiencia en su convergencia que la sigmoide.

Otra posible función de activación es ReLU que viene de Rectified Linear Unit, y consiste en valor 0 constante como la función escalón, pero en el eje positivo de x tenemos una línea recta de una dada pendiente, es muy utilizada en las Hidden layers o layers intermedias de las NN de regresión. ReLU tiene el aspecto

$$\text{if } x \leq 0 ; \Rightarrow y = 0$$

$$\text{if } x < 0 ; \Rightarrow y = x$$

La diferencia con las anteriores es que el límite inferior es cero, como en las anteriores salvo la tangente hiperbólica que tenía -1, pero, en el caso de ReLU, no tiene límite superior.

Es una función que se ejecuta bien computacionalmente, sin mucho coste y se utiliza además en las hidden layers porque introduce no linealidad, sin embargo, en la Output layer no se suele utilizar ya que para problemas de clasificación no está acotada superiormente. Ocurre parecido a como ocurría en problemas de clasificación con la función escalón, que no servía para pequeños cambios por ese cambio tan brusco y por ser entonces una función tan difícil de controlar.

Finalmente podemos ver dónde usar cada tipo de función de activación, cualquiera de ellas nos sirve para clasificación, pero para regresión deberemos usar ReLU, y acerca de en qué neurona colocarlas, Step/Escalón por lo general como Output, Sigmoid/TanH generalmente como Hidden/Output layer y ReLu para las Hidden layers.

Otra pregunta a hacernos es acerca de si las Hidden layers y la Output layer pueden tener diferentes funciones de activación, la respuesta es que sí, ya que por lo general usaremos ReLU en las Hidden layers y Sigmoid en la Output Layer.

La siguiente pregunta es ¿Qué es clasificación multi-clase? ¿Tiene alguna función de activación que sea más conveniente de usar? La clasificación multclases es cuando tenemos más de dos posibles diferentes Output, para ellos tenemos un tipo de función de activación más específica llamada Softmax, funciona de forma parecida a Sigmoid pero tiene un paso adicional, y es colocar tantas Output Neurons como clases se tengan, si tenemos m clases, tendremos m Outputs entonces, sería colocar una última Hidden layer con m Neuronas y esas m Neuronas van a tener la Sigmoid Activation Function, y así el Output de cada uno de ellos será relativo en el rango [0,1] y entonces veremos cada uno de estos m Outputs nos va a dar cuenta de si es o no una característica. Además, veremos que para ver si es o no cada una de las m características una por una, requeriremos que sea una Fully-Connected Network.

Es decir, si fuese un algoritmo de detección de perros, gatos y loros, tendríamos 3 Output Neurons con Sigmoid y una de ellas nos diría información de probabilidad de ser perro o no, la siguiente de gato o no y la última de loro o no.

Posibles preguntas sobre Gradient Descent

Una pregunta que podemos hacernos es cuál es la diferencia entre Gradient Descent y Stochastic Gradient Descent. El tipo de Gradient Descent que hemos estudiado es Stochastic Gradient Descent, ya que hemos hecho la Forward Propagation, Backward Propagation, volviendo a iterar el proceso variando los pesos y el bias, el Gradient Descent consiste en lo mismo pero a todo el conjunto de entrenamiento y encontrar el error promedio e iterar. Otro posible método es coger fragmentos pequeños del Input o del conjunto de entrenamiento y es Mini Batch Gradient Descent y realizar exactamente lo mismo.

La diferencia entre Stochastic Gradient Descent y Gradient Descent es que SGD varía rápidamente los parámetros pero de forma muy abrupta y puede tener problemas para converger mientras que Gradient Descent varía lentamente pero converge de forma apropiada, es más lento porque actualiza todo el conjunto de training set o entrenamiento.

Épocas/Epoch

Una epoch es en ANN un ciclo a todo el conjunto entero de training data, es diferente de las iteraciones, las iteraciones van a ser las veces que tratemos con uno o varios datos pero no con su totalidad.

Hiperparámetros

Como una serie de recomendaciones veremos siempre tendremos una única Input Layer, Hidden Layers depende del problema concreto pero típicamente entre 1 y 5, y todas esas Hidden Layers con función de activación ReLU para introducir la no linealidad. Para diferentes tipos de problemas como para un problema binario tendremos 1 Output Neuron, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy, para un problema de clasificación multclase tendremos tantas Neuronas Output como clases tengamos, función de activación Softmax y una función de pérdida de tipo Cross-Entropy, y finalmente para un problema binario multicaracterística tendremos tantas Neuronas Output como características, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy.

Tenemos tantas Input Neuron como características haya. Tenemos además de 1-5 Hidden Layers, cada una de ellas con 10-100 Neuronas y ellas con función de activación ReLU. Finalmente tendremos tantas Output Neurons como características predichas, sin función de activación y todas ellas con función de pérdida ó error tipo MSE (Mean Squared Error).

Esos hiperparámetros que tendremos que dar, son los primeros que hemos mencionado, el número de Input

Keras

Keras es una librería o environment para el deep learning que tiene recursos para poder entrenar casi todos los problemas o modelos de Neural Networks, Keras trabaja a nivel de modelo, pero no maneja las operaciones a bajo nivel. Las manipulaciones de datos o trabajos a bajo nivel requieren de librerías más especializadas como son TensorFlow o Theano o CNTK... Lo bueno de Keras es que puede operar sin estas lower-level libraries, sin embargo, TensorFlow es la librería que más se adapta, más escalable, y con mayor producción.

TensorFlow por ser de bajo nivel, vamos a ver requiere de mayor poder de procesamiento, el cual en un ordenador va a venir dado por el GPU ó por el CPU. Vamos a usar Keras para definir el modelo y vamos a ver que Keras llame a TensorFlow para el backend para que se encargue de las operaciones.

In [2]:

```
##### Implementación/Instalación de Keras y TensorFlow
```

```
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

# !conda install tensorflow

# !conda install pip

# !pip install --upgrade tensorflow==2.0.0rc1

import tensorflow as tf
from tensorflow import keras

#keras.__version__
# '2.4.0'

#tf.__version__
# '2.3.1'
```

```
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\_distributor_init.py:3
 0: UserWarning: loaded more than 1 DLL from .libs:
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\.libs\libopenblas.NOIJ
JG62EMASZI6NYURL6JBKM4EVBGM7.gfortran-win_amd64.dll
C:\Users\PORTATIL\anaconda3\lib\site-packages\numpy\.libs\libopenblas.PYQH
XLVVQ7VESDPUVUADXEVJOBGHJPAY.gfortran-win_amd64.dll
  warnings.warn("loaded more than 1 DLL from .libs:\n%s" %
```

In [3]:

```
##### Problema de Clasificación

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras

# Vamos a tener un dataset de 60.000 imágenes para el train y 10.000 para el test. Las
# imágenes serán de 28x28 píxeles greyscale
# en una escala de 0 a 255. Van a ser 10 tipos de prendas o accesorios y queremos distinguirlos
# El dataset es muy conocido que está en keras, es el de 'fashion articles'

fashion_mnist = keras.datasets.fashion_mnist
(x_train_full, y_train_full), (x_test_full, y_test_full) = fashion_mnist.load_data()

object_number = 8

plt.imshow(x_train_full[object_number])
# Para poder visualizar las variables, las imágenes realmente

y_train_full[object_number]
# Es la clase y nos devuelve un número, correspondiente cada uno a una clase

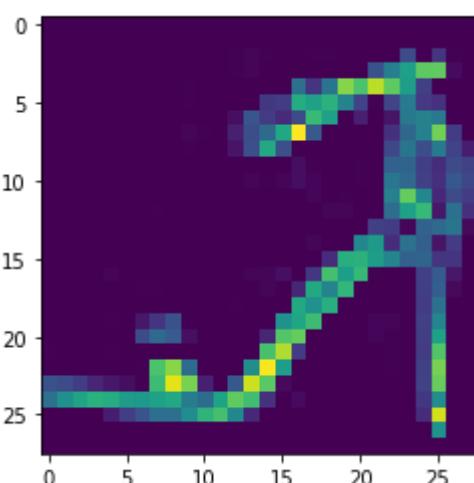
# Creamos una lista de las labels
class_names = ['T-shirt/Top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
               'Sneaker', 'Bag', 'Ankle Boot']

#Y así podemos ver a qué se corresponde
class_names[y_train_full[object_number]]

print('This item is a ', class_names[y_train_full[object_number]] )

# x_train_full[object_number]
# Nos va a dar una descripción numérica de esos píxeles que conforman las imágenes y se
# rán arrays de números de 0 a 255
```

This item is a Sandal



In [4]:

```
##### DATA NORMALIZATION AND TRAIN-TEST SPLIT

# Se divide entre 255 para normalizar ya que eran arrays de 0 a 255 para ir en esa grey scale
# Solo normalizamos x_train_full y x_test_full porque son las imágenes, recordemos y_train_full y y_test_full son las labels
x_train_n = x_train_full/255.0
x_test_n = x_test_full/255.0

# En un caso general de ML no sabríamos siempre la escala y deberíamos normalizarlo haciendo la suma y dividiendo por la
# desviación estándar, en este caso no es necesario por saber que es entre 0 y 255

# Y como vamos a usar Gradient Descent es que necesitamos normalizarlo y por ello dividimos por 255.0 para tenerlos entre 0 y 1,
# se divide por 255.0 y no 255 por si acaso es el resultado un entero o un real dependiendo de la versión de Python pero por si
# acaso así evitamos problemas

# Ahora vamos a dividirlo en Train-Test-Validation cogiendo los primeros 5.000 de test para validation

# Validation con 5.000 data
x_val = x_train_n[:5000]
y_val = y_train_full[:5000]

# Train con 55.000 valores
x_train = x_train_n[5000:]
y_train = y_train_full[5000:]

# Test con 10.000 valores
x_test = x_test_n
y_test = y_test_full
```

Model Creation

Vamos a ir con este problema de clasificación de imágenes, hay dos formas de crear un ANN en Keras:

- 1) Sequential model API: muy directa y muy sencilla, modelos sencillos layer por layer
- 2) Functional API: algo más complicada pero con mayor flexibilidad, como que una de las hidden layer tenga como input el de algunas capas atrás

In [5]:

```
##### 1) Sequential Model API for a Classification Problem

# Lo primero vamos a definir random_seed 42 para poder replicar el mismo modelo todas las veces
np.random.seed(42)
tf.random.set_seed(42)

# La estructura del modelo en este problema particular va a consistir en unos datos que van a ser 28x28 pixeles.
# Esos pixeles los vamos a llevar a la Input Layer, después vamos a colocar 2 Hidden Layer con función de activación
# de tipo ReLU, y finalmente el Output con 10 características/labels que queremos predecir, la Output Neuron tendrá
# una función de activación tipo Softmax

# Lo primero que tenemos que hacer es convertir esa matriz 2D que son los 28x28 pixeles en un array 1D de 784(=28*28) pixeles

model = keras.models.Sequential()

model.add(keras.layers.Flatten(input_shape = [28,28]))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(10, activation = 'softmax'))

# La primera es para definir el modelo como Sequential Model API, la siguiente es para convertir de 2D 28x28 a 1D 784 pixeles
# Las dos siguientes son las Hidden Layers, como bien sabemos función de activación ReLU, y unas 100 Neuronas por capa, y
# finalmente la Output Layer con 10 Neuronas, una por cada característica y función de activación Softmax

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 100)	78500
dense_1 (Dense)	(None, 100)	10100
dense_2 (Dense)	(None, 10)	1010
<hr/>		
Total params: 89,610		
Trainable params: 89,610		
Non-trainable params: 0		

In [21]:

```
# None significa "No limit in the input data"

#!pip uninstall pydot
#!pip uninstall pydotplus
#!pip uninstall graphviz

#!pip install pydot
#!pip install pydotplus
#!pip install pydotprint
#!pip install graphviz
import pydot

keras.utils.plot_model(model)
```

('Failed to import pydot. You must `pip install pydot` and install graphviz (<https://graphviz.gitlab.io/download/>), ', 'for `pydotprint` to work.')

In [24]:

```
weights, biases = model.layers[1].get_weights()
print('weights.shape:', weights.shape, '& biases.shape:', biases.shape)

weights.shape: (784, 100) & biases.shape: (100,)
```

In [27]:

```
# Ahora vamos a ver que antes de entrenar el modelo, tenemos que ver la tasa de aprendizaje

model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics = ['accuracy'])
# Loss = 'sparse_categorical_crossentropy'; porque tenemos el data en 10 categorías diferentes, si tuviésemos probabilidades
# para cada característica entonces tendríamos que usar 'categorical_crossentropy', si fueran muchas y binarias sería
# 'binary_crossentropy'

# optimizer = 'sgd'; SGD Stochastic Gradient Descent, que le estamos contando a Keras que use un algoritmo de backpropagation

# metrics = ['accuracy'] ; por estar usando clasificación, estaremos usando 'accuracy', si quisiésemos regresión, deberíamos de
# usar 'Mean Squared Error'

model_history = model.fit(x_train, y_train, epochs = 30, validation_data = (x_val, y_val))
```

Epoch 1/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2856 -
accuracy: 0.8963 - val_loss: 0.3160 - val_accuracy: 0.8850
Epoch 2/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2811 -
accuracy: 0.8974 - val_loss: 0.3156 - val_accuracy: 0.8876
Epoch 3/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2767 -
accuracy: 0.9001 - val_loss: 0.4002 - val_accuracy: 0.8554
Epoch 4/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2730 -
accuracy: 0.9011 - val_loss: 0.3177 - val_accuracy: 0.8828
Epoch 5/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2708 -
accuracy: 0.9015 - val_loss: 0.3128 - val_accuracy: 0.8842
Epoch 6/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2648 -
accuracy: 0.9041 - val_loss: 0.3126 - val_accuracy: 0.8886
Epoch 7/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2625 -
accuracy: 0.9057 - val_loss: 0.3170 - val_accuracy: 0.8874
Epoch 8/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2589 -
accuracy: 0.9066 - val_loss: 0.3272 - val_accuracy: 0.8768
Epoch 9/30
1719/1719 [=====] - 4s 2ms/step - loss: 0.2556 -
accuracy: 0.9071 - val_loss: 0.3172 - val_accuracy: 0.8800
Epoch 10/30
1719/1719 [=====] - 4s 2ms/step - loss: 0.2522 -
accuracy: 0.9094 - val_loss: 0.3134 - val_accuracy: 0.8832
Epoch 11/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2495 -
accuracy: 0.9102 - val_loss: 0.3116 - val_accuracy: 0.8862
Epoch 12/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2453 -
accuracy: 0.9114 - val_loss: 0.3174 - val_accuracy: 0.8842
Epoch 13/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2427 -
accuracy: 0.9122 - val_loss: 0.3127 - val_accuracy: 0.8848
Epoch 14/30
1719/1719 [=====] - 4s 2ms/step - loss: 0.2402 -
accuracy: 0.9125 - val_loss: 0.3283 - val_accuracy: 0.8790
Epoch 15/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2361 -
accuracy: 0.9154 - val_loss: 0.3091 - val_accuracy: 0.8882
Epoch 16/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2339 -
accuracy: 0.9159 - val_loss: 0.3036 - val_accuracy: 0.8870
Epoch 17/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2322 -
accuracy: 0.9155 - val_loss: 0.3288 - val_accuracy: 0.8814
Epoch 18/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2286 -
accuracy: 0.9179 - val_loss: 0.3068 - val_accuracy: 0.8912
Epoch 19/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2259 -
accuracy: 0.9191 - val_loss: 0.3053 - val_accuracy: 0.8888
Epoch 20/30
1719/1719 [=====] - 3s 2ms/step - loss: 0.2233 -
accuracy: 0.9190 - val_loss: 0.3313 - val_accuracy: 0.8790
Epoch 21/30

```
1719/1719 [=====] - 3s 2ms/step - loss: 0.2205 -  
accuracy: 0.9207 - val_loss: 0.3103 - val_accuracy: 0.8918  
Epoch 22/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2179 -  
accuracy: 0.9210 - val_loss: 0.3002 - val_accuracy: 0.8916  
Epoch 23/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2156 -  
accuracy: 0.9221 - val_loss: 0.3075 - val_accuracy: 0.8860  
Epoch 24/30  
1719/1719 [=====] - 4s 2ms/step - loss: 0.2125 -  
accuracy: 0.9237 - val_loss: 0.3190 - val_accuracy: 0.8832  
Epoch 25/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2100 -  
accuracy: 0.9240 - val_loss: 0.3072 - val_accuracy: 0.8900  
Epoch 26/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2078 -  
accuracy: 0.9247 - val_loss: 0.3226 - val_accuracy: 0.8852  
Epoch 27/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2056 -  
accuracy: 0.9269 - val_loss: 0.3111 - val_accuracy: 0.8906  
Epoch 28/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2032 -  
accuracy: 0.9263 - val_loss: 0.3134 - val_accuracy: 0.8890  
Epoch 29/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2012 -  
accuracy: 0.9271 - val_loss: 0.3159 - val_accuracy: 0.8848  
Epoch 30/30  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1984 -  
accuracy: 0.9278 - val_loss: 0.3210 - val_accuracy: 0.8894
```

In [33]:

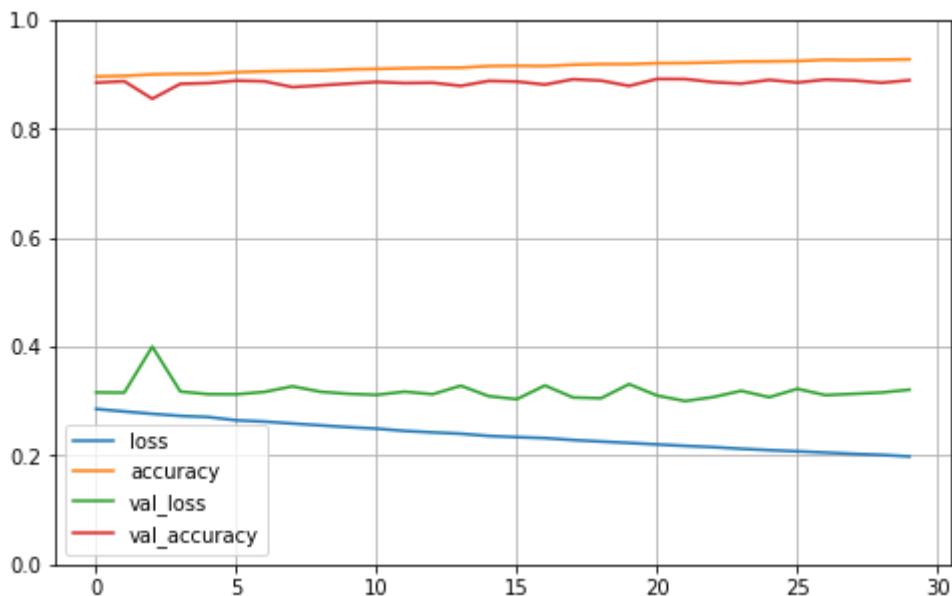
```
# Podemos ver la accuracy crece con cada epoch hasta llegar a un 92.78%, lo cual está
# muy bien, y obviamente es mejor que la
# primera de las accuracy en esa primera epoch de 89.73%
# model_history.params
# model_history.history

import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

# Con cada época la accuracy, y la validation accuracy crecen con cada época, mientras
# que las pérdidas decrecen.
# También se puede ver el modelo no ha convergido, ambas rectas siguen siendo monótonas
# pero no paran, habría que
# aumentar el número de épocas/epochs

# Vamos a aumentar el #epochs de 30 a más
```



In [34]:

```
model = keras.models.Sequential()

model.add(keras.layers.Flatten(input_shape = [28,28]))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(100, activation = 'relu'))

model.add(keras.layers.Dense(10, activation = 'softmax'))

model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics = [
'accuracy'])

model_history = model.fit(x_train, y_train, epochs = 100, validation_data = (x_val, y_val))
```

Epoch 1/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.7926 -
accuracy: 0.7331 - val_loss: 0.5580 - val_accuracy: 0.8064
Epoch 2/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.5122 -
accuracy: 0.8197 - val_loss: 0.4578 - val_accuracy: 0.8482
Epoch 3/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.4622 -
accuracy: 0.8359 - val_loss: 0.5353 - val_accuracy: 0.7982
Epoch 4/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.4344 -
accuracy: 0.8476 - val_loss: 0.4145 - val_accuracy: 0.8612
Epoch 5/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.4158 -
accuracy: 0.8539 - val_loss: 0.4008 - val_accuracy: 0.8624
Epoch 6/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3978 -
accuracy: 0.8598 - val_loss: 0.3925 - val_accuracy: 0.8642
Epoch 7/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3858 -
accuracy: 0.8637 - val_loss: 0.3837 - val_accuracy: 0.8662
Epoch 8/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3748 -
accuracy: 0.8663 - val_loss: 0.4026 - val_accuracy: 0.8538
Epoch 9/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3649 -
accuracy: 0.8705 - val_loss: 0.3743 - val_accuracy: 0.8636
Epoch 10/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.3558 -
accuracy: 0.8742 - val_loss: 0.3696 - val_accuracy: 0.8650
Epoch 11/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.3474 -
accuracy: 0.8759 - val_loss: 0.3607 - val_accuracy: 0.8710
Epoch 12/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3389 -
accuracy: 0.8782 - val_loss: 0.3499 - val_accuracy: 0.8754
Epoch 13/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3321 -
accuracy: 0.8808 - val_loss: 0.3488 - val_accuracy: 0.8754
Epoch 14/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3256 -
accuracy: 0.8831 - val_loss: 0.3671 - val_accuracy: 0.8686
Epoch 15/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3184 -
accuracy: 0.8846 - val_loss: 0.3378 - val_accuracy: 0.8766
Epoch 16/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3135 -
accuracy: 0.8870 - val_loss: 0.3272 - val_accuracy: 0.8822
Epoch 17/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3082 -
accuracy: 0.8888 - val_loss: 0.3618 - val_accuracy: 0.8678
Epoch 18/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.3024 -
accuracy: 0.8911 - val_loss: 0.3358 - val_accuracy: 0.8820
Epoch 19/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2978 -
accuracy: 0.8923 - val_loss: 0.3236 - val_accuracy: 0.8854
Epoch 20/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2929 -
accuracy: 0.8944 - val_loss: 0.3416 - val_accuracy: 0.8756
Epoch 21/100

```
1719/1719 [=====] - 3s 2ms/step - loss: 0.2880 -  
accuracy: 0.8960 - val_loss: 0.3177 - val_accuracy: 0.8876  
Epoch 22/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2835 -  
accuracy: 0.8974 - val_loss: 0.3113 - val_accuracy: 0.8870  
Epoch 23/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2798 -  
accuracy: 0.8980 - val_loss: 0.3120 - val_accuracy: 0.8910  
Epoch 24/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.2757 -  
accuracy: 0.9007 - val_loss: 0.3179 - val_accuracy: 0.8860  
Epoch 25/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2715 -  
accuracy: 0.9015 - val_loss: 0.3117 - val_accuracy: 0.8900  
Epoch 26/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2678 -  
accuracy: 0.9037 - val_loss: 0.3184 - val_accuracy: 0.8844  
Epoch 27/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2651 -  
accuracy: 0.9050 - val_loss: 0.3046 - val_accuracy: 0.8914  
Epoch 28/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2615 -  
accuracy: 0.9056 - val_loss: 0.3119 - val_accuracy: 0.8872  
Epoch 29/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2571 -  
accuracy: 0.9077 - val_loss: 0.3226 - val_accuracy: 0.8840  
Epoch 30/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2547 -  
accuracy: 0.9087 - val_loss: 0.3238 - val_accuracy: 0.8866  
Epoch 31/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2509 -  
accuracy: 0.9085 - val_loss: 0.3110 - val_accuracy: 0.8906  
Epoch 32/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2478 -  
accuracy: 0.9105 - val_loss: 0.3049 - val_accuracy: 0.8928  
Epoch 33/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2446 -  
accuracy: 0.9110 - val_loss: 0.3139 - val_accuracy: 0.8936  
Epoch 34/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2411 -  
accuracy: 0.9119 - val_loss: 0.3097 - val_accuracy: 0.8924  
Epoch 35/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.2379 -  
accuracy: 0.9146 - val_loss: 0.2997 - val_accuracy: 0.8918  
Epoch 36/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2363 -  
accuracy: 0.9146 - val_loss: 0.3078 - val_accuracy: 0.8928  
Epoch 37/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2330 -  
accuracy: 0.9155 - val_loss: 0.3039 - val_accuracy: 0.8922  
Epoch 38/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2297 -  
accuracy: 0.9172 - val_loss: 0.2971 - val_accuracy: 0.8956  
Epoch 39/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2286 -  
accuracy: 0.9176 - val_loss: 0.3020 - val_accuracy: 0.8936  
Epoch 40/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.2249 -  
accuracy: 0.9191 - val_loss: 0.3047 - val_accuracy: 0.8962  
Epoch 41/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.2220 -
```

```
accuracy: 0.9210 - val_loss: 0.2935 - val_accuracy: 0.8978
Epoch 42/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2204 -
accuracy: 0.9219 - val_loss: 0.3051 - val_accuracy: 0.8930
Epoch 43/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2163 -
accuracy: 0.9218 - val_loss: 0.3339 - val_accuracy: 0.8788
Epoch 44/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2150 -
accuracy: 0.9234 - val_loss: 0.3081 - val_accuracy: 0.8914
Epoch 45/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.2129 -
accuracy: 0.9227 - val_loss: 0.3083 - val_accuracy: 0.8868
Epoch 46/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2091 -
accuracy: 0.9247 - val_loss: 0.3102 - val_accuracy: 0.8922
Epoch 47/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2074 -
accuracy: 0.9256 - val_loss: 0.3153 - val_accuracy: 0.8922
Epoch 48/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2049 -
accuracy: 0.9263 - val_loss: 0.3127 - val_accuracy: 0.8906
Epoch 49/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.2019 -
accuracy: 0.9272 - val_loss: 0.3006 - val_accuracy: 0.8946
Epoch 50/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1998 -
accuracy: 0.9284 - val_loss: 0.3127 - val_accuracy: 0.8930
Epoch 51/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1981 -
accuracy: 0.9287 - val_loss: 0.2994 - val_accuracy: 0.8938
Epoch 52/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.1961 -
accuracy: 0.9299 - val_loss: 0.3150 - val_accuracy: 0.8898
Epoch 53/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1941 -
accuracy: 0.9304 - val_loss: 0.3115 - val_accuracy: 0.8900
Epoch 54/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.1921 -
accuracy: 0.9313 - val_loss: 0.2964 - val_accuracy: 0.8948
Epoch 55/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1890 -
accuracy: 0.9322 - val_loss: 0.3165 - val_accuracy: 0.8928
Epoch 56/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1878 -
accuracy: 0.9324 - val_loss: 0.3059 - val_accuracy: 0.8956
Epoch 57/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1848 -
accuracy: 0.9332 - val_loss: 0.3108 - val_accuracy: 0.8902
Epoch 58/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1823 -
accuracy: 0.9340 - val_loss: 0.3068 - val_accuracy: 0.8944
Epoch 59/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1817 -
accuracy: 0.9361 - val_loss: 0.3055 - val_accuracy: 0.8978
Epoch 60/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1790 -
accuracy: 0.9370 - val_loss: 0.2975 - val_accuracy: 0.8972
Epoch 61/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1771 -
accuracy: 0.9363 - val_loss: 0.2936 - val_accuracy: 0.8992
```

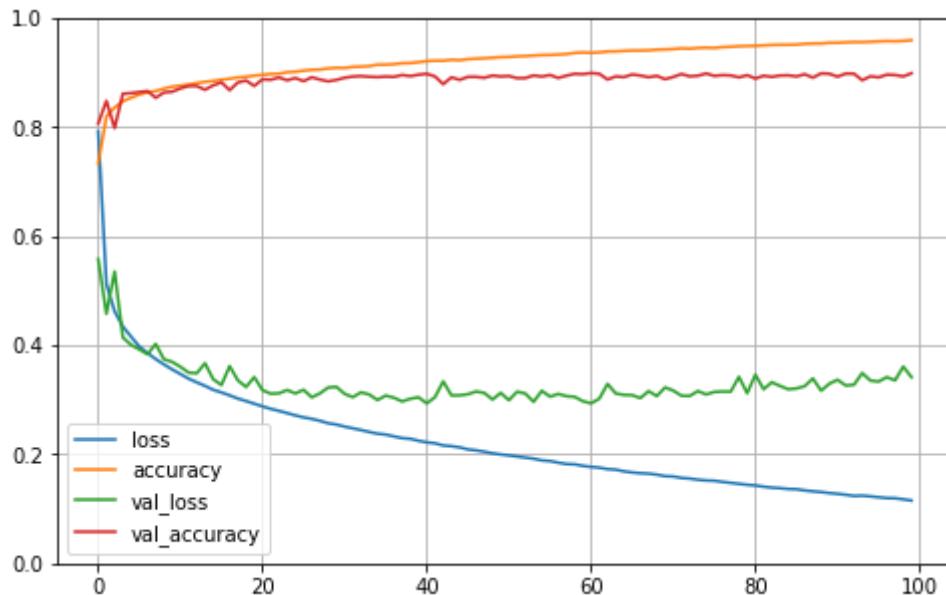
```
Epoch 62/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1756 -
accuracy: 0.9371 - val_loss: 0.3021 - val_accuracy: 0.8978
Epoch 63/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1730 -
accuracy: 0.9388 - val_loss: 0.3292 - val_accuracy: 0.8874
Epoch 64/100
1719/1719 [=====] - 4s 2ms/step - loss: 0.1720 -
accuracy: 0.9389 - val_loss: 0.3117 - val_accuracy: 0.8930
Epoch 65/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1688 -
accuracy: 0.9398 - val_loss: 0.3090 - val_accuracy: 0.8918
Epoch 66/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1669 -
accuracy: 0.9405 - val_loss: 0.3086 - val_accuracy: 0.8966
Epoch 67/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1654 -
accuracy: 0.9404 - val_loss: 0.3032 - val_accuracy: 0.8946
Epoch 68/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1647 -
accuracy: 0.9409 - val_loss: 0.3159 - val_accuracy: 0.8916
Epoch 69/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1628 -
accuracy: 0.9421 - val_loss: 0.3069 - val_accuracy: 0.8940
Epoch 70/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1600 -
accuracy: 0.9425 - val_loss: 0.3227 - val_accuracy: 0.8880
Epoch 71/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1595 -
accuracy: 0.9432 - val_loss: 0.3170 - val_accuracy: 0.8920
Epoch 72/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1567 -
accuracy: 0.9444 - val_loss: 0.3075 - val_accuracy: 0.8974
Epoch 73/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1557 -
accuracy: 0.9439 - val_loss: 0.3071 - val_accuracy: 0.8932
Epoch 74/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1535 -
accuracy: 0.9451 - val_loss: 0.3161 - val_accuracy: 0.8942
Epoch 75/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1521 -
accuracy: 0.9457 - val_loss: 0.3099 - val_accuracy: 0.8984
Epoch 76/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1517 -
accuracy: 0.9452 - val_loss: 0.3145 - val_accuracy: 0.8940
Epoch 77/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1495 -
accuracy: 0.9468 - val_loss: 0.3152 - val_accuracy: 0.8954
Epoch 78/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1478 -
accuracy: 0.9474 - val_loss: 0.3151 - val_accuracy: 0.8948
Epoch 79/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1459 -
accuracy: 0.9484 - val_loss: 0.3421 - val_accuracy: 0.8914
Epoch 80/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1442 -
accuracy: 0.9489 - val_loss: 0.3121 - val_accuracy: 0.8952
Epoch 81/100
1719/1719 [=====] - 3s 2ms/step - loss: 0.1430 -
accuracy: 0.9491 - val_loss: 0.3460 - val_accuracy: 0.8892
Epoch 82/100
```

```
1719/1719 [=====] - 3s 2ms/step - loss: 0.1407 -  
accuracy: 0.9499 - val_loss: 0.3194 - val_accuracy: 0.8942  
Epoch 83/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1390 -  
accuracy: 0.9507 - val_loss: 0.3322 - val_accuracy: 0.8922  
Epoch 84/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1380 -  
accuracy: 0.9510 - val_loss: 0.3255 - val_accuracy: 0.8946  
Epoch 85/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1365 -  
accuracy: 0.9511 - val_loss: 0.3194 - val_accuracy: 0.8950  
Epoch 86/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1361 -  
accuracy: 0.9514 - val_loss: 0.3209 - val_accuracy: 0.8930  
Epoch 87/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1338 -  
accuracy: 0.9528 - val_loss: 0.3251 - val_accuracy: 0.8970  
Epoch 88/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1322 -  
accuracy: 0.9534 - val_loss: 0.3391 - val_accuracy: 0.8908  
Epoch 89/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1309 -  
accuracy: 0.9533 - val_loss: 0.3168 - val_accuracy: 0.8986  
Epoch 90/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1290 -  
accuracy: 0.9547 - val_loss: 0.3296 - val_accuracy: 0.8978  
Epoch 91/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1277 -  
accuracy: 0.9547 - val_loss: 0.3368 - val_accuracy: 0.8926  
Epoch 92/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1259 -  
accuracy: 0.9555 - val_loss: 0.3262 - val_accuracy: 0.8982  
Epoch 93/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1238 -  
accuracy: 0.9560 - val_loss: 0.3276 - val_accuracy: 0.8980  
Epoch 94/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1243 -  
accuracy: 0.9558 - val_loss: 0.3490 - val_accuracy: 0.8860  
Epoch 95/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1230 -  
accuracy: 0.9565 - val_loss: 0.3353 - val_accuracy: 0.8932  
Epoch 96/100  
1719/1719 [=====] - 4s 2ms/step - loss: 0.1213 -  
accuracy: 0.9573 - val_loss: 0.3336 - val_accuracy: 0.8914  
Epoch 97/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1198 -  
accuracy: 0.9578 - val_loss: 0.3416 - val_accuracy: 0.8962  
Epoch 98/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1196 -  
accuracy: 0.9575 - val_loss: 0.3357 - val_accuracy: 0.8956  
Epoch 99/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1175 -  
accuracy: 0.9582 - val_loss: 0.3609 - val_accuracy: 0.8930  
Epoch 100/100  
1719/1719 [=====] - 3s 2ms/step - loss: 0.1154 -  
accuracy: 0.9594 - val_loss: 0.3416 - val_accuracy: 0.8988
```

In [35]:

```
import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [37]:

```
# Ahora vamos a ver cómo evaluar el modelo y cómo predecir clases usando el modelo
print('loss, accuracy: ', model.evaluate(x_test, y_test))
```

```
313/313 [=====] - 0s 1ms/step - loss: 0.3864 - accuracy: 0.8871
loss, accuracy: [0.3864385485649109, 0.8870999813079834]
```

In [48]:

```
# Vamos a coger ahora "nuevos" datos y ver qué clase predice

x_new = x_test[:3]

y_pred_new = model.predict_classes(x_new)
y_pred_new

print('The objects are:',np.array(class_names)[y_pred_new], '\n and the images are')

plt.imshow(x_new[0])
plt.title(np.array(class_names)[y_pred_new[0]])
plt.show()

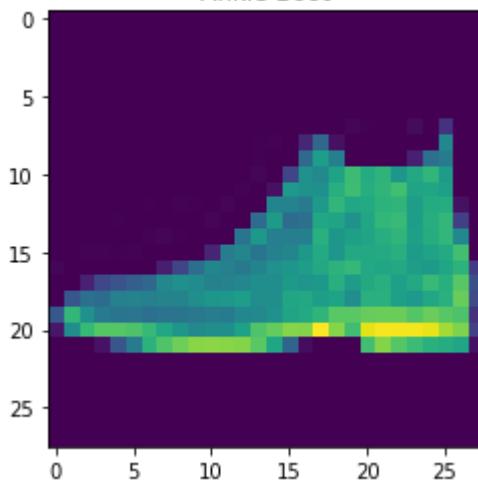
plt.imshow(x_new[1])
plt.title(np.array(class_names)[y_pred_new[1]])
plt.show()

plt.imshow(x_new[2])
plt.title(np.array(class_names)[y_pred_new[2]])
plt.show()

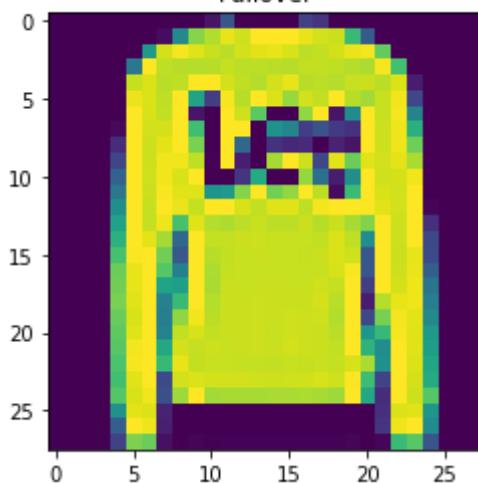
# Y podemos ver funciona
```

The objects are: ['Ankle Boot' 'Pullover' 'Trouser']
and the images are

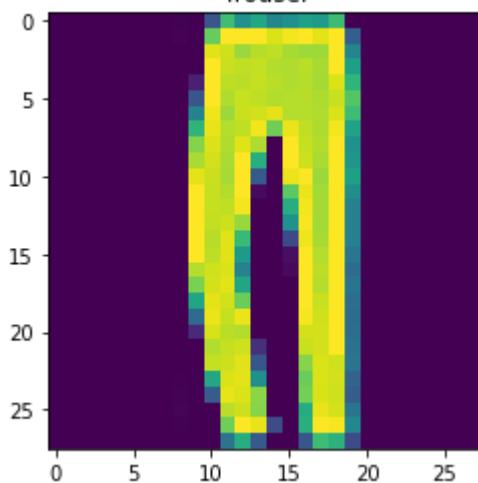
Ankle Boot



Pullover



Trouser



In [59]:

```
##### 1) Sequential Model API for a Regression Problem

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib as mlp
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras

# Un dataset muy común para la regresión, el objetivo aquí es la predicción del precio
# de una casa usando diferentes variables
# independientes
from sklearn.datasets import fetch_california_housing
# Un dataset de 20.640 datos. Son 8 atributos o características diferentes

housing = fetch_california_housing()

#print('feature_names:',housing.feature_names)
# Las diferentes variables que observamos son 'target', y los nombres de los atributos
# son los siguientes
# {'MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup', 'Latitude',
# 'Longitude'}

# MedInc: median income in block, HouseAge: media de las edades de las casas, AveRooms
# y AveBedrms como promedio de habitaciones
# y de dormitorios, Population como población promedio, AveOccup como ocupación promedi
# o de esas casas y las últimas como la ubi

# La variable objetivo es la media de la casa en unidades de 100.000 distritos de Calif
# ornia

# Separamos en train, test y val usando un truquito, primero separar en train_full y te
# st y de train_full separarlo en train y
# val
from sklearn.model_selection import train_test_split

# como default es un 25%, así tendremos un 75%trainfull y 25%test
x_train_full, x_test, y_train_full, y_test = train_test_split(housing.data, housing.tar
get, random_state = 42)

# ahora tendremos un 25% del 75% que es 75%/4=18.75% del val y por ello un 61.25% tra
n
x_train, x_val, y_train, y_val = train_test_split(x_train_full, y_train_full, random_st
ate = 42)

# Y ahora tenemos:
# x_train, x_test, x_val
# y_train, y_test, y_val

# Procedemos a normalizar los datos
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
x_val = scaler.transform(x_val)
```

```
x_train.shape
# (11610, 8)
# Tenemos 11610 datos en las 8 características que hemos mencionado, son solo 11610 ya
# que es un 61.25% de los 20.640 datos

# Lo siguiente es que vamos a definir random_seed 42 para poder replicar el mismo modelo
# o todas las veces
np.random.seed(42)
tf.random.set_seed(42)
```

In [64]:

```
# Vamos ahora a generar el modelo de ANN para regresión con 2 Hidden Layers y sin función de activación en la única Output
# Neuron y le vamos a pasar también en la primera de estas Input Layer el número de características o variables independientes.
# Otra forma más genérica de escribirlo sería:

# 'num = x_train.shape[:1]
# model.add(keras.layers.Dense(30, activation = 'relu', input_shape = [num]))'

model = keras.models.Sequential()

model.add(keras.layers.Dense(30, activation = 'relu', input_shape = [8]))

model.add(keras.layers.Dense(30, activation = 'relu'))

model.add(keras.layers.Dense(1))

model.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
<hr/>		
dense_12 (Dense)	(None, 30)	270
dense_13 (Dense)	(None, 30)	930
dense_14 (Dense)	(None, 1)	31
<hr/>		
Total params:	1,231	
Trainable params:	1,231	
Non-trainable params:	0	

In [65]:

```
model.compile(loss = 'mean_squared_error', optimizer = keras.optimizers.SGD(lr = 1e-3),
metrics = ['mae'])
# Lr: el Learning rate
# mae: Mean Absolute Error

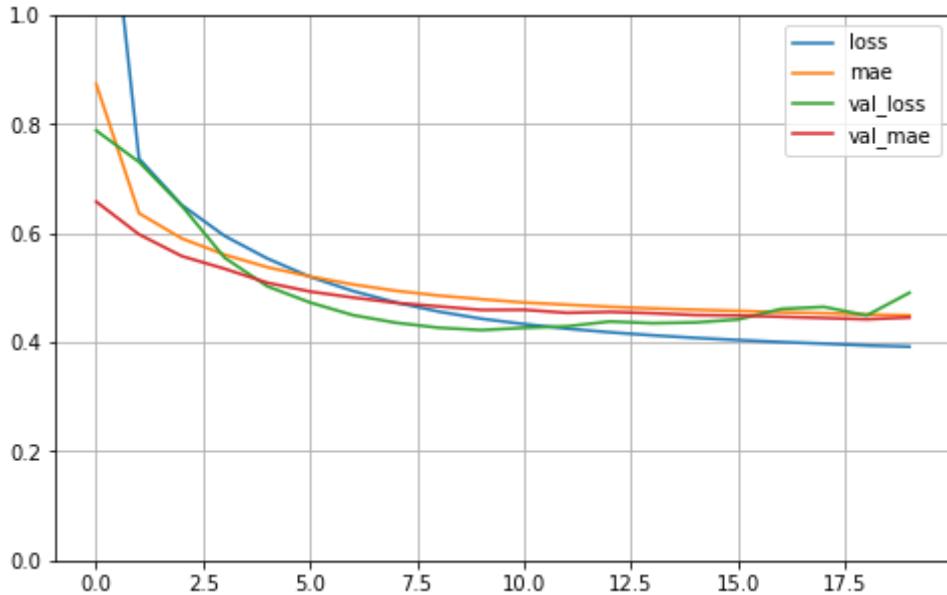
model_history = model.fit(x_train, y_train, epochs = 20, validation_data = (x_val, y_val))
```

```
Epoch 1/20
363/363 [=====] - 2s 6ms/step - loss: 1.4581 - mae: 0.8737 - val_loss: 0.7883 - val_mae: 0.6579
Epoch 2/20
363/363 [=====] - 1s 2ms/step - loss: 0.7368 - mae: 0.6366 - val_loss: 0.7306 - val_mae: 0.5979
Epoch 3/20
363/363 [=====] - 1s 2ms/step - loss: 0.6516 - mae: 0.5904 - val_loss: 0.6501 - val_mae: 0.5580
Epoch 4/20
363/363 [=====] - 1s 3ms/step - loss: 0.5951 - mae: 0.5606 - val_loss: 0.5551 - val_mae: 0.5348
Epoch 5/20
363/363 [=====] - 1s 2ms/step - loss: 0.5536 - mae: 0.5376 - val_loss: 0.5025 - val_mae: 0.5091
Epoch 6/20
363/363 [=====] - 1s 2ms/step - loss: 0.5199 - mae: 0.5206 - val_loss: 0.4728 - val_mae: 0.4930
Epoch 7/20
363/363 [=====] - 1s 2ms/step - loss: 0.4940 - mae: 0.5059 - val_loss: 0.4497 - val_mae: 0.4821
Epoch 8/20
363/363 [=====] - 1s 2ms/step - loss: 0.4730 - mae: 0.4944 - val_loss: 0.4359 - val_mae: 0.4723
Epoch 9/20
363/363 [=====] - 1s 2ms/step - loss: 0.4563 - mae: 0.4856 - val_loss: 0.4267 - val_mae: 0.4658
Epoch 10/20
363/363 [=====] - 1s 2ms/step - loss: 0.4431 - mae: 0.4790 - val_loss: 0.4224 - val_mae: 0.4593
Epoch 11/20
363/363 [=====] - 1s 2ms/step - loss: 0.4333 - mae: 0.4729 - val_loss: 0.4265 - val_mae: 0.4595
Epoch 12/20
363/363 [=====] - 1s 3ms/step - loss: 0.4250 - mae: 0.4689 - val_loss: 0.4293 - val_mae: 0.4540
Epoch 13/20
363/363 [=====] - 1s 3ms/step - loss: 0.4183 - mae: 0.4651 - val_loss: 0.4381 - val_mae: 0.4557
Epoch 14/20
363/363 [=====] - 1s 2ms/step - loss: 0.4128 - mae: 0.4621 - val_loss: 0.4350 - val_mae: 0.4533
Epoch 15/20
363/363 [=====] - 1s 2ms/step - loss: 0.4081 - mae: 0.4597 - val_loss: 0.4364 - val_mae: 0.4499
Epoch 16/20
363/363 [=====] - 1s 2ms/step - loss: 0.4041 - mae: 0.4575 - val_loss: 0.4418 - val_mae: 0.4490
Epoch 17/20
363/363 [=====] - 1s 2ms/step - loss: 0.4005 - mae: 0.4552 - val_loss: 0.4612 - val_mae: 0.4468
Epoch 18/20
363/363 [=====] - 1s 3ms/step - loss: 0.3975 - mae: 0.4533 - val_loss: 0.4649 - val_mae: 0.4445
Epoch 19/20
363/363 [=====] - 1s 3ms/step - loss: 0.3941 - mae: 0.4508 - val_loss: 0.4492 - val_mae: 0.4422
Epoch 20/20
363/363 [=====] - 1s 3ms/step - loss: 0.3920 - mae: 0.4494 - val_loss: 0.4909 - val_mae: 0.4454
```

In [66]:

```
import pandas as pd

pd.DataFrame(model_history.history).plot(figsize = (8,5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [68]:

```
# Ahora vamos a ver cómo evaluar el modelo y cómo predecir clases usando el modelo

print('mae_test: ', model.evaluate(x_test, y_test))

##### VIDEO 41 MIN 14:55
```

```
162/162 [=====] - 0s 2ms/step - loss: 0.3892 - ma
e: 0.4483
mae_test: [0.3891874849796295, 0.4483468532562256]
```

In []:

Recomendaciones Globales para un código cualquiera de ANN

Dónde usar cada tipo de función de activación, cualquiera de ellas nos sirve para clasificación, pero para regresión deberemos usar ReLU, y acerca de en qué neurona colocarlas, Step/Escalón por lo general como Output, Sigmoid/TanH generalmente como Hidden/Output layer y ReLu para las Hidden layers

Como una serie de recomendaciones veremos siempre tendremos una única Input Layer, Hidden Layers depende del problema concreto pero típicamente entre 1 y 5, y todas esas Hidden Layers con función de activación ReLU para introducir la no linealidad. Para diferentes problemas como para un problema binario tendremos 1 Output Neuron, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy, para un problema de clasificación multiclase tendremos tantas Neuronas Output como clases tengamos, función de activación Softmax y una función de pérdida de tipo Cross-Entropy, y finalmente para un problema binario multicaracterística tendremos tantas Neuronas Output como características, con una función de activación de tipo logística y una función de pérdida de tipo Cross-Entropy.

Tenemos tantas Input Neuron como características haya. Tenemos además de 1-5 Hidden Layers, cada una de ellas con 10-100 Neuronas y ellas con función de activación ReLU. Finalmente tendremos tantas Output Neurons como características predichas, sin función de activación y todas ellas con función de pérdida ó error tipo MSE (Mean Squared Error).

Otras serie de recomendaciones surgen en compilar el modelo:

```
model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'sgd', metrics = ['accuracy'])
```

loss = 'sparse_categorical_crossentropy'; porque tenemos el data en 10 categorías diferentes, si tuviésemos probabilidades para cada característica entonces tendríamos que usar 'categorical_crossentropy', si fuesen muchas y binarias sería 'binary_crossentropy'

optimizer = 'sgd'; SGD Stochastic Gradient Descent, que le estamos contando a Keras que use un algoritmo de backpropagation

metrics = ['accuracy']; por estar usando clasificación, estaremos usando 'accuracy', si quisiésemos regresión, deberíamos de usar 'Mean Squared Error'

In []:

In [7]:

```
### Curso Udemy: Machine Learning & AI (Including Hands-on 3 Projects)
```

```
#!pip install tensorflow
```

```
# Si keras es para el backend, django es el entire planet o la chicha de la predicción  
#!pip install django
```

```
# Creamos una página web powered by Django
```

```
!django-admin.exe startproject myWebApp
```

Si buscamos en internet localhost es la pagina de Home Page de Jupyter Notebook y podremos ver hay uno llamado myWebApp dentro de él podemos ver dos subcarpetas llamadas myWebApp y manage.py

También se puede encontrar en las carpetas del ordenador C:\Users\PORTATIL\myWebApp

Lo siguiente a realizar va a ser crear una App dentro de esta WebApplication

In [11]:

```
#!ls -l  
!python.exe . C:\Users\PORTATIL\myWebApp\manage.py startapp imgUpload
```

```
C:\Users\PORTATIL\anaconda3\python.exe: can't find '__main__' module in  
'.'
```

In []:

```
from django.conf.urls import url  
from django.contrib import admin
```

In [26]:

```
##### Backend Test (Backend creation using pre-trained keras)

# Se puede elegir cualquier modelo ya creado en ese enlace a continuación y entonces ya
# podremos buscar
# https://keras.io/api/applications/

from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread

# El modelo de predicción ya implementado
model = ResNet50(weights = 'imagenet')

# Nos descargamos una foto random del animal que queramos en google para ver cómo iría
# funcionando esto
image_path = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\images\ducky.jpg'
#image_path = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\images\wolf.jpeg'

img = imread(image_path) # Una serie de números respectivos a cada pixel
plt.imshow(img)

# Ahora para un caso más real en el que tuviésemos una base de datos con muchos animales
# importamos los datos y
# se va a realizar la predicción

# Preparamos los datos
img = image.load_img(image_path, target_size = (224, 224))
X = image.img_to_array(img)
x= np.expand_dims(X, axis = 0)
x = preprocess_input(x)

# El modelo realiza la predicción
preds = model.predict(x)
print('Predicted:', decode_predictions(preds, top = 3)[0])
```

Predicted: [('n02051845', 'pelican', 0.14397138), ('n02494079', 'squirrel_monkey', 0.09544161), ('n01855672', 'goose', 0.05647832)]

In [15]:

```
##### Dog Breed Prediction

import os
import numpy as np
import pandas as pd
from keras.utils import np_utils
from sklearn.preprocessing import LabelEncoder

from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.preprocessing.image import img_to_array, load_img
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
from keras.callbacks import EarlyStopping

# Data
# https://www.kaggle.com/c/dog-breed-identification/data

# Ahora tenemos una carpeta llamada dog_breed_identification, con dos sub-carpetas llamadas train y test
# tenemos en cada uno de ellos un .csv y en ellos van a estar las 'id' y 'breed', esas dos columnas

folder_path = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\dog_breed_identification'

train_path = r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\dog_breed_identification\train'
files = os.listdir(train_path)

labels = pd.read_csv(os.path.join(folder_path, 'labels.csv'))
labels.head()

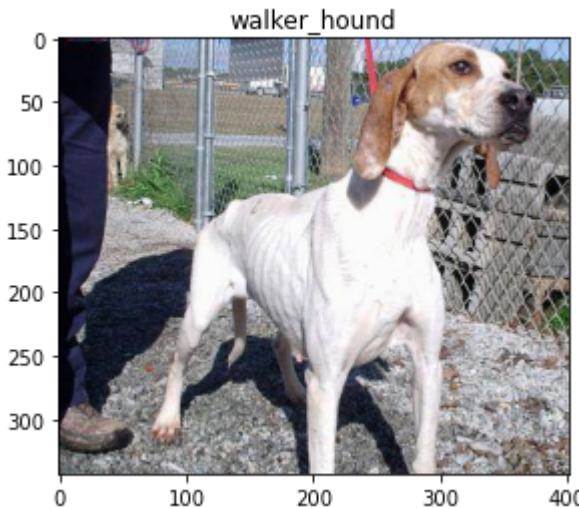
# Creamos el DataFrame

files_df = pd.DataFrame({'id':list(map(lambda x: x.replace('.jpg',''),files))})
files_df.head()

doggo_number = 11
img = plt.imread(os.path.join(train_path,files[doggo_number]))
plt.imshow(img)
plt.title(labels.iloc[doggo_number]['breed'])
plt.show()

# Tenemos que convertir a One-Hot-Encoder

classes_number = len(labels.breed.unique())
# print('classes_number:',classes_number)
# 120
```



In [16]:

```
LE = LabelEncoder()
breed = LE.fit_transform(labels.breed)
Y = np_utils.to_categorical(breed,num_classes=classes_number)
print('Y.shape: ', Y.shape)
# (10222, 120)

# Convertir las imágenes a array de numpy

input_dim = (224, 224)

X = np.zeros((Y.shape[0],*input_dim,3))

for i,img in enumerate(files):
    image = load_img(os.path.join(train_path,img), target_size = input_dim)
    image = img_to_array(image)
    image = image.reshape(1, *image.shape)
    image = preprocess_input(image)
    X[i] = image

print('X.shape: ', X.shape)
```

Y.shape: (10222, 120)

-

```
MemoryError                                                 Traceback (most recent call last)
t)
<ipython-input-16-256a3f85aa4a> in <module>
    9     input_dim = (224, 224)
   10
--> 11 X = np.zeros((Y.shape[0],*input_dim,3))
   12
   13 for i,img in enumerate(files):
```

MemoryError: Unable to allocate 11.5 GiB for an array with shape (10222, 224, 224, 3) and data type float64

In [13]:

```
# Crear "Callbacks"

earlystop = EarlyStopping(monitor = 'val_loss', min_delta = 0, patience = 2, verbose = 0, mode = 'auto')
# Le vamos a dar al modelo una cierta paciencia, es decir, que si deja de converger, de tener la ejecución

from keras.applications.vgg19 import VGG19
from keras.models import Model
from keras.layers import Dropout, Flatten, Dense, GlobalAveragePooling2D
from keras.optimizers import Adam

vgg_model = VGG19(weights = 'imagenet', include_top = False)

x = vgg_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.2)(x)
out = Dense(120, activation = 'softmax')(x)

model = Model(inputs=vgg_model.input, outputs = out)

for layer in vgg_model.layers:
    layer.trainable = False

opt = Adam()

model.compile(optimizer = opt, loss = 'categorical_crossentropy', metrics = [ 'accuracy' ])
model.summary()
```

Model: "functional_3"

Layer (type)	Output Shape	Param #
<hr/>		
input_5 (InputLayer)	[None, None, None, 3]	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1792
block1_conv2 (Conv2D)	(None, None, None, 64)	36928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73856
block2_conv2 (Conv2D)	(None, None, None, 128)	147584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295168
block3_conv2 (Conv2D)	(None, None, None, 256)	590080
block3_conv3 (Conv2D)	(None, None, None, 256)	590080
block3_conv4 (Conv2D)	(None, None, None, 256)	590080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1 (Conv2D)	(None, None, None, 512)	1180160
block4_conv2 (Conv2D)	(None, None, None, 512)	2359808
block4_conv3 (Conv2D)	(None, None, None, 512)	2359808
block4_conv4 (Conv2D)	(None, None, None, 512)	2359808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2359808
block5_conv2 (Conv2D)	(None, None, None, 512)	2359808
block5_conv3 (Conv2D)	(None, None, None, 512)	2359808
block5_conv4 (Conv2D)	(None, None, None, 512)	2359808
block5_pool (MaxPooling2D)	(None, None, None, 512)	0
global_average_pooling2d_4 ((None, 512)		0
dropout_3 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 120)	61560
<hr/>		
Total params:	20,085,944	
Trainable params:	61,560	
Non-trainable params:	20,024,384	

In [14]:

```
model.fit(X, Y, batch_size = 256, epochs = 20, validation_split = 0.2, verbose = 2, callbacks = [earlystop])
```

```

-
MemoryError                                         Traceback (most recent call last)
t)
<ipython-input-14-f3b79378ef40> in <module>
----> 1 model.fit(X, Y, batch_size = 256, epochs = 20, validation_split =
0.2, verbose = 2, callbacks = [earlystop])

~\anaconda3\lib\site-packages\tensorflow\python\keras\engine\training.py i
n _method_wrapper(self, *args, **kwargs)
    106     def _method_wrapper(self, *args, **kwargs):
    107         if not self._in_multi_worker_mode(): # pylint: disable=protec
ted-access
--> 108             return method(self, *args, **kwargs)
    109
    110     # Running inside `run_distribute_coordinator` already.

~\anaconda3\lib\site-packages\tensorflow\python\keras\engine\training.py i
n fit(self, x, y, batch_size, epochs, verbose, callbacks, validation_spli
t, validation_data, shuffle, class_weight, sample_weight, initial_epoch, s
teps_per_epoch, validation_steps, validation_batch_size, validation_freq,
max_queue_size, workers, use_multiprocessing)
    1047         training_utils.RespectCompiledTrainableState(self):
    1048             # Creates a `tf.data.Dataset` and handles batch and epoch it
eration.
-> 1049             data_handler = data_adapter.DataHandler(
    1050                 x=x,
    1051                 y=y,

~\anaconda3\lib\site-packages\tensorflow\python\keras\engine\data_adapter.
py in __init__(self, x, y, sample_weight, batch_size, steps_per_epoch, ini
tial_epoch, epochs, shuffle, class_weight, max_queue_size, workers, use_mu
ltiprocessing, model, steps_per_execution)
    1103
    1104     adapter_cls = select_data_adapter(x, y)
-> 1105     self._adapter = adapter_cls(
    1106         x,
    1107         y,

~\anaconda3\lib\site-packages\tensorflow\python\keras\engine\data_adapter.
py in __init__(self, x, y, sample_weights, sample_weight_modes, batch_siz
e, epochs, steps, shuffle, **kwargs)
    263             **kwargs):
    264     super(TensorLikeDataAdapter, self).__init__(x, y, **kwargs)
--> 265     x, y, sample_weights = _process_tensorlike((x, y, sample_weigh
ts))
    266     sample_weight_modes = broadcast_sample_weight_modes(
    267         sample_weights, sample_weight_modes)

~\anaconda3\lib\site-packages\tensorflow\python\keras\engine\data_adapter.
py in _process_tensorlike(inputs)
    1019     return x
    1020
-> 1021     inputs = nest.map_structure(_convert_numpy_and_scipy, inputs)
    1022     return nest.list_to_tuple(inputs)
    1023

~\anaconda3\lib\site-packages\tensorflow\util\nest.py in map_struct
ure(func, *structure, **kwargs)
    633
    634     return pack_sequence_as(

```

```

--> 635     structure[0], [func(*x) for x in entries],
636         expand_composites=expand_composites)
637

~\anaconda3\lib\site-packages\tensorflow\python\util\nest.py in <listcomp>
(0.)
633
634     return pack_sequence_as(
--> 635         structure[0], [func(*x) for x in entries],
636         expand_composites=expand_composites)
637

~\anaconda3\lib\site-packages\tensorflow\python\keras\engine\data_adapter.
py in _convert_numpy_and_scipy(x)
1014     if issubclass(x.dtype.type, np.floating):
1015         dtype = backend.floatx()
-> 1016     return ops.convert_to_tensor(x, dtype=dtype)
1017     elif scipy_sparse and scipy_sparse.issparse(x):
1018         return _scipy_sparse_to_sparse_tensor(x)

~\anaconda3\lib\site-packages\tensorflow\python\framework\ops.py in conver
t_to_tensor(value, dtype, name, as_ref, preferred_dtype, dtype_hint, ctx,
accepted_result_types)
1497
1498     if ret is None:
-> 1499         ret = conversion_func(value, dtype=dtype, name=name, as_ref=
as_ref)
1500
1501     if ret is NotImplemented:
1502         raise TypeError("Type %s not convertible" % type(value))

~\anaconda3\lib\site-packages\tensorflow\python\framework\tensor_conversio
n_registry.py in _default_conversion_function(**failed resolving argument
s**)
50 def _default_conversion_function(value, dtype, name, as_ref):
51     del as_ref # Unused.
---> 52     return constant_op.constant(value, dtype, name=name)
53
54

~\anaconda3\lib\site-packages\tensorflow\python\framework\constant_op.py i
n constant(value, dtype, shape, name)
261     ValueError: if called on a symbolic tensor.
262     """
--> 263     return _constant_impl(value, dtype, shape, name, verify_shape=False,
264                             allow_broadcast=True)
265

~\anaconda3\lib\site-packages\tensorflow\python\framework\constant_op.py i
n _constant_impl(value, dtype, shape, name, verify_shape, allow_broadcast)
273     with trace.Trace("tf.constant"):
274         return _constant_eager_impl(ctx, value, dtype, shape, veri
fy_shape)
--> 275     return _constant_eager_impl(ctx, value, dtype, shape, verify_
shape)
276
277     g = ops.get_default_graph()

~\anaconda3\lib\site-packages\tensorflow\python\framework\constant_op.py i
n _constant_eager_impl(ctx, value, dtype, shape, verify_shape)
298 def _constant_eager_impl(ctx, value, dtype, shape, verify_shape):

```

```
299     """Implementation of eager constant."""
--> 300     t = convert_to_eager_tensor(value, ctx, dtype)
301     if shape is None:
302         return t

~\anaconda3\lib\site-packages\tensorflow\python\framework\constant_op.py i
n convert_to_eager_tensor(value, ctx, dtype)
    96         dtype = dtypes.as_dtype(dtype).as_datatype_enum
    97     ctx.ensure_initialized()
---> 98     return ops.EagerTensor(value, ctx.device_name, dtype)
    99
100
```

MemoryError: Unable to allocate 4.59 GiB for an array with shape (8177, 224, 224, 3) and data type float32

In []:

In []:

In []:

In []:

APRENDIZAJE NO SUPERVISADO

Cuando el conjunto de datos no tiene etiquetas. Estos son los algoritmos de Machine Learning que usamos cuando los datos "no tienen fundamento". Son algoritmos que consiguen interpretar patrones en conjuntos de datos sin etiquetas, sin referencia y construyen de esos unos datos ordenados o etiquetados. No se pueden aplicar en un problema de regresión o clasificación porque no se tiene idea de cuáles o cómo son los datos de salida, es por ello que no se entrenan esos datos como lo haría ese algoritmo de clasificación o regresión.

El aprendizaje no supervisado sirve para ver la estructura subyacente de una serie de datos, nos permiten realizar tareas de preprocesamiento más complejas en comparación con el supervisado. Como desventaja podemos ver es más impredecible que los algoritmos de aprendizaje supervisado. Los algoritmos de aprendizaje no supervisado se aprovechan para estudiar los conjuntos de datos no estructurados, según sus limitaciones y patrones distintos en el conjunto de datos.

Se dice no supervisado porque no está guiado como los algoritmos de aprendizaje supervisado. No se enseña sino que se aprende a partir de unos datos. Se busca encontrar patrones escondidos en los datos, pero muchas veces son aproximaciones deficientes de lo que el aprendizaje supervisado puede encontrar, y ya que no se sabe cómo se está haciendo, no se puede medir la precisión, por lo que el aprendizaje supervisado es más aplicable a los problemas del mundo real.

El aprendizaje no supervisado se utiliza principalmente ya que encuentra todo tipo de patrones en los datos, ayuda a encontrar características que pueden ser útiles para la categorización y porque en el mundo real se encuentran los datos desetiquetados.

Los algoritmos de aprendizaje no supervisado manejan los datos sin hacer entrenamiento, en cierto modo se deja a su suerte. Funcionan con datos no etiquetados, funcionan bajo condiciones en las que los resultados son desconocidos por lo que es necesario definirlos en el proceso. Los algoritmos del aprendizaje no supervisado están acostumbrados a explorar la información y detectar patrones distintos, extraer ideas valiosas y aplicarlas en su proceso de toma de decisiones (para aumentar su eficiencia).

Tipos de Aprendizaje No Supervisado:

-Agrupamiento/Clusters: Encontrar una estructura o patrón en datos no etiquetados, van a procesar los datos y encontrar grupos o clusters naturales si es que existen en los datos (se puede modificar cuántos grupos deben modificar sus algoritmos, es decir, permite modificar la granularidad). Existen diferentes tipos de Clusters que se pueden utilizar:

-Clusters Exclusivos: Los datos se agrupan de forma que los datos pueden pertenecer únicamente a un Cluster.

-Cluster Agrupativo: Cada dato es un Cluster, las uniones iterativas entre los dos Clusters más cercanos reduce el número de Clusteres. (Agrupaciones jerárquicas)

-Cluster Solapamiento: Conjuntos difusos para agrupar datos, cada dato puede pertenecer a dos o más Clusters con distintos grados de unión.

-Cluster Probabilístico: Utiliza técnicas de distribución de probabilidad para la formación de Clusters.

-Asociación: Reglas de Asociación te permiten obtener relaciones entre objetos de datos dentro de grandes bases de datos, trata de obtener relaciones interesantes

Método de Agrupamiento/Clustering

Agrupar datos en Clusters basándose en los patrones entre datos y poder luego tomar decisiones. El Clustering es una de las técnicas de aprendizaje no supervisado más utilizada, útil para dar sentido a los datos no etiquetados y agruparlos. Puede descifrar estructuras y patrones no visibles a ojo humano.

Un Cluster es una colección de datos similares entre sí por características comunes y son diferentes de los objetos de los otros Clusters. La agrupación es una técnica en la que hay información previa o clases predefinidas que define cómo se deben agrupar o etiquetar los datos en clases separadas, también podría verse como un proceso de análisis exploratorio para buscar patrones ocultos de interés o estructuras en los datos, todo esto además de por la información nos puede servir para hacer preprocesamiento.

Se puede ver de forma gráfica, para intentar agruparlos, pero por lo general un Cluster debe cumplir las siguientes propiedades:

- Los puntos de un mismo Cluster deben ser lo más parecidos posibles.
- Los puntos de distintos Cluster deben ser lo más diferentes posibles de otros Cluster y así tener Cluster más restrictivos.

Clasificación (supervisado) vs Clustering (no supervisado):

- En el supervisado se aprende un método para intentar predecir la clase instancia a partir de una instancia pre-etiquetada o clasificada
- En el no supervisado se trata de encontrar la agrupación natural de instancias de un dato dado sin etiquetar.

Un Cluster de alta calidad se puede conseguir mejorando un cluster actual, minimizando la distancia intra-Cluster, y maximizando la distancia frente a otros, inter-Cluster. Estamos buscando agrupaciones, pero mejor que sean agrupaciones buenas y significativas.

Métricas de Evaluación:

-Inercia: Nos da cuenta de cómo de lejos están los puntos dentro de un Cluster a partir de la distancia al centroide. Una vez calculada la inercia de todos los Clusters, se suman y eso es la distancia inercia final, es la distancia intra-Cluster. Queremos esta inercia sea lo menor posible.

-Índice Dunn: Nos da cuenta de que los grupos sean lo más distintos posibles, es la distancia entre los centros de dos Clusters distintos, a mayor sea esta distancia inter-Cluster, mejor.

Índice Dunn = $\frac{\text{mínimo}(\text{distancia inter-Cluster})}{\text{máximo}(\text{distancia intra-Cluster})}$
Queremos maximizarlo, por ello maximizar el numerador que es el mínimo de la distancia inter-Cluster, y queremos maximizar las distancias entre diferentes Clusters y consideramos la menor de ellas para ponernos en el peor de los escenarios posibles. Lo mismo ocurre con el denominador que es el máximo de la distancia intra-Cluster, y queremos minimizar las distancias entre un mismo Cluster y consideramos la mayor de ellas para ponernos en el peor de los escenarios posibles de nuevo.

Tiene aplicaciones en la actualidad en la selección de clientes, agrupación de documentos, segmentación de imágenes y motores de recomendación.

Algoritmos más comunes en Clustering

Agrupamiento K-means

El algoritmo más fácil de entender e implementar. Inicialmente elegimos un número de clusters e inicializamos aleatoriamente dónde colocar el centroide. Antes de empezar es bueno echar un vistazo rápido a los datos y tratar de ver agrupaciones distintas. Los puntos centrales son vectores de la misma longitud que cada vector de puntos de datos.

Cada punto se clasifica calculando la distancia a los centros de los Clusters, y luego asignándolo al Cluster más cercano. Basándonos en estos puntos ya clasificados calculamos la media de todos los vectores pertenecientes a cada Cluster y luego recalculamos la posición del Cluster. Se itera el proceso hasta un cierto número de iteraciones o hasta que los centroides no cambien mucho entre iteraciones. También se puede hacer las iteraciones repitiendo el proceso de elección de centroides aleatorios, es decir, hacerlo varias veces e iterar en cada una y finalmente quedarnos con el que tenga mejores resultados.

Es un algoritmo muy rápido por ser cálculos de distancias y medias. Por otro lado algunas desventajas es tener que seleccionar cuántos Clusters hay y no siempre es trivial, y nos gustaría el algoritmo lo hiciese por nosotros, también por ser una elección inicial aleatoria de los centros de esos Clusters vemos de una ejecución a la siguiente vamos a tener posibles resultados diferentes y carecer de consistencia.

Tiene una variante K-medians, que calcula la mediana, por lo que es menos sensible a outliers pero es mucho más lento al clasificar en cada iteración tras calcular la mediana.

Agrupamiento Mean Shift

El algoritmo basado en ventanas deslizantes que intenta encontrar áreas densas de puntos de datos. Se basa en el centroide, busca ese punto centroide y lo que hace es ir actualizando los puntos para que los puntos centrales sean la media de los puntos de la ventana deslizante. Estas ventanas son filtradas en una época de post-procesamiento para evitar los duplicamientos y así formando esos agrupamientos. Se elige una geometría para la ventana (por ejemplo circular) y un tamaño (en este caso un radio) y se va haciendo un desplazamiento en cada iteración de menor densidad hasta mayor densidad hasta la convergencia. En cada iteración se va desplazando esa ventana hacia regiones de mayor densidad desplazando su punto central hacia la media de los puntos dentro de la ventana, la densidad es proporcional al número de puntos dentro de ella, se va a ir aumentando esa densidad, hasta que llegue un punto en el que no se pueda agrupar una mayor densidad.

Todo ese proceso se itera con muchas ventanas, cuando se superponen las ventanas se conserva la que mayor densidad agrupe. A continuación en el conjunto de datos los puntos se agrupan en la ventana en la que residen.

A diferencia de K-means no necesita saber el número de Clusters ya que ese desplazamiento medio lo descubre automáticamente. El hecho de que los Clustering converjan hacia el punto de mayor densidad es lo mejor, ya que es muy intuitivo y encaja bien en el modelo. El problema es la selección del tamaño de la ventana puede ser no trivial.

Agrupamiento DBSCAN

Este algoritmo basado en la densidad similar a Mean Shift pero con un par de ventajas notables, las siglas significan Agrupamiento Espacial Basado en la Densidad de Aplicaciones con Ruido. DBSCAN comienza con un conjunto de datos de inicio no visitados aún arbitrario, el vecindario de este punto se extrae usando una distancia epsilon. Si hay un número de puntos elevados en este vecindario, en ese rango epsilon, el

proceso de agrupación comienza y el primer punto de datos se convierte en el siguiente al que avanzamos, de lo contrario el punto será etiquetado como ruido (luego sí que podrá ser visitado de nuevo y pasar a ser parte del Cluster); en ambos casos el punto cuenta como visitado.

Para meter ese primer punto en un nuevo Cluster, los puntos de su vecindario epsilon pasar a ser parte del mismo Cluster también, se itera para todos los nuevos puntos que acaban de ser agregados al Cluster y se hace esto hasta que todos los puntos nuevos del Cluster hayan sido visitados y etiquetados como pertenecientes a ese Cluster.

Lo siguiente es ir a un nuevo punto no visitado y que no pertenezca a ese Cluster y entonces buscaremos de la misma forma encontrando otros Clusters o ruido.

Tenemos ahora ciertas ventajas, como que tampoco requiere del conocimiento del número de Clusters, etiquetará los valores atípicos como ruido, a diferencia del Agrupamiento Mean Shift, que los coloca en algún Cluster incluso si es muy diferente. Además se pueden encontrar Clusters de diferente tamaño.

Sin embargo tiene desventajas como su incorrecto funcionamiento en ocasiones cuando el Cluster es de densidad variable, esto se debe a la configuración del umbral de distancia epsilon para identificar los puntos del vecindario variará de un Cluster a otro cuando la densidad variable. También tenemos problemas a grandes dimensiones cuando la distancia epsilon se vuelve difícil de estimar.

Agrupamiento utilizando modelos de media gaussiana (Gaussian Mean Model, GMM)

Uno de los mayores inconvenientes de K-means es su ingenuo uso del valor medio para el centro del Cluster, también por su uso del valor medio podemos ver falla cuando el Cluster no es circular. Los GMM nos dan más flexibilidad, asumimos que los datos están distribuidos de forma normal o gaussiana, lo cual es una aproximación menos restrictiva que decir que es circular usando la media. Tomando un ejemplo en dos dimensiones tendremos media y desviación estándar, por lo que los Clusters podrán tener formas elípticas, ya que tendremos desviación estándar tanto en dirección X como en dirección Y. Para encontrar los valores de los parámetros de desviación estándar y valor promedio usaremos un algoritmo de Maximización de Expectativas (EM).

Lo primero es definir el número de Clusters, tal y como hacíamos con K-means e inicializamos aleatoriamente los parámetros de esa distribución gaussiana para cada Cluster, sino se puede intentar hacer a ojo, pero podemos ver da igual ya que aunque al principio no se ajuste, se optimizan rápidamente. Dadas estas distribuciones gaussianas para cada Cluster, calcular la probabilidad de que un dato pertenezca a un Cluster particular, cuanto más cerca esté de un centro de los Cluster más probable será que pertenezca a este grupo, ya que por ser distribución gaussiana asumimos los datos estarán muy cercanos al centro del Cluster. Una vez hechas estas probabilidades calculamos un nuevo valor de esos parámetros de tal forma que maximicen las probabilidades de pertenecer a sus Clusters, se calculan estos nuevos parámetros usando una suma ponderada de las posiciones de los puntos de datos, donde las ponderaciones son los puntos de datos que pertenecen a ese dato en particular.

Todos estos pasos se repiten hasta la convergencia y hasta que no se cambie mucho de una iteración a la siguiente.

Tienen una serie de ventajas, son mucho más flexibles en términos de la forma de los Clusters debido al parámetro de la desviación estándar, de forma que se pueden adaptar a elipsoides en vez de solo a círculos como K-means. También, como los modelos de mezcla gaussiana utilizan probabilidades pueden tener conglomerados múltiples por puntos de datos, si un punto de datos está en medio de dos Clusters, no le asignamos a un Cluster u otro sino que le indicamos una probabilidad de pertenencia a cada Cluster.

Agrupamiento Jerárquico Ascendente (también está el descendente que funciona a la inversa)

Son un tipo de algoritmo que trabajan considerando inicialmente todos los puntos de datos como Clusters y luego los van agrupando hasta llegar a un único Cluster. La raíz de un árbol es el Cluster único y cada hoja es un único dato que inicialmente si tenemos N datos, tendremos N hojas, siendo cada una de ellas un Cluster.

Tenemos que seleccionar una métrica de distancia, en cada iteración buscaremos agrupar dos Clusters en uno considerando aquellos con la vinculación media más pequeña, es decir los más cercanos entre sí ya que por lo tanto serán los más parecidos entre sí y deben ser combinados. De esta forma se itera el proceso y podremos ver se llegará a un único Cluster que agrupe todos los datos al final y así podremos determinar cuántos Clusters querremos al final, es decir, cuándo podemos dejar de agrupar o dejar de construir el árbol.

Como ventaja, la agrupación jerárquica no requiere de especificarle el número de Clusters e incluso podemos seleccionar qué número de Clusters se ve mejor a medida que se va construyendo el árbol, además, este algoritmo no es sensible a la métrica de distancia mientras que con otros algoritmos la elección de esta métrica es crítica, y además, un caso particular bueno en el que este algoritmo es realmente útil es cuando los datos subyacentes tienen una estructura jerárquica y es lo que se desea encontrar. Sin embargo tiene sus desventajas como en términos de eficiencia, a diferencia de la complejidad lineal y gaussiana de otros algoritmos.

Métodos de Selección del Número de Clusters

Vamos a buscar los métodos para encontrar ese número óptimo de Clusters para los algoritmos anteriores que lo requieran.

Método del Codo

Probablemente el método más conocido, se calcula y se grafica la suma cuadrado en cada número de Clusters, y se busca un cambio de pendiente de tal forma que cambie de mucha pendiente a poca, un codo. Así se puede determinar el número óptimo de codos, es un método potencialmente inexacto pero útil ya que nos da cuenta de cómo el aumento de los números de Clusters contribuye de forma que también separa a los Clusters entre sí y no de forma marginal, es decir un método de solución ingenua basada en la varianza intracluster.

La estadística de la brecha es un método más sofisticado para tratar con datos con una distribución no obvia, se calcula la suma de los errores cuadrados dentro del Cluster para diferentes valores de K y se elige la K para la cual la suma empieza a decrecer, eso es visible de nuevo como un codo. Este error cuadrado es la suma de las distancias punto-centro del Cluster y veremos se puede usar cualquier métrica de distancia como la Euclídea o Manhattan. No obstante no siempre tendremos funcionará este método, ya que no siempre estarán los datos más o menos agrupados, por lo que no podremos usar este método.

Así pues podemos agrupar estos datos con el algoritmo K-means para diferentes valores de K y por ejemplo hacer esto para K de 1 a 10 calculando para todo el rango los valores del error cuadrado

Método de la silueta

Es un método que puede utilizarse para calcular la distancia de separación entre los grupos resultantes, el gráfico de la silueta muestra cuánto de cerca está cada punto en un Cluster, en los clústeres vecinos y por ello proporciona una forma de evaluar tanto los parámetros como el número de Clusters de forma visual. A mayor sea la distancia veremos es los clusters son más cercanos al propio centro del Cluster y a menor estarán los puntos más cerca del Cluster vecino. Este método es mejor ya que hace que la decisión sobre el número óptimo de Clusters sea más costosa y clara.

Es una métrica compleja y se calcula el coeficiente en cada caso por lo que la decisión sobre la métrica óptima a elegir es en función del problema concreto. Este coeficiente de silueta se calcula usando la distancia media intracluster "a" y la distancia más cercana a un Cluster que no pertenece el dato es "b" para cada muestra, entonces el, coeficiente es $b-a/\max(a,b)$ Así tendremos que calcular el coeficiente promedio sobre todos los grupos y hacer el promedio.

Es similar al Método del Codo y se puede calcular de la siguiente forma, calcular el algoritmo de Clustering K-Means para diferentes valores de K y hacerlo con valores de K de 2 a 11 ya que K=1 no se puede hacer por haber más de 1 Cluster sí o sí. Para cada K calcular la silueta promedio de sus observaciones, trazar la curva con la silueta promedio de acuerdo al número de Clusters K, y finalmente la ubicación del máximo se considera como el número ideal de Clusters.

Método de Estadística de la Brecha

Este método compara el total de variación intracluster para diferentes valores de K con sus valores esperados bajo una distribución de referencia no nula de los datos. La estimación de los Clusters óptimos será un valor que maximice la estadística de la brecha, es decir, que produzca la estadística de la brecha más grande. Todo esto significa que el Cluster estará muy lejos del conjunto de puntos de la distribución uniforme y aleatoria de los puntos.

El gráfico de estadísticas de brecha muestra las estadísticas por número de Clusters con errores estándar dibujados con segmentos verticales y el valor óptimo del número K de Clusters. (Gap statistics, k en el eje vertical frente al número de Clusters en el eje horizontal).

In [53]:

```
##### Métodos de Selección del Número de Clusters

##### Método del Codo

# Importar las librerías
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt

# Definir los datos a utilizar
x1 = np.array([3, 1, 1, 2, 1, 6, 6, 6, 5, 6, 7, 8, 9, 8, 9, 9, 8])
x2 = np.array([5, 4, 5, 6, 5, 8, 6, 7, 6, 7, 1, 2, 1, 2, 3, 2, 3])
plt.plot()
plt.xlim([0, 10])
plt.ylim([0, 10])
plt.title('Conjunto de datos')
plt.scatter(x1, x2)
plt.show()

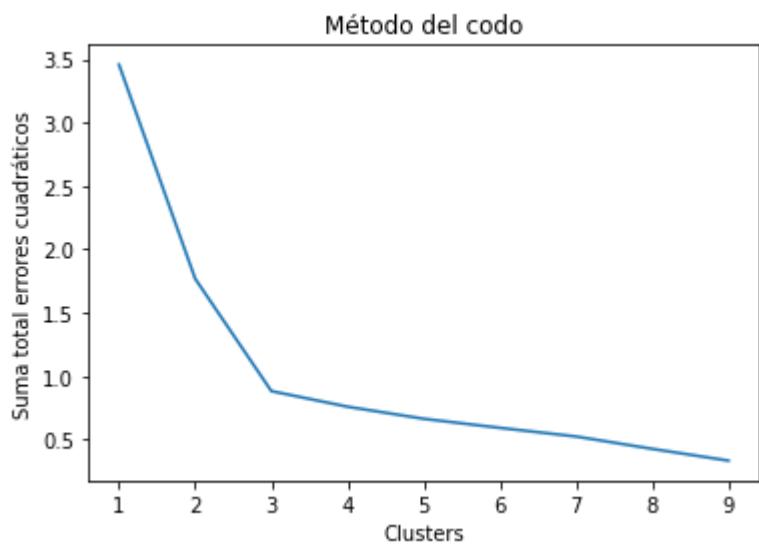
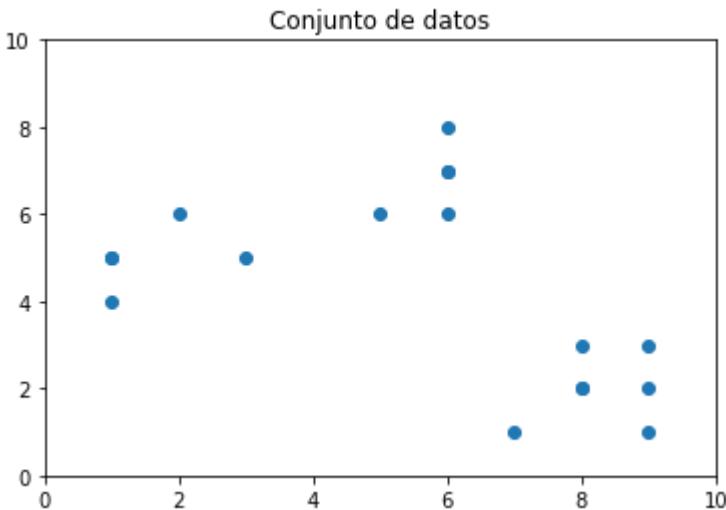
# Se pueden ver esos 3 clusters pero vamos a dejar que nos lo diga el criterio del codo

# Vamos a unir estas dos variables que tenemos
X = np.array(list(zip(x1,x2))).reshape(len(x1),2)

# Ahora el criterio que vamos a seguir es que tenemos que calcular el algoritmo K-Means
# para varios valores

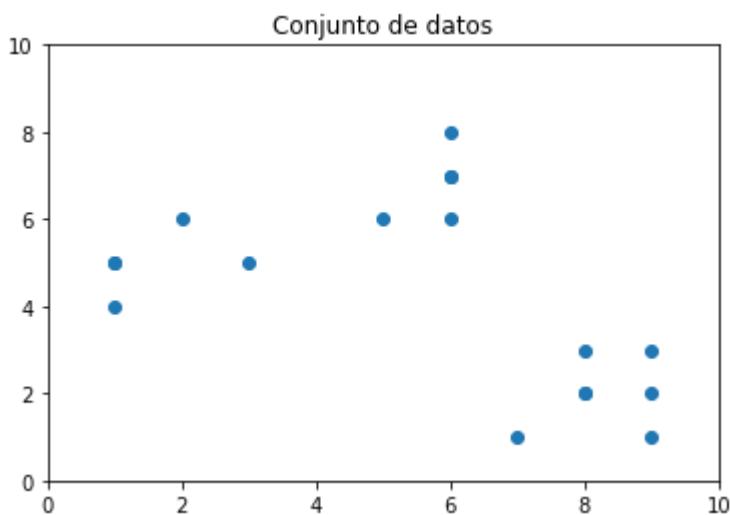
# Determinamos el número de Clusters
clusters = []
K = range(1,10)
for k in K:
    Modelokmean = KMeans(n_clusters = k)
    Modelokmean.fit(X)
    clusters.append(sum(np.min(cdist(X, Modelokmean.cluster_centers_, 'euclidean'),axis=1))/X.shape[0])
# plt.plot(K, clusters, 'bx=')
plt.plot(K, clusters)
plt.xlabel('Clusters')
plt.ylabel('Suma total errores cuadráticos')
plt.title('Método del codo')
plt.show()

# Y podemos ver el codo aparece aproximadamente a los 3 clusters
```



In [54]:

```
##### Métodos de Selección del Número de Clusters  
##### Método de la Silueta  
  
# Importar Las Librerías  
from sklearn.metrics import silhouette_score  
from sklearn.cluster import KMeans  
from scipy.spatial.distance import cdist  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Definir Los datos a utilizar  
x1 = np.array([3, 1, 1, 2, 1, 6, 6, 6, 5, 6, 7, 8, 9, 8, 9, 9, 8])  
x2 = np.array([5, 4, 5, 6, 5, 8, 6, 7, 6, 7, 1, 2, 1, 2, 3, 2, 3])  
plt.plot()  
plt.xlim([0, 10])  
plt.ylim([0, 10])  
plt.title('Conjunto de datos')  
plt.scatter(x1, x2)  
plt.show()  
  
# Se pueden ver esos 3 clusters pero vamos a dejar que nos lo diga el criterio del codo  
  
# Vamos a unir estas dos variables que tenemos  
X = np.array(list(zip(x1,x2))).reshape(len(x1),2)  
  
# Ahora el criterio que vamos a seguir es que tenemos que calcular el algoritmo K-Means  
para varios valores  
  
silhouette = []  
  
K = range(2,11)  
for k in K:  
    kmeans = KMeans(n_clusters = k)  
    kmeans.fit(X)  
    labels = kmeans.labels_  
    silhouette.append(silhouette_score(X, labels, metric = 'euclidean'))  
plt.plot(K, silhouette)  
plt.xlabel('Clusters')  
plt.ylabel('Puntuaje de la Silueta')  
plt.title('Método de la Silueta')  
plt.show()
```



Una vez encontrado el número óptimo de Clusters tendremos que realizar la partición en Clusters

Agrupamiento K-means

El algoritmo más fácil de entender e implementar. Inicialmente elegimos un número de clusters e inicializamos aleatoriamente dónde colocar el centroide. Antes de empezar es bueno echar un vistazo rápido a los datos y tratar de ver agrupaciones distintas. Los puntos centrales son vectores de la misma longitud que cada vector de puntos de datos.

Es un Clustering basado en prototipos o centroides, cada punto se clasifica calculando la distancia a los centros de los Clusters (centroides), y luego asignándolo al Cluster más cercano. Basándonos en estos puntos ya clasificados calculamos la media de todos los vectores pertenecientes a cada Cluster y luego recalculamos la posición del Cluster. Se itera el proceso hasta un cierto número de iteraciones o hasta que los centroides no cambien mucho entre iteraciones, o los puntos permanecen perteneciendo al mismo Cluster (estas últimas implican que el algoritmo ya no está aprendiendo). También se puede hacer las iteraciones repitiendo el proceso de elección de centroides aleatorios, es decir, hacerlo varias veces e iterar en cada una y finalmente quedarnos con el que tenga mejores resultados. Estaremos también teniendo que considerar algoritmos con diferentes posibles números de Clusters, por lo que tendremos que ir cambiándolos. Por lo general suele ser una buena elección empezar con $K=2,3$ e ir actualizando. Al hacer la iteración tenemos que elegir un nuevo punto del Cluster como centroide, ir corrigiendo y buscando esos centroides reales, ya que los centroides iniciales fueron elegidos aleatoriamente y sabemos el nuevo centroide podría o no aparecer en el conjunto de datos, no suele serlo, ya que se elige el promedio de esos puntos de los clusters.

Es un algoritmo muy rápido por ser cálculos de distancias y medias. Por otro lado algunas desventajas es tener que seleccionar cuántos Clusters hay y no siempre es trivial, y nos gustaría el algoritmo lo hiciese por nosotros, también por ser una elección inicial aleatoria de los centros de esos Clusters vemos de una ejecución a la siguiente vamos a tener posibles resultados diferentes y carecer de consistencia, también veremos Clusters de uniformes tamaños aunque los datos pueden comportarse de forma diferente, es además muy sensible a valores atípicos y ruido. Otra desventaja es que por la media se asume una distancia circular o esférica alrededor de cada Cluster, lo que no será siempre así.

Como en la vida real tenemos datos ruidosos, complejos y desordenados ya veremos esta desventaja de no ser Clusters circulares ya se va a manifestar, por lo que tenemos que ver si hay alguna de las desventajas mencionadas apareciendo en nuestro problema particular y ver si están presentes, y además de identificarlos, saber qué hacer si los hay. Una de las soluciones posibles es la conversión de las coordenadas a polares y así resolver esa limitación esférica. Otra de las soluciones es considerar algoritmos de agrupación en Cluster si hay demasiadas limitaciones, donde veríamos como alternativas el uso de algoritmos jerárquicos o basados en la densidad, lidiando con las limitaciones de la agrupación K-means pero esos algoritmos también tienen sus propias limitaciones.

Tiene variantes, K-Modo K-Medians, que calculan el modo y la mediana, por lo que es menos sensible a outliers pero es mucho más lento al clasificar en cada iteración tras calcular la mediana.

Los resultados de este algoritmo o método son los centroides de los K Clusters que pueden ser utilizados para etiquetar datos, nos da también los datos de formación y cada punto de datos se adjudica a un único Cluster. En vez de encontrar Clusters antes de examinar datos, esta agrupación permite encontrar y organizar los grupos que se han formado orgánicamente. Cada centroide de un Cluster es un conjunto de valores de características que definen los grupos resultantes. Si se estudia las ponderaciones de las características del centroide utilizarse para interpretar cualitativamente qué tipo de grupo representa cada conglomerado.

Una de las métricas de evaluación de K, el número de Clusters predeterminado, para chequear cómo funciona nuestro algoritmo para un cierto valor del parámetro K tenemos que buscar la distancia media entre los datos y sus respectivos centroides. Dado que el aumento del número de Clusters reducirá dichas distancias, también implica que ese K es mayor y es el K de valores de datos que le damos al programa, por eso esta métrica no puede usarse como único objetivo, en su lugar se calcula la distancia al centroide en función de K y se puede utilizar el Método del Codo, donde la tasa de la disminución se desplaza bruscamente para determinar de forma aproximada el número de Clusters. Existen otras formas de validar K, como ya sean Validación Cruzada, los criterios de información y el método de salto teórico de información y el Método de la Silueta.

También tenemos que tener en cuenta la métrica de evaluación para la calidad de un Cluster y ver que estos sean buenos y significativos, es decir, cuando los datos de un Cluster están muy cerca entre sí y muy separados del resto de Clusters. Ya vimos varios criterios como la Inercia (de 0 a superiores) o el Índice de Dunn para evaluar la calidad de un Cluster, y también podríamos verlo cuantificado por medio del Punto de Silueta, conseguido en el Método de la Silueta que nos da cuenta de la distancia entre los puntos de datos de un Cluster y los puntos de datos de otro Cluster (de -1 a 1, el óptimo).

Algunos de sus usos son segmentación del comportamiento (por historial de compras, actividades en la aplicación o web, definir personas en base a sus intereses, crear perfiles basados en supervisión de actividades), categorización de inventario (por actividad de ventas, por métricas de fabricación), clasificación de medidas de los sensores (detectar actividad en los sensores de movimiento, agrupar imágenes, audio separado identificar grupos en la vigilancia de la salud) y detección de robots o anomalías (separar grupos

Agrupamiento K-Mean

```
from sklearn.cluster import KMeans
```

KMeans: va a tener varias funciones

definimos el modelo

```
kmeans = KMeans()
```

Tenemos que ver ahora los parámetros importantes en la función KMeans()

n_clusters: (default=8) el número de parámetros de los Clusters, y por ende, número de centroides.
Conviene ajustar el valor de este parámetro ya que es una de las desventajas de este algoritmo y tendremos que estimar el número de Clusters con cualquiera de los Métodos de Selección del Número de Clusters estudiados.

init: (default: 'k-means++') el tipo de inicialización de estos centroides, tenemos varios tipos 'k-means++', 'random' o 'ndarray'. El primero de ellos 'k-means++' es elegir los centroides de forma inteligente para una rápida convergencia, 'random' elige al azar esos K observaciones o filas al azar a partir de los datos de los centroides iniciales y en caso de elegir 'ndarray' tendremos que indicarle al programa la forma de 'n_clusters' y 'n_characteristics', y así esos serán los centroides iniciales. Por lo general se suele elegir 'k-means++', o dejarlo por defecto, y luego mirar a ver si ha habido cambios chequeando con 'random'.

n_init: (default: 10) va a ser el número de veces que se va a inicializar nuestro problema con diferentes centroides, de todas formas el resultado será la mejor salida de n_init consecutivos en términos de inercia.

max_iter: (default: 300) el número de iteraciones máximas que va a realizar nuestro problema, el

criterio de parada para buscar esa convergencia de los centroides. Se puede jugar con este parámetro para buscar convergencias.

Ahora podemos ver qué parámetros podemos cambiar en estas funciones para tener un mayor control sobre el algoritmo K-Means, y poder tener mejores resultados con este modelo

entrenamos el modelo

kmeans.fit(X)

X serán los datos, no tendremos una variable dependiente 'y' como en el aprendizaje supervisado, ya que no buscamos una solución sino la comprensión de unos valores sin etiquetas.

realizar una predicción

kmeans.predict(X)

Determinar los Clusters más cercanos al que pertenece la muestra X, en este caso podemos seleccionar unas muestras o datos nuevos para ver a cuál de esos Clusters pertenece ya que tenemos este modelo K-Means ya entrenado. Una vez hecho esto el resultado que vamos a obtener es número del Cluster asociado a este dato en el que queramos hacer una predicción.

Una vez explicados esos parámetros podemos ver los dos atributos importantes que podemos obtener a partir de este modelo, sobretodo si es que queremos representar los resultados de forma gráfica o en tabla.

cluster_centers: nos devuelve las coordenadas de los centroides, obviamente el número de coordenadas deberá de coincidir con el número de Clusters o centroides, por lo que habrá que considerarlo a la hora de usarlo porque sino dará error.

labels_: aquí se obtienen los labels o etiquetas de los Clusters calculados.

inertia_: devuelve la suma al cuadrado de la distancias muestras a sus respectivos Clusters más cercanos.

In [55]:

```
##### Método K-Means

# Importar las librerías
import pandas as pd

data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Wholesale customers data.csv')
data
#print(data)
# Siempre hay que hacer un análisis previo de los datos

# Channel: La etiqueta que van a ser supermercados, tiendas, hoteles, gasolineras... que hagan la venta al pormenor/clientes
# Region: nos da cuenta de qué sitio es
# Fresh, Milk, Grocery, Frozen, Detergents_Paper, Delicassen; lo que nos importa, los productos, su gasto anual

# Vamos a ver ahora la forma de estos datos
data.shape
# (440, 8): 440 datos y 8 columnas
# como es aprendizaje no supervisado es por lo que no tenemos una variable target o dependiente, las 8 son independientes

# Vamos a buscar missing values
data.isnull().sum()
# Podemos ver se obtiene
#Channel          0
#Region           0
#Fresh            0
#Milk             0
#Grocery          0
#Frozen           0
#Detergents_Paper 0
#Delicassen        0
#dtype: int 64

# y ese dtype: int 64 nos indica que son todos enteros,
# se puede comprobar para cada variable por individual con la siguiente línea
#data.dtypes

# Una vez realizado este preprocesamiento tendremos que coger de 'data' unas 3 filas para poder comprobar nuestros resultados,
# ya que se trata de un problema no supervisado

# Vamos a seleccionar datos al azar para tomarlos como muestras

# Ahora para el análisis de datos tendremos que eliminar las columnas que no afectan a nuestros datos, Channel y Region
# que además de porque sabemos a qué hacen referencia, podemos ver los rangos son muy diferentes.
data = data.drop(["Region", "Channel"], axis = 1)
data

indexes = [42, 123, 404]
sample = pd.DataFrame(data.iloc[indexes], columns = data.keys()).reset_index(drop = True)
sample
```

```

data = data.drop(indexes, axis = 0)
data
# Una vez generada nuestra muestra con 3 valores que luego ajustaremos a los centroides o para chequear que pertenecen
# a esos, es decir, un subconjunto test, para lo cual necesitaremos de eliminarlos de data para evitar correlaciones

# El siguiente paso debe ser el escalamiento de los datos, ya que son muy diferentes en magnitud

from sklearn import preprocessing

scaled_data = preprocessing.Normalizer().fit_transform(data)
scaled_sample = preprocessing.Normalizer().fit_transform(sample)
# Es importante recordar que muestras debe tener el mismo número de columnas que data, por lo que si hacemos alguna modificación en una de ellas deberemos hacer lo mismo en la otra para evitar posteriores errores

scaled_data

### Toca implementar el Machine Learning después de todo este preprocesamiento de datos

from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score

X = scaled_data.copy()
X

# Como ya sabemos antes de implementar K-Means tendremos que estimar el número de Clusters, vamos a usar los métodos de selección del número de Clusters que tenemos;

# Método del Codo

clusters = []
K = range(1,20)
for k in K:
    Modelokmean = KMeans(n_clusters = k)
    Modelokmean.fit(X)
    clusters.append(sum(np.min(cdist(X, Modelokmean.cluster_centers_, 'euclidean'), axis = 1))/X.shape[0])
# plt.plot(K, clusters, 'bx-')
plt.plot(K, clusters)
plt.xlabel('Clusters')
plt.ylabel('Suma total errores cuadráticos')
plt.title('Método del codo')
plt.show()

# Visto este método yo diría K=6 número de Clusters

# Método de La silueta

silhouette = []

```

```
K = range(2,21)
for k in K:
    kmeans = KMeans(n_clusters = k)
    kmeans.fit(X)
    labels = kmeans.labels_
    silhouette.append(silhouette_score(X, labels, metric = 'euclidean'))
plt.plot(K, silhouette)
plt.xlabel('Clusters')
plt.ylabel('Puntuaje de la Silueta')
plt.title('Método de la Silueta')
plt.show()

# En teoría debería de haber un máximo en K=6

# Conocido el número de Clusters definimos el algoritmo
kmeans = KMeans(n_clusters = 6, init = 'k-means++', max_iter = 300, n_init = 10)

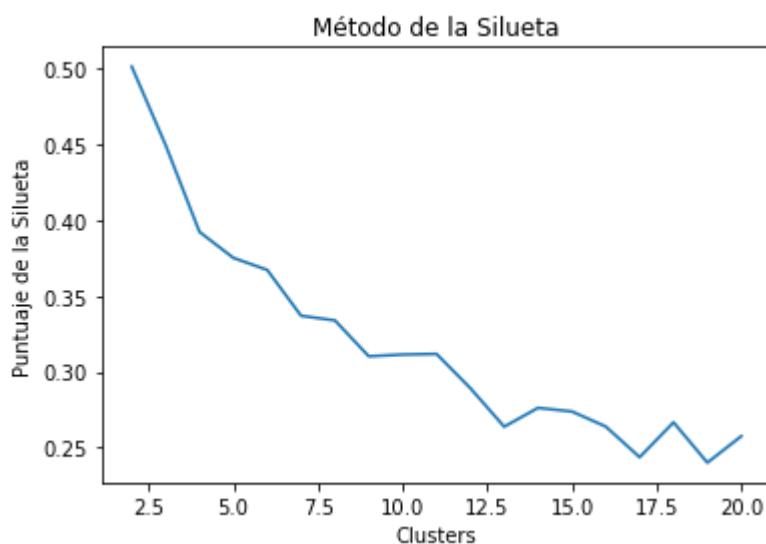
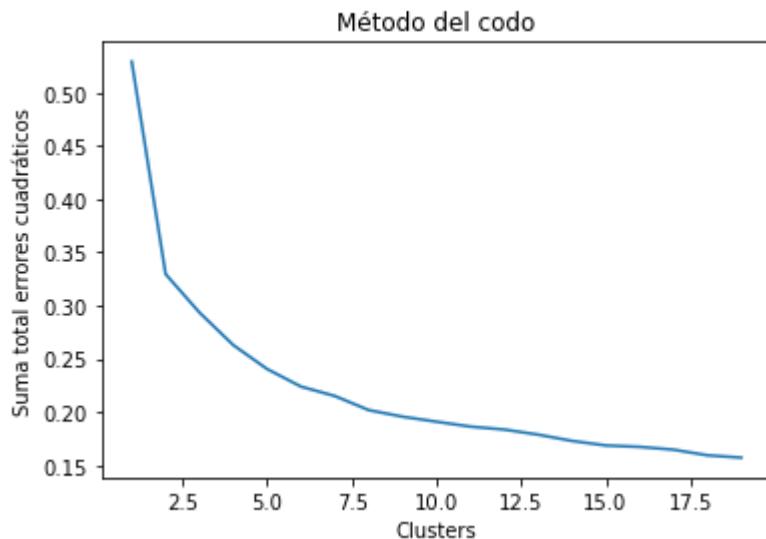
# Realizamos el ajuste a nuestros datos
kmeans.fit(X)

# Ahora queremos obtener las etiquetas de los Clusters y las posiciones de los centroides
labels = kmeans.labels_
centroides = kmeans.cluster_centers_
#print('Labels:',labels)
#print('Centroides:',centroides)

# A continuación vamos con los datos de scaled_sample para hacer nuestra predicción

sample_pred = kmeans.predict(scaled_sample)

for i in enumerate(sample_pred):
    print('Muestra ',i,'-ésima en el Cluster ', sample_pred)
```



Muestra (0, 5) -ésima en el Cluster [5 5 1]
Muestra (1, 5) -ésima en el Cluster [5 5 1]
Muestra (2, 1) -ésima en el Cluster [5 5 1]

In [56]:

```

#### Vamos a pintar Los Clusters pero antes hay que aplicar reducción de dimensionalidad
#### No es muy difícil de entender, si tenemos 6 columnas, 6D, y si queremos un grafo 2
D, tenemos un problema

from sklearn.decomposition import PCA

modelo_pca = PCA(n_components = 2)
modelo_pca.fit(X)
pca = modelo_pca.transform(X)

# Se va a aplicar la reducción de dimensionalidad a Los centroides (Clusters)
centroides_pca = modelo_pca.transform(centroides)

# Se pueden definir Los colores de cada Cluster
colours = ['blue', 'red', 'green', 'orange', 'gray', 'brown']

# Se le asignan Los colores a Los Clusters
cluster_colour = [colours[labels[i]] for i in range(len(pca))]

# Se grafica Los componentes del PCA
plt.scatter(pca[:, 0], pca[:, 1], c = cluster_colour, marker = 'o', alpha = 0.4)

# Se grafican Los centroides
plt.scatter(centroides_pca[:, 0], centroides_pca[:, 1], marker = 'x', s = 100, linewidths = 3, c = colours)

# Se guardan en variables para que sea más fácil el código
x_vector = modelo_pca.components_[0] * max(pca[:,0])
y_vector = modelo_pca.components_[1] * max(pca[:,1])
columns = data.columns

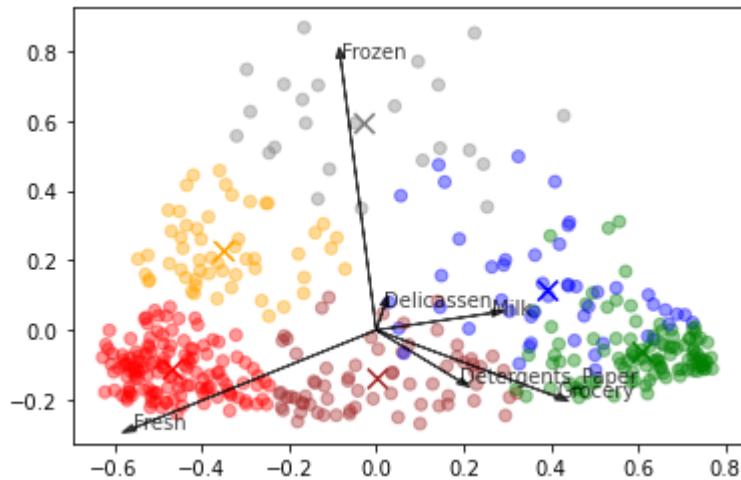
# Se grafican Los nombres de Los Clusters con la distancia del vector
for i in range(len(columns)):
    # Se grafican Los vectores
    plt.arrow(0, 0, x_vector[i], y_vector[i], color = 'black', width = 0.0005, head_width = 0.02, alpha = 0.75)

    # Se ponen Los nombres
    plt.text(x_vector[i], y_vector[i], list(columns)[i], color = 'black', alpha = 0.75)

plt.show()

### Las cruces marcan el centroide de cada distribución y las flechas es el resultado de la proyección multidimensional y
### vemos es más importante en este caso el que tenga la flecha más larga... frozen y resh; más importantes
### Los otros se deben de colocar juntos ya que son más probables que se cojan si están juntos
### La diferencia con el Aprendizaje Supervisado es que aquí no hay un resultado sino que tenemos que hacer el análisis nosotros

```



Agrupamiento Jerárquico

Son un tipo de algoritmo que trabajan considerando inicialmente todos los puntos de datos como Clusters y luego los van agrupando hasta llegar a un único Cluster, buscando encontrar una jerarquía en los datos que tendrán forma de agrupamiento en forma de árbol o dendrograma. La raíz de un árbol es el Cluster único y cada hoja es un único dato que inicialmente si tenemos N datos, tendremos N hojas, siendo cada una de ellas un Cluster.

Agrupamiento Jerárquico Aglomerativo

Tenemos que seleccionar una métrica de distancia, en cada iteración buscaremos agrupar dos Clusters en uno considerando aquellos con la vinculación media más pequeña, es decir los más cercanos entre sí ya que por lo tanto serán los más parecidos entre sí y deben ser combinados. De esta forma se itera el proceso y podremos ver se llegará a un único Cluster que agrupe todos los datos al final y así podremos determinar cuántos Clusters querremos al final, es decir, cuándo podemos dejar de agrupar o dejar de construir el árbol.

El algoritmo como tal es sencillo, calcular la matriz de proximidad (de cada Cluster al resto de Clusters), considerando inicialmente cada uno de los puntos iniciales como Cluster fusionar los dos Clusters más cercanos y actualizar la matriz, una vez hecho este proceso, iterarlo hasta que solo quede un único Cluster. En esta técnica la clave es el cálculo de proximidad entre dos Clusters.

Una vez tenemos la matriz en cada iteración lo que hacemos es buscar la menor de esas distancias y fusionaremos esos Clusters en uno único, una vez hecho esto veremos hemos reducido en 1 el número de Clusters, volvemos a determinar la matriz de distancias y tendremos que iterar el proceso hasta que solo quede uno.

Medidas de Vinculación/Linkage Methods: como se puede comprobar estamos calculando la distancia de cada uno de los puntos con todos los demás y luego la matriz se actualiza para medir la distancia entre Clusters. Pero claro, ¿qué consideramos la unión de dos Clusters? En muchas ocasiones consideramos diferentes Medidas de Vinculación o Linkage Methods.

-Enlace Completo: Calcula la distancia máxima entre Clusters antes de la vinculación, para cada par de Clusters, los calcula y los fusiona antes de la fusión, calcula todas las fusiones posibles y se queda con la que minimice esa distancia, en otras palabras, la distancia entre los elementos más lejanos.

-Enlace Simple: La distancia entre los dos Clusters como la distancia más corta entre un elemento de cada Cluster con el otro, y calcula la distancia mínima de todas las posibles antes de la fusión. Este enlace se puede utilizar para detectar outliers, que serán los últimos en unirse.

-Enlace Promedio: Similar al método de Enlace Completo, pero en este caso calcula la distancia promedio.

-Método del Centroide: Calcula el centroide de ambos Clusters y considera la distancia entre ellos antes de fusionarse.

-Método de Ward: Consideran las sumas de las distancias cuadradas y las fusionan para minimizarlas. Estadísticamente se está reduciendo la varianza de cada Cluster resultante

La elección de cada Clúster depende de cada quién y no hay un método rápido y robusto mejor que el resto. Además, diferentes Métodos de Vinculación conducen a diferentes Clusterings.

También hay que tener en cuenta las métricas de distancias:

-Distancia Euclideana: distancia ordinaria en línea recta entre dos puntos del espacio métrico (euclidean).

-Distancia Manhattan: similar a la Euclideana pero tenemos ahora que nos desplazamos primero en el eje horizontal y luego en el vertical, como una torre en el ajedrez.

-Distancia del Coseno: reduce el ruido al tener en cuenta la forma de las variables más que sus valores, tiende a asociar observaciones que tienen las mismas variables máximas y mínimas independientemente de su valor efectivo.

Dendrograma: Es la forma de mostrar los resultados de la agrupación jerárquica, puede ser interpretado de abajo a arriba, abajo siendo cada dato por individual agrupado con el más cercano y luego podemos ver se van agrupando hasta que quedan todos agrupados en un único Cluster arriba. La forma en la que elegir el número de Clusters se puede hacer observando el dendrograma y buscando el número de líneas verticales cortadas al trazar una línea en el eje horizontal, que puede cortar la máxima distancia vertical sin intersectar un Cluster. A medida que esta línea horizontal de corte vaya bajando podremos encontrar vamos a tener mayor número de Clusters a más líneas verticales corte.

De nuevo, usar esa línea horizontal no es la forma más efectiva para poder dar una estimación del número de Clusters sino que tendremos que ver también podremos usar otros métodos como el Método del Codo o el Método de la Silueta.

Agrupamiento Jerárquico Divisivo

Un agrupamiento divisivo es aquel en el que consideramos inicialmente todas las muestras como un único Cluster y se va separando en sub-Clusters, que no sean similares, hasta que cada dato por individual sea un Cluster, es exactamente lo opuesto a la aglomerativa, solo que la técnica divisiva no se utiliza tanto.

Como ventaja, la agrupación jerárquica no requiere de especificarle el número de Clusters e incluso podemos seleccionar qué número de Clusters se ve mejor a medida que se va construyendo el árbol, además, este algoritmo no es sensible a la métrica de distancia mientras que con otros algoritmos la elección de esta métrica es crítica, y además, un caso particular bueno en el que este algoritmo es realmente útil es cuando los datos subyacentes tienen una estructura jerárquica y es lo que se desea encontrar. Podemos ver además otra de las ventajas es que podremos ver las asociaciones entre Clusters y podremos tener en cuenta de qué distancias intra-Cluster e inter-Cluster tenemos.

Sin embargo tiene sus desventajas como en términos de eficiencia (computacionalmente costoso), a diferencia de la complejidad lineal y gaussiana de otros algoritmos. Es por esto que no se recomienda para grandes conjuntos de datos sino que K significa usar en tiempo real. Es sensible al ruido y outliers, además tenemos que indicar K, ese número de Clusters como en K-Means.

Agrupamiento Jerárquico

```
from sklearn.cluster import AgglomerativeClustering
```

Definir el algoritmo

```
agg_cluster = AgglomerativeClustering()
```

Vamos a ver los parámetros que vamos a configurar para tener un mejor control del algoritmo

n_clusters: (default = 2), y tendremos que estimarlo de forma previa y luego calcularlo

affinity: (default = 'euclidean'), hace referencia a las medidas de distancia de la vinculación, además de 'euclidean', tenemos 'manhattan', 'cosine', 'precomputed', 'ward', 'i1' o 'i2'. En caso de querer usar 'ward' tendremos que definir 'euclidean', y en caso de querer usar 'precomputed' tendremos que tener una matriz de distancia en lugar de una matriz de similitud

linkage: (default = 'ward') el criterio a usar en la vinculación, y puede ser también 'euclidean', 'average', 'complete' o 'single'.

Entrenamos el modelo

```
agg_cluster.fit(X)
```

Recordar no es un algoritmo supervisado y tendremos que considerar solo las variables independientes, no hay variables dependientes

Realizamos entrenamiento y además una predicción

```
agg_cluster.fit_predict(sample)
```

Una vez implementados estas funciones vamos a ver tendremos como resultado el número de Clusters

asociado a cada dato en la predicción

In [57]:

```
##### Agrupamiento Jerárquico
```

```
# Importar las librerías
import pandas as pd

data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Wholesale customers data.csv')
#print(data)
# Siempre hay que hacer un análisis previo de los datos

# Channel: la etiqueta que van a ser supermercados, tiendas, hoteles, gasolineras... que hagan la venta al por menor/clientes
# Region: nos da cuenta de qué sitio es
# Fresh, Milk, Grocery, Frozen, Detergents_Paper, Delicassen; lo que nos importa, los productos, su gasto anual

# Vamos a ver ahora la forma de estos datos
data.shape
# (440, 8): 440 datos y 8 columnas
# como es aprendizaje no supervisado es por lo que no tenemos una variable target o dependiente, las 8 son independientes

# Vamos a buscar missing values
data.isnull().sum()
# Podemos ver se obtiene
#Channel          0
#Region           0
#Fresh            0
#Milk             0
#Grocery          0
#Frozen           0
#Detergents_Paper 0
#Delicassen        0
#dtype: int 64

# y ese dtype: int 64 nos indica que son todos enteros,
# se puede comprobar para cada variable por individual con la siguiente línea
#data.dtypes

# Una vez realizado este preprocesamiento tendremos que coger de 'data' unas 3 filas para poder comprobar nuestros resultados,
# ya que se trata de un problema no supervisado

# Vamos a seleccionar datos al azar para tomarlos como muestras

# Ahora para el análisis de datos tendremos que eliminar las columnas que no afectan a nuestros datos, Channel y Region
# que además de porque sabemos a qué hacen referencia, podemos ver los rangos son muy diferentes.
data = data.drop(["Region", "Channel"], axis = 1)
data

indexes = [42, 123, 404]
sample = pd.DataFrame(data.iloc[indexes], columns = data.keys()).reset_index(drop = True)
sample
data = data.drop(indexes, axis = 0)
```

```

data
# Una vez generada nuestra muestra con 3 valores que luego ajustaremos a los centroides
o para chequear que pertenecen
# a esos, es decir, un subconjunto test, para lo cual necesitaremos de eliminarlos de d
ata para evitar correlaciones

# El siguiente paso debe ser el escalamiento de los datos, ya que son muy diferentes en
magnitud

from sklearn import preprocessing

scaled_data = preprocessing.Normalizer().fit_transform(data)
scaled_sample = preprocessing.Normalizer().fit_transform(sample)
# Es importante recordar que muestras debe tener el mismo número de columnas que data,
por lo que si hacemos alguna
# modificación en una de ellas deberemos hacer lo mismo en la otra para evitar posterio
res errores

scaled_data

### Toca implementar el Machine Learning después de todo este preprocesamiento de datos
# Como ya sabemos antes de implementar Agrupamiento Jerárquico tendremos que estimar el
número de Clusters, vamos a usar los
# métodos de selección del número de Clusters que tenemos para este algoritmo, el dendr
ograma
#¿Podrían también usarse el Método del Codo y Silueta?

# Necesitamos esta librería para hacer ese dendrograma
import scipy.cluster.hierarchy as shc
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import numpy as np
from sklearn.metrics import silhouette_score

X = scaled_data.copy()
X

plt.figure(figsize=(10,7))
plt.title('Dendrogram')
dendrogram = shc.dendrogram(shc.linkage(X, method = 'ward'))
plt.show()

# Tenemos que ubicar la línea vertical más larga y cortarla en lo más bajo posible cuan
do esa línea horizontal no corte
# a otras en el eje horizontal, y podemos ver de resultado es 3, ya que la línea recta
más larga en la vertical es la de
# arriba del todo a la izquierda y si se corta en la parte más baja y entonces tendremo
s el naranja y dos verdes, 3 Clusters

# Método del Codo

clusters = []
K = range(1,20)
for k in K:
    Modelokmean = KMeans(n_clusters = k)
    Modelokmean.fit(X)

```

```
    clusters.append(sum(np.min(cdist(X, Modelokmean.cluster_centers_, 'euclidean')), axis = 1))/X.shape[0])
# plt.plot(K, clusters, 'bx=')
plt.plot(K, clusters)
plt.xlabel('Clusters')
plt.ylabel('Suma total errores cuadráticos')
plt.title('Método del codo')
plt.show()

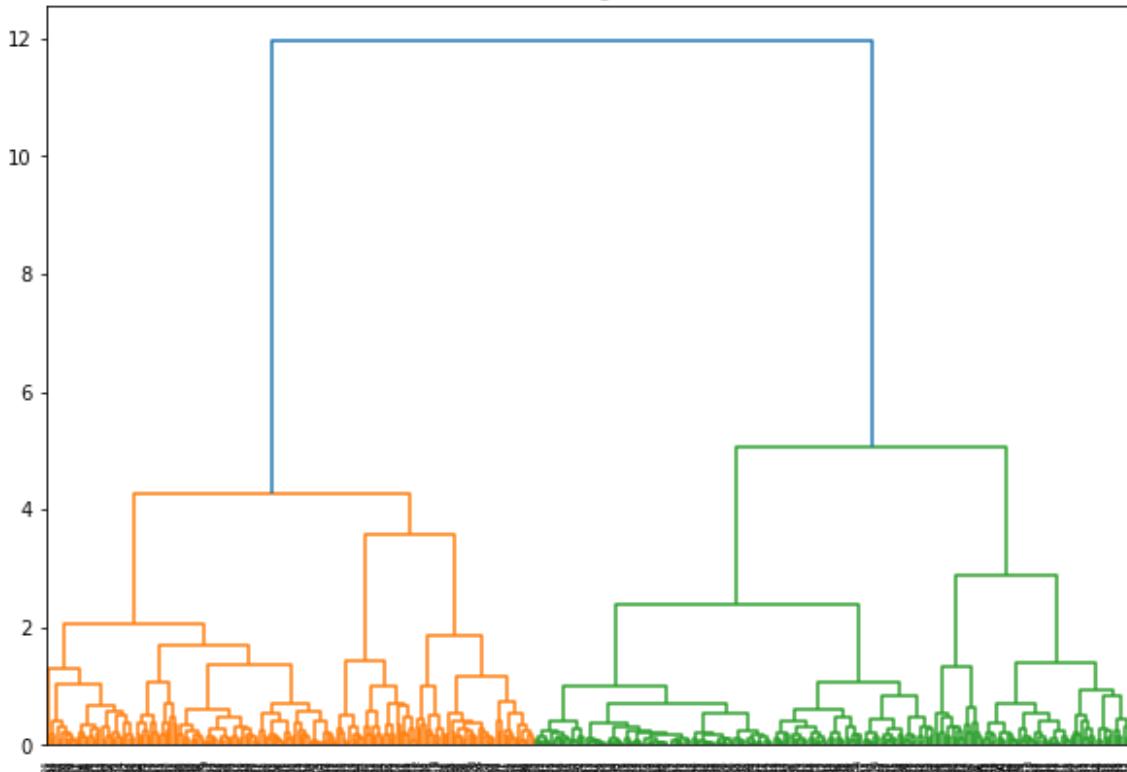
# Visto este método yo diría K=6 número de Clusters

# Método de La silueta

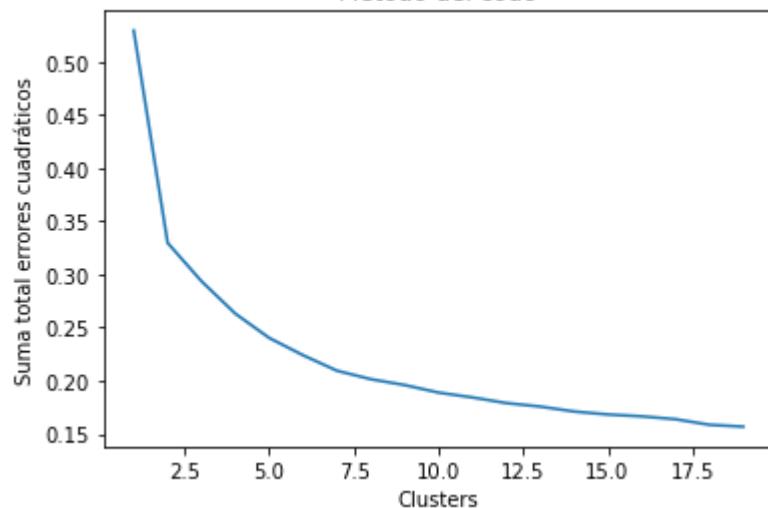
silhouette = []

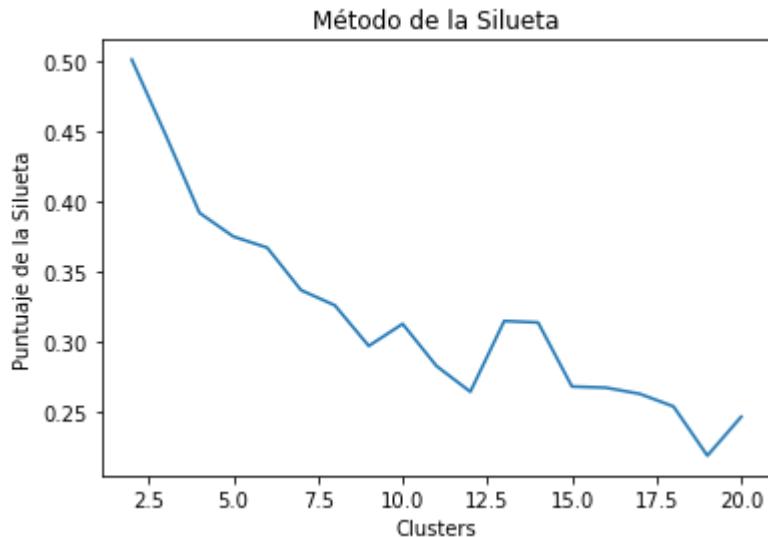
K = range(2,21)
for k in K:
    kmeans = KMeans(n_clusters = k)
    kmeans.fit(X)
    labels = kmeans.labels_
    silhouette.append(silhouette_score(X, labels, metric = 'euclidean'))
plt.plot(K, silhouette)
plt.xlabel('Clusters')
plt.ylabel('Puntuaje de la Silueta')
plt.title('Método de la Silueta')
plt.show()
```

Dendrogram



Método del codo





In [58]:

```
from sklearn.cluster import AgglomerativeClustering

# Definimos el algoritmo
agg_cluster = AgglomerativeClustering(n_clusters = 3, affinity = 'euclidean', linkage = 'ward')

# Entrenamos el algoritmo
agg_cluster.fit(X)

# A continuación vamos con los datos de scaled_sample para hacer nuestra predicción
X_pred = agg_cluster.fit_predict(X)

# A continuación vamos con los datos de scaled_sample para hacer nuestra predicción
sample_pred = agg_cluster.fit_predict(scaled_sample)

for i, pred in enumerate(sample_pred):
    print('Muestra: ', i, 'en el Cluster ', pred)
```

```
Muestra:  0 en el Cluster  2
Muestra:  1 en el Cluster  1
Muestra:  2 en el Cluster  0
```

Agrupamiento DBSCAN

Este algoritmo basado en la densidad similar a Mean Shift pero con un par de ventajas notables, las siglas significan Agrupamiento Espacial Basado en la Densidad de Aplicaciones con Ruido. DBSCAN se basa en la densidad para identificar Clusters de diferentes y variadas formas aún en un conjunto de datos con ruido y outliers. Se trata de un algoritmo que requiere de dos parámetros, epsilon (la distancia a la que podemos considerar que si están dos puntos, pertenezcan a un mismo Cluster), es decir, dos puntos a distancia epsilon o menor que epsilon van a considerarse vecinos en un mismo Cluster, el segundo parámetro es el número mínimo de puntos de datos vecinos necesarios como para considerar que se ha formado un Cluster. También hay que considerar que se tienen 3 tipos de datos;

- Los puntos de datos en un Punto Núcleo (básicamente un Cluster, es decir, una agrupación de puntos ya construida que están a distancia epsilon o menor y son más del número mínimo especificado).
- Los puntos de Datos en el Borde (en el borde del área delimitada por esa distancia epsilon, vemos es un punto en esa frontera, a distancia epsilon del centro de un punto núcleo, y cuando a un área en distancia epsilon agrega menos puntos vecinos que el punto núcleo ya creado).
- Los puntos de datos de ruido son cualesquiera que no sean Puntos Núcleos o Datos en el Borde.

Así, se ve tenemos dos ideas importantes a considerar en DBSCAN, como son el borde de densidad, es decir, si tenemos 2 Puntos Núcleos y que están separados a distancia menor o igual que epsilon, los podemos conectar mediante una línea imaginaria que va a ser ese borde de densidad. Una vez están unidos vemos podemos tener densidades de puntos conectados cuando tenemos un Punto Núcleo A conectado con otro B, que a su vez está conectado con A y un tercero C, el cual conecta con B y otro D, etc. Y así vemos tenemos una cadena de conexión o una trayectoria formada por esos puntos de conexión.

DBSCAN comienza con un conjunto de datos de inicio no visitados aún arbitrario, el vecindario de este punto se extrae usando una distancia epsilon. Si hay un número de puntos elevados en este vecindario, en ese rango epsilon, el proceso de creación de Clusters comienza y el primer punto de datos se convierte en el siguiente al que avanzamos, de lo contrario el punto será etiquetado como ruido (luego sí que podrá ser visitado de nuevo y pasar a ser parte del Cluster); en ambos casos el punto cuenta como visitado.

Típicamente la regla para tomar valores mínimos es tomar los mínimos para ser mayor o igual a la dimensionalidad del conjunto de datos. Si el ruido es elevado se tendrá que elegir un gran número de puntos mínimos. La elección de este número de mínimos depende a su vez del conocimiento del dominio o dataset.

Para la elección de epsilon, se puede realizar el siguiente método, se calcula las distancias vecinas más cercanas en una matriz de puntos, calcular las distancias promedio de cada uno a sus k-vecinos más cercanos, (ese k habrá de ser proporcionado por el usuario y se corresponde con los puntos mínimos), una vez obtenido esto se puede dibujar esas k distancias de menor a mayor, obteniendo una curva ascendente, y el objetivo es determinar la "rodilla" o punto de inflexión de la curva, para que nos indique el valor de la epsilon para poder usarlo en nuestro algoritmo ya que es el umbral antes de un cambio brusco.

Para meter ese primer punto en un nuevo Cluster, los puntos de su vecindario epsilon pasar a ser parte del mismo Cluster también, se itera para todos los nuevos puntos que acaban de ser agregados al Cluster y se hace esto hasta que todos los puntos nuevos del Cluster hayan sido visitados y etiquetados como pertenecientes a ese Cluster.

Lo siguiente es ir a un nuevo punto no visitado y que no pertenezca a ese Cluster y entonces buscaremos de la misma forma encontrando otros Clusters o ruido.

Tenemos ahora ciertas ventajas, como que tampoco requiere del conocimiento del número de Clusters (como sí que era una desventaja presente en K-Means), la elección de puntos mínimos permite eliminar los valores atípicos, es decir, se etiquetará los valores atípicos como ruido, a diferencia del Agrupamiento Mean Shift, que los coloca en algún Cluster incluso si es muy diferente. Además se pueden encontrar Clusters de diferente tamaño y forma, no solo circular. También es excelente para separar Clusters según densidades.

Sin embargo tiene desventajas como su incorrecto funcionamiento en ocasiones cuando el Cluster es de densidad variable, esto se debe a la configuración del umbral de distancia epsilon para identificar los puntos del vecindario variará de un Cluster a otro cuando la densidad variable. También tenemos problemas a grandes dimensiones cuando la distancia epsilon se vuelve difícil de estimar. Es además extremadamente sensible a hiperparámetros, por lo que un ligero cambio en ellos puede ocasionar un gran cambio en los resultados obtenidos.

In [59]:

```
##### Método Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

import numpy as np
import pandas as pd
from sklearn.cluster import DBSCAN
from collections import Counter
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
from pylab import rcParams
rcParams['figure.figsize'] = 14, 6

# Importamos data
data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Wholesale customers data.csv')
data
#print(data)
# Siempre hay que hacer un análisis previo de los datos

# Channel: La etiqueta que van a ser supermercados, tiendas, hoteles, gasolineras... que hagan la venta al pormenor/clientes
# Region: nos da cuenta de qué sitio es
# Fresh, Milk, Grocery, Frozen, Detergents_Paper, Delicassen; lo que nos importa, los productos, su gasto anual

# Vamos a ver ahora la forma de estos datos
data.shape
# (440, 8): 440 datos y 8 columnas
# como es aprendizaje no supervisado es por lo que no tenemos una variable target o dependiente, las 8 son independientes

# Vamos a buscar missing values
data.isnull().sum()
# Podemos ver se obtiene
#Channel      0
#Region       0
#Fresh        0
#Milk         0
#Grocery      0
#Frozen        0
#Detergents_Paper  0
#Delicassen    0
#dtype: int 64

# y ese dtype: int 64 nos indica que son todos enteros,
# se puede comprobar para cada variable por individual con la siguiente línea
#data.dtypes

# Una vez realizado este preprocessamiento tendremos que coger de 'data' unas 3 filas para poder comprobar nuestros resultados,
# ya que se trata de un problema no supervisado

# Vamos a seleccionar datos al azar para tomarlos como muestras

# Ahora para el análisis de datos tendremos que eliminar las columnas que no afectan a nuestros datos, Channel y Region
# que además de porque sabemos a qué hacen referencia, podemos ver los rangos son muy diferentes.
```

```

data = data.drop(["Region", "Channel"], axis = 1)
data

# Convertimos a array tipo numpy para poder meterlo en el modelo
data_np = data.values.astype('float32', copy = False)
data_np

# Normalizamos los datos
scaled_data = preprocessing.Normalizer().fit_transform(data_np)
scaled_data

### HABRÍA QUE CALCULAR UNA ESTIMACIÓN DE EPS Y MIN_SAMPLES PREVIAMENTE CON MÉTODO DEL CODO???

### ???
# Método del Codo
K = range(1,5)
J = range(1,5)
for k in K:
    for j in J:
        dbscan = DBSCAN(eps = k*0.05, min_samples = j, metric = 'euclidean')
        dbscan.fit(scaled_data)
        print('With eps=', k*0.05, 'and min_samples =',j,'we get a number of Clusters=',format(len(Counter(dbscan.labels_))-1))

# Definimos el modelo
dbscan = DBSCAN(eps = 0.05, min_samples = 4, metric = 'euclidean')

# Lo entrenamos
dbscan.fit(scaled_data)

# Visualizamos los resultados separando para ello los outliers de los clusters
outliers_data = data[dbscan.labels_ == -1]
clusters_data = data[dbscan.labels_ != -1]
# Esto lo hace el modelo, nos separa datos en outliers o pertenecientes a Clusters, y así creamos nuestros datasets separados

# Asignamos colores a datos por pertenencia a sus respectivos Clusters en función del índice del Cluster.
colors = dbscan.labels_
colors_clusters = colors[colors != -1]
colors_outliers = 'black'

# Vamos a obtener información de los Clusters

clusters = Counter(dbscan.labels_)
clusters
#print(data[dbscan.labels_ == -1].head())
print('Number of Clusters:', format(len(clusters)-1))

### Vamos a pintar los Clusters pero antes hay que aplicar reducción de dimensionalidad
### No es muy difícil de entender, si tenemos 6 columnas, 6D, y si queremos un grafo 2 D, tenemos un problema

from sklearn.decomposition import PCA

modelo_pca = PCA(n_components = 2)
modelo_pca.fit(scaled_data)
pca = modelo_pca.transform(scaled_data)

```

```

# Se va a aplicar la reducción de dimensionalidad a los centroides (Clusters)
centroides_pca = modelo_pca.transform(dbSCAN.components_)

# Se pueden definir los colores de cada Cluster
colours = ['blue', 'red', 'green', 'orange', 'gray', 'brown', 'purple', 'yellow']
labels = dbSCAN.labels_

# Se le asignan los colores a los Clusters
cluster_colour = [colours[labels[i]] for i in range(len(pca))]

# Se grafica los componentes del PCA
plt.scatter(pca[:, 0], pca[:, 1], c = cluster_colour, marker = 'o', alpha = 0.4)

# Se grafican los centroides
plt.scatter(centroides_pca[:, 0], centroides_pca[:, 1], marker = 'x', s = 100, linewidths = 3)

# Se guardan en variables para que sea más fácil el código
x_vector = modelo_pca.components_[0] * max(pca[:,0])
y_vector = modelo_pca.components_[1] * max(pca[:,1])
columns = data.columns

# Se grafican los nombres de los Clusters con la distancia del vector
for i in range(len(columns)):
    # Se grafican los vectores
    plt.arrow(0, 0, x_vector[i], y_vector[i], color = 'black', width = 0.0005, head_width = 0.02, alpha = 0.75)

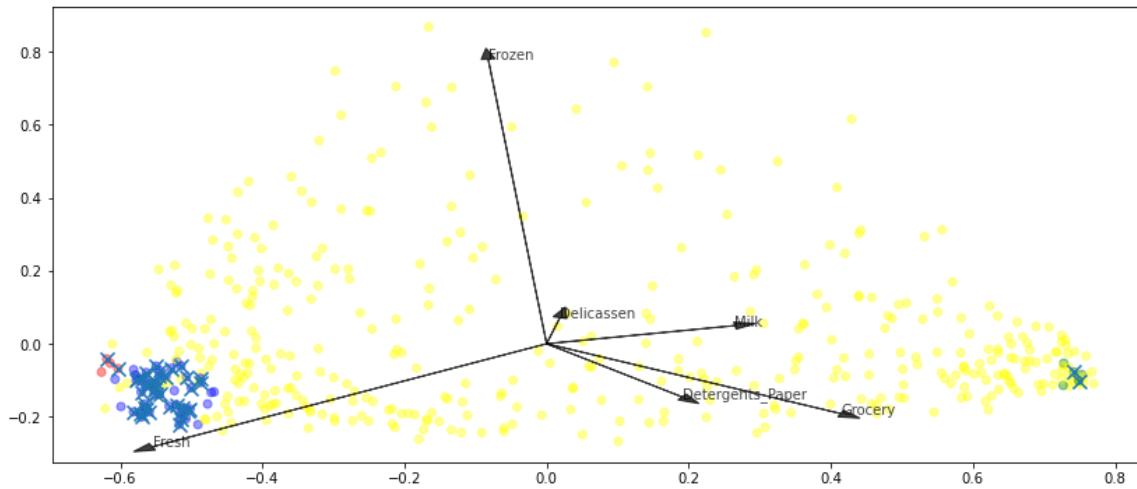
    # Se ponen los nombres
    plt.text(x_vector[i], y_vector[i], list(columns)[i], color = 'black', alpha = 0.75)

plt.show()

### Las cruces marcan el centroide de cada distribución y las flechas es el resultado de la proyección multidimensional y
### vemos es más importante en este caso el que tenga la flecha más larga... frozen y refresh; más importantes
### Los otros se deben de colocar juntos ya que son más probables que se cojan si están juntos
### La diferencia con el Aprendizaje Supervisado es que aquí no hay un resultado sino que tenemos que hacer el análisis nosotros

```

With $\text{eps} = 0.05$ and $\text{min_samples} = 1$ we get a number of Clusters= 367
With $\text{eps} = 0.05$ and $\text{min_samples} = 2$ we get a number of Clusters= 15
With $\text{eps} = 0.05$ and $\text{min_samples} = 3$ we get a number of Clusters= 3
With $\text{eps} = 0.05$ and $\text{min_samples} = 4$ we get a number of Clusters= 3
With $\text{eps} = 0.1$ and $\text{min_samples} = 1$ we get a number of Clusters= 208
With $\text{eps} = 0.1$ and $\text{min_samples} = 2$ we get a number of Clusters= 21
With $\text{eps} = 0.1$ and $\text{min_samples} = 3$ we get a number of Clusters= 11
With $\text{eps} = 0.1$ and $\text{min_samples} = 4$ we get a number of Clusters= 7
With $\text{eps} = 0.1500000000000002$ and $\text{min_samples} = 1$ we get a number of Clusters= 88
With $\text{eps} = 0.1500000000000002$ and $\text{min_samples} = 2$ we get a number of Clusters= 16
With $\text{eps} = 0.1500000000000002$ and $\text{min_samples} = 3$ we get a number of Clusters= 5
With $\text{eps} = 0.1500000000000002$ and $\text{min_samples} = 4$ we get a number of Clusters= 3
With $\text{eps} = 0.2$ and $\text{min_samples} = 1$ we get a number of Clusters= 34
With $\text{eps} = 0.2$ and $\text{min_samples} = 2$ we get a number of Clusters= 8
With $\text{eps} = 0.2$ and $\text{min_samples} = 3$ we get a number of Clusters= 5
With $\text{eps} = 0.2$ and $\text{min_samples} = 4$ we get a number of Clusters= 4
Number of Clusters: 3



Agrupamiento utilizando modelos de media gaussiana (Gaussian Mean Model/ Gaussian Mixture Model, GMM)

Uno de los mayores inconvenientes de K-means es su ingenuo uso del valor medio para el centro del Cluster, también por su uso del valor medio podemos ver falla cuando el Cluster no es circular. Los GMM nos dan más flexibilidad, asumimos que los datos están distribuidos de forma normal o gaussiana, lo cual es una aproximación menos restrictiva que decir que es circular usando la media. Tomando un ejemplo en dos dimensiones tendremos media y desviación estándar, por lo que los Clusters podrán tener formas elípticas, ya que tendremos desviación estándar tanto en dirección X como en dirección Y. Para encontrar los valores de los parámetros de desviación estándar y valor promedio usaremos un algoritmo de Maximización de Expectativas (EM).

Tenemos que recordar la distribución Gaussiana es tal que

$$f(x) = (1/(sigmasqrt(2pi))\exp(-(x-\mu)^2 / 2\sigma^2)$$

Donde μ es el valor medio de la distribución y σ es la desviación estandar. Lo importante de esta distribución es que es simétrica, unimodal y tiene la propiedad de que el data a medida que se aleja del valor medio es menos probable que ocurra y podemos ver tenemos casi todos los datos centrados en el valor medio. Sin embargo, podemos ver Maximización de Expectativas (EM) se va a basar principalmente en el data alejado del valor medio, en las colas de la Gaussiana.

Lo siguiente podría ser intentar ver cualquier distribuciones de tal forma que las asumamos como una combinación de Gaussianas, de tal forma que se puedan ver como Gaussianas latentes que combinadas acaben formando alguna distribución y queremos saber de qué forma se combinan dichas gaussianas intrínsecas y para ello es que tenemos el Maximización de Expectativas ("Expectation Maximization Algorithm", EM) aplicado para este nuestro caso de distribuciones gaussianas o normales y ver como, agrupadas, conforman el patrón de datos que se pueda encontrar.

K-Means y EM/GMM tienen ciertas similitudes y diferencias, por ejemplo en K-Means se asignaban los datos como perteneciendo a algún Cluster concreto, sin ninguna otra pertenecencia a otros Clusters, Hard Assignment, mientras que en EM/GMM lo que vamos a tener es, para cada dato, una cierta probabilidad de pertenecencia a cada uno de los Clusters presentes, no algo binario como en K-Means, así esto es Soft/Probabilistic Assignment.

La otra diferencia entre K-Means y EM/GMM va a ser que en K-Means todos tienen la asunción de que todos los Clusters eran circulares o esféricos (lo cual no suele ser cierto pero aún así funciona bien así que) y por ello tenían la misma varianza, mientras que con esta asunción podemos perdernos patrones presentes en dicha distribución ya que hemos asumido que sean esféricos cuando no lo son. Por otro lado en EM/GMM tenemos que asumen distribuciones que bien pueden ser elípticas o elipsoides, por lo que es un concepto y algoritmo más flexible, ya que con EM desde un primer momento se va a calcular la varianza de forma explícita para poder utilizarla en vez de asumir la misma para todos.

Lo primero en EM es definir el número de Clusters, tal y como hacíamos con K-means e inicializamos aleatoriamente los K parámetros de esa distribución gaussiana para cada Cluster, sino se puede intentar hacer a ojo, pero podemos ver da igual ya que aunque al principio no se ajuste, se optimizan rápidamente (ya que en cada iteración se va a recalcular los centros de cada uno de los Clusters en base a un análisis de los puntos más cercanos al anterior centro). Dichas iteraciones finalizaban tras un cierto número de repeticiones o cuando los centros no variaban ya mucho de sus posiciones encontradas.

Lo siguiente a realizar sería que dadas estas distribuciones gaussianas para cada Cluster, calcular la probabilidad (y desviación estándar de cada probabilidad) de que un dato pertenezca a cada Cluster particular, cuanto más cerca esté de un centro de los Cluster más probable será que pertenezca a este

grupo, ya que por ser distribución gaussiana asumimos los datos estarán muy cercanos al centro del Cluster.
(Fase de Expectation)

Una vez hechas estas probabilidades calculamos un nuevo valor de esos parámetros de tal forma que maximicen las probabilidades de pertenecer a sus Clusters (es decir, a mayor sea la probabilidad de pertenencia ese punto o puntos van a ser más relevantes en el Cluster por lo que es más probable que sean asignados al centro), se calculan estos nuevos parámetros usando una suma ponderada de las posiciones de los puntos de datos, donde las ponderaciones son los puntos de datos que pertenecen a ese dato en particular. (Fase de Maximization)

Todos estos pasos(E + M) se repiten hasta la convergencia y hasta que no se cambie mucho de una iteración a la siguiente.

Tienen una serie de ventajas, son mucho más flexibles en términos de la forma de los Clusters debido al parámetro de la desviación estándar, de forma que se pueden adaptar a elipsoides en vez de solo a círculos como K-means. También, como los modelos de mezcla gaussiana utilizan probabilidades pueden tener conglomerados múltiples por puntos de datos, si un punto de datos está en medio de dos Clusters, no le asignamos a un Cluster u otro sino que le indicamos una probabilidad de pertenencia a cada Cluster.

In [60]:

```

##### Agrupamiento utilizando modelos de media gaussiana (Expectation Maximization algorithm, EM )
##### (Gaussian Mean Model/ Gaussian Mixture Model, GMM )
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import scipy
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from scipy.spatial.distance import cdist
from sklearn.metrics import silhouette_score

data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Wholesale customers data.csv')

data.shape

data.isnull().sum()

data = data.drop(['Region', 'Channel'], axis = 1)
data

features = ['Fresh', 'Milk']

X = data[features] # En caso de querer hacerlo solo para dos variables

# Convertimos a array tipo numpy para poder meterlo en el modelo
data_np = data.values.astype('float32', copy = False)
data_np

# Normalizamos los datos
scaled_data = preprocessing.Normalizer().fit_transform(data_np)
scaled_X = preprocessing.Normalizer().fit_transform(X)
scaled_data

### Vamos a ver qué pasa si se implementa el algoritmo con solo 2 variables independientes o características
data.head()

#EM = GaussianMixture(n_components = 3)
#n_components como una estimación del número de Clusters

#EM.fit(scaled_X)

#clusters = EM.predict(scaled_X)
#clusters

# Otra sub-función es para saber las probabilidades
#clusters_p = EM.predict_proba(scaled_X)
#clusters_p

# Al igual que en el caso de K-Means nos interesa buscar ahora el número de Clusters

```

Método de la silueta

```

silhouette = []
K = range(2,21)
for k in K:
    EM = GaussianMixture(n_components = k)
    EM.fit(scaled_data)
    clusters = EM.predict(scaled_data)
    silhouette.append(silhouette_score(scaled_data, clusters, metric = 'euclidean'))
plt.plot(K, silhouette)
plt.xlabel('Clusters')
plt.ylabel('Puntuaje de la Silueta')
plt.title('Método de la Silueta')
plt.show()
# Se ve el segundo pico en torno al 6

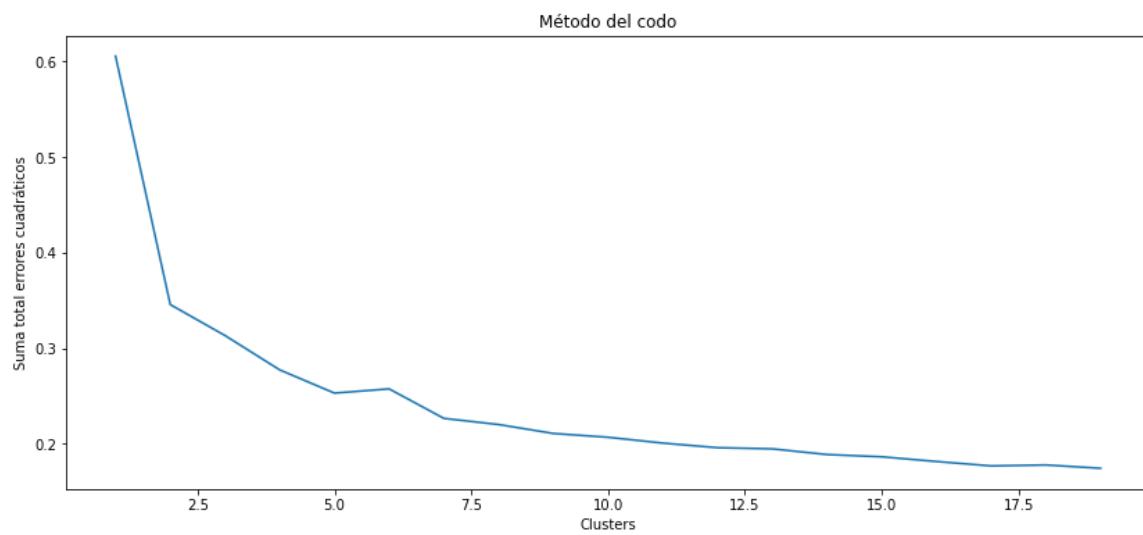
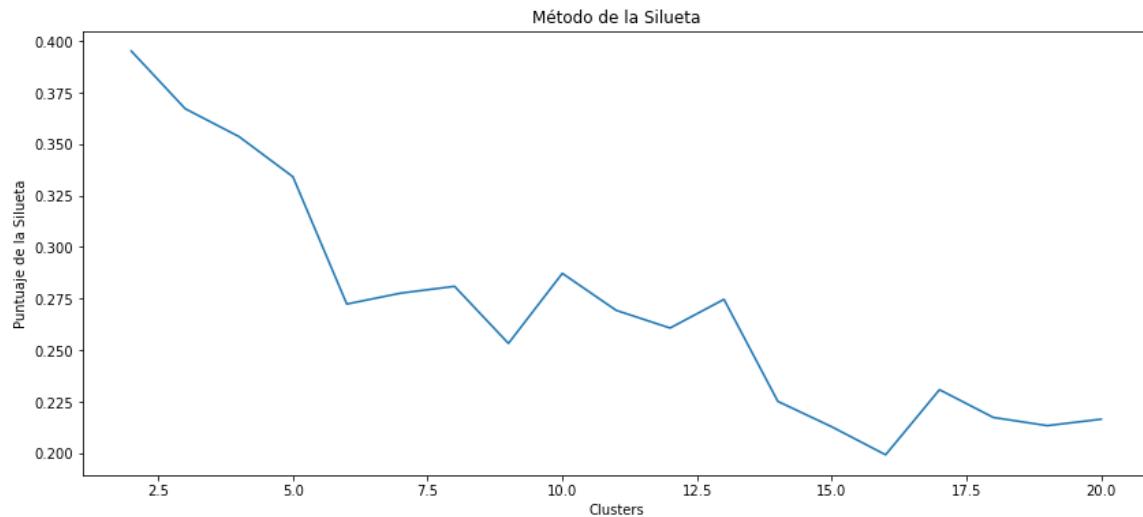
```

Método del Codo

```

clust = []
K = range(1,20)
for k in K:
    EM = GaussianMixture(n_components = k)
    EM.fit(scaled_data)
    centroides = np.empty(shape=(EM.n_components, scaled_data.shape[1]))
    for i in range(EM.n_components):
        density = scipy.stats.multivariate_normal(cov=EM.covariances_[i], mean=EM.means_[i]).logpdf(scaled_data)
        centroides[i, :] = scaled_data[np.argmax(density)]
    clust.append(sum(np.min(cdist(scaled_data, centroides, metric = 'euclidean'), axis=1)/scaled_data.shape[0]))
    #clusters debería de ser clusters_centers, por eso que tendría que ser las coordenadas de los clusters
plt.plot(K, clust)
plt.xlabel('Clusters')
plt.ylabel('Suma total errores cuadráticos')
plt.title('Método del codo')
plt.show()

```



In [61]:

```

#### Vamos a pintar Los Clusters pero antes hay que aplicar reducción de dimensionalidad
#### No es muy difícil de entender, si tenemos 6 columnas, 6D, y si queremos un grafo 2
D, tenemos un problema

EM = GaussianMixture(n_components = 6)
EM.fit(scaled_data)
centroides = np.empty(shape=(EM.n_components, scaled_data.shape[1]))
for i in range(EM.n_components):
    density = scipy.stats.multivariate_normal(cov=EM.covariances_[i], mean=EM.means_[i]).logpdf(scaled_data)
    centroides[i, :] = scaled_data[np.argmax(density)]

from sklearn.decomposition import PCA

modelo_pca = PCA(n_components = 2)
modelo_pca.fit(scaled_data)
pca = modelo_pca.transform(scaled_data)

# Se va a aplicar la reducción de dimensionalidad a los centroides (Clusters)
centroides_pca = modelo_pca.transform(centroides)

# Se pueden definir los colores de cada Cluster
colours = ['blue', 'red', 'green', 'orange', 'gray', 'brown']

labels = EM.predict(scaled_data)

# Se le asignan los colores a los Clusters
cluster_colour = [colours[labels[i]] for i in range(len(pca))]

# Se grafica los componentes del PCA
plt.scatter(pca[:, 0], pca[:, 1], c = cluster_colour, marker = 'o', alpha = 0.4)

# Se grafican los centroides
plt.scatter(centroides_pca[:, 0], centroides_pca[:, 1], marker = 'x', s = 100, linewidths = 3, c = colours)

# Se guardan en variables para que sea más fácil el código
x_vector = modelo_pca.components_[0] * max(pca[:,0])
y_vector = modelo_pca.components_[1] * max(pca[:,1])
columns = data.columns

# Se grafican los nombres de los Clusters con la distancia del vector
for i in range(len(columns)):
    # Se grafican los vectores
    plt.arrow(0, 0, x_vector[i], y_vector[i], color = 'black', width = 0.0005, head_width = 0.02, alpha = 0.75)

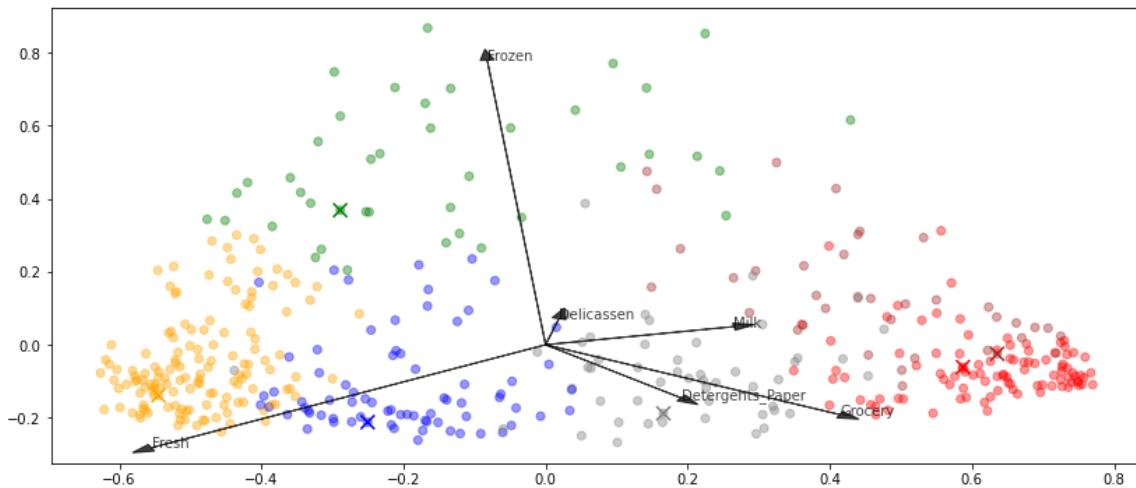
    # Se ponen los nombres
    plt.text(x_vector[i], y_vector[i], list(columns)[i], color = 'black', alpha = 0.75)

plt.show()

### Las cruces marcan el centroide de cada distribución y las flechas es el resultado de la proyección multidimensional y
### vemos es más importante en este caso el que tenga la flecha más larga... frozen y resh; más importantes
### Los otros se deben de colocar juntos ya que son más probables que se cojan si están juntos

```

La diferencia con el Aprendizaje Supervisado es que aquí no hay un resultado sino que tenemos que hacer el análisis nosotros



Agrupamiento Mean Shift

El algoritmo basado en ventanas deslizantes que intenta encontrar áreas densas de puntos de datos. Se basa en el centroide, busca ese punto centroide y lo que hace es ir actualizando los puntos para que los puntos centrales sean la media de los puntos de la ventana deslizante. Estas ventanas son filtradas en una época de post-procesamiento para evitar los duplicamientos y así formando esos agrupamientos. Se elige una geometría para la ventana (por ejemplo circular) y un tamaño (en este caso un radio) y se va haciendo un desplazamiento en cada iteración de menor densidad hasta mayor densidad hasta la convergencia. En cada iteración se va desplazando esa ventana hacia regiones de mayor densidad desplazando su punto central hacia la media de los puntos dentro de la ventana, la densidad es proporcional al número de puntos dentro de ella, se va a ir aumentando esa densidad, hasta que llegue un punto en el que no se pueda agrupar una mayor densidad.

Todo ese proceso se itera con muchas ventanas, cuando se superponen las ventanas se conserva la que mayor densidad agrupe. A continuación en el conjunto de datos los puntos se agrupan en la ventana en la que residen.

A diferencia de K-means no necesita saber el número de Clusters ya que ese desplazamiento medio lo descubre automáticamente. El hecho de que los Clustering converjan hacia el punto de mayor densidad es lo mejor, ya que es muy intuitivo y encaja bien en el modelo. El problema es la selección del tamaño de la ventana puede ser no trivial.

In [62]:

```
##### Agrupamiento Mean Shift

import numpy as np
import pandas as pd
import scipy
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing#, cross_validation
#from sklearn import cross_validation
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift, estimate_bandwidth
from scipy.spatial.distance import cdist
from sklearn.metrics import silhouette_score

data = pd.read_csv(r'C:\Users\PORTATIL\Desktop\Master\Primer Cuatrimestre\Aprendizaje Automático Machine Learning\ML autodidacta\Wholesale customers data.csv')
data

data.shape

data.isnull().sum()

data = data.drop(["Region", "Channel"], axis = 1)
data

indexes = [42, 123, 404]
sample = pd.DataFrame(data.iloc[indexes], columns = data.keys()).reset_index(drop = True)
sample
data = data.drop(indexes, axis = 0)
data

scaled_data = preprocessing.Normalizer().fit_transform(data)
scaled_sample = preprocessing.Normalizer().fit_transform(sample)

scaled_data

# Implementamos el algoritmo
bandwidth = estimate_bandwidth(scaled_data, quantile = 0.1, n_samples = len(scaled_data))

ms = MeanShift(bandwidth = bandwidth, bin_seeding = True)
ms.fit(scaled_data)

labels = ms.labels_
clusters = ms.cluster_centers_

print('len(clusters)=',len(clusters))

from sklearn.decomposition import PCA

modelo_pca = PCA(n_components = 2)
modelo_pca.fit(scaled_data)
pca = modelo_pca.transform(scaled_data)

# Se va a aplicar la reducción de dimensionalidad a los centroides (Clusters)
centroides_pca = modelo_pca.transform(clusters)

# Se pueden definir los colores de cada Cluster
```

```

colours = 2*['blue', 'red', 'green', 'orange', 'gray', 'brown', 'purple', 'yellow']
labels = ms.labels_

# Se le asignan los colores a Los Clusters
cluster_colour = [colours[labels[i]] for i in range(len(pca))]

# Se grafica los componentes del PCA
plt.scatter(pca[:, 0], pca[:, 1], c = cluster_colour, marker = 'o', alpha = 0.4)

# Se grafican los centroides
plt.scatter(centroides_pca[:, 0], centroides_pca[:, 1], marker = 'x', s = 100, linewidths = 3)

# Se guardan en variables para que sea más fácil el código
x_vector = modelo_pca.components_[0] * max(pca[:,0])
y_vector = modelo_pca.components_[1] * max(pca[:,1])
columns = data.columns

# Se grafican los nombres de Los Clusters con La distancia del vector
for i in range(len(columns)):
    # Se grafican los vectores
    plt.arrow(0, 0, x_vector[i], y_vector[i], color = 'black', width = 0.0005, head_width = 0.02, alpha = 0.75)

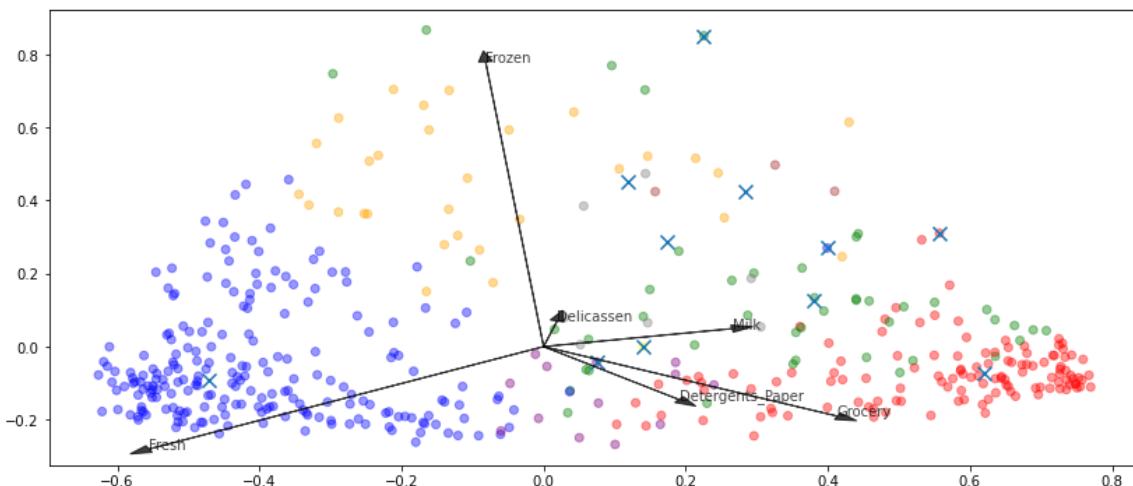
    # Se ponen los nombres
    plt.text(x_vector[i], y_vector[i], list(columns)[i], color = 'black', alpha = 0.75)

plt.show()

### Las cruces marcan el centroide de cada distribución y las flechas es el resultado de la proyección multidimensional y
### vemos es más importante en este caso el que tenga la flecha más larga... frozen y fresh; más importantes
### Los otros se deben de colocar juntos ya que son más probables que se cojan si están juntos
### La diferencia con el Aprendizaje Supervisado es que aquí no hay un resultado sino que tenemos que hacer el análisis nosotros

```

len(clusters)= 11



APRENDIZAJE REFORZADO/ REINFORCEMENT LEARNING

La idea del Aprendizaje Reforzado es recompensar a nuestro sistema cuando realice una buena acción y una penalización cuando se trate de una mala acción. Es decir, podemos imaginar el comeculos, si lo quisiésemos manejar mediante una IA podríamos pensar en estas acciones positivas como los 'cocos' o la 'cereza', y los fantasmas serían una acción negativa.

El concepto que hay detrás consiste en ver el sistema va a estar un estado de 'avance' en un espacio de configuraciones que va a ser el total de posiciones en las que se pueda mover en cada una de estas. Es decir, realmente va a ser espacio y error, va a probar las diferentes posibles configuraciones hasta encontrar la mejor de ellas.

El output de nuestro sistema depende obviamente del input y es un proceso de aprendizaje secuencial; tenemos varios elementos principales a considerar en este nuestro problema, como bien serán:

- Agente
- Ambiente
- Acciones (A)
- Factor de descuento (gamma)
- Estados (S)
- Recompensas inmediatas (R)
- Estados Nuevos (S')

Sistema Determinístico

Podemos ver ahora vamos a basar este sistema de aprendizaje en castigo-recompensa y una búsqueda de por qué camino se obtienen mejores resultados, esto viene dado de forma cuantitativa por la fórmula de Campbell

$$V(S) = \max\{ R(S,A) + \text{gamma} * V(S') \}$$

El valor de un estado S como el máximo de la recompensa inmediata, R, más el valor del estado siguiente, siendo A la acción que se va a realizar. Si estamos en S y realizamos en una acción A vamos a ir al estado S', y si A es la mejor acción posible, pues sabemos S' será el mejor estado siguiente posible. Ese valor gamma o factor de descuento se trata de un número entero.

Esta recompensa inmediata R es una matriz de dimensiones $R = S \times A$ (S estados posibles en las filas y las A acciones como posibles). Realmente va a ser una matriz en la que, dependiendo de en qué estado se esté, tendremos posibilidades de avance en algunos sentidos y en otros no, y en algunos de ellos tendremos avances que resulten en: penalización / recompensa / neutro / imposible (ó no permitida).

Entonces, una vez veamos este valor de recompensa inmediata podemos ver, si asumimos la penalización como -1 y la recompensa como +1, podremos partir de la casilla de recompensa como estado final, y entonces cualquiera de los anteriores va a contar como neutro, aportando 0, de tal forma que veremos los estados a un paso de distancia, anteriores al de la recompensa serán tal que en ese mismo estado anterior deberá ser neutro, y con $\text{gamma}=0.9$ y este N-1 siendo N el paso final para alcanzar la recompensa (+1), tendremos

$$V(s(N)) = 1.0$$

$$V(S(N-1)) = \max \{ 0 + 0.9 * 1 \} = 0.9$$

Iterando el proceso para la posición anterior y viceversa

$$V(S(N-2)) = \max \{ 0 + 0.9 * 0.9 \} = 0.81$$

$$V(S(N-3)) = \max \{ 0 + 0.9 * 0.81 \} = 0.729$$

...

Y podemos ver un gradiente positivo que deberemos de seguir para llegar a la recompensa

Sistema Estocástico

En el caso determinístico, una vez elegido un avance, nos encontramos inicialmente en un estado y en el siguiente ya estamos, con un 100% de probabilidad, de estar en ese siguiente estado acorde a esa dirección en la que se ha avanzado.

En el caso estocástico, no es seguro que se haya acabado en la dirección deseada, sino que se tiene una probabilidad de avance y una probabilidad de equivocarnos e ir en cualquiera de las otras permitidas pero no idóneas.

Este escenario se define como la ecuación de Bellmann Estocástica

$$V(S) = \max \{ R(S, A) + \gamma \sum_{S'} P(S'|S, A) * V(S') \}$$

Tendremos en cuenta ahora la probabilidad condicional, podemos asumir que $P(S'|S, A) = 0.1$, y entonces ya tenemos

$$V(s(N)) = 1.0$$

$$V(S(N-1)) = \max \{ 0 + 0.9 * 0.8 * 1 \} = 0.72$$

Iterando el proceso para la posición anterior y viceversa

...

Pero en este caso, no siempre solo el gradiente sino que tendremos que tener cuidado con las posibles casillas cercanas a la penalización, ya que es posible caer.

Ahora, para resolver este problema, vamos a definir la matriz Q (Quality) para cuantificar la calidad de una acción que es dada por la recompensa inmediata (R), y simplemente podemos ver que la calidad es buena si acabamos en un estado bueno.

$$V(S) = \max \{ R(S, A) + \gamma \sum_{S'} P(S'|S, A) * V(S') \}$$

Se define la matriz calidad como

$$V(S) = \max \{ Q(S, A) \}$$

Y sustituyendo

$$Q(S,A) = R(S,A) + \text{gamma} * \text{SUMATORIO}_{\{S'\}} P(S'|S,A) * V(S')$$

Pero también sabemos que

$$V(S') = \max\{ Q(S',A) \}$$

Y vemos entonces

$$Q(S,A) = R(S,A) + \text{gamma} * \text{SUMATORIO}_{\{S'\}} P(S'|S,A) * \max\{ Q(S',A) \}$$

Podemos ver $Q(S,A)$ depende de $\max\{Q(S',A)\}$, es decir, en el estado N depende del estado N+1, o al revés, y vamos a considerar ese valor máximo de los posibles en el estado siguiente,

$$Q(S,A)(N) = Q(S,A)(N-1) + \text{alpha} * TD(A,S)$$

Es decir, vamos a ver realmente esta matriz va a ir actualizando su valor, TD va a sostenerse por Temporal Difference

$$TD(A,S) = R(S,A) + \text{gamma} * \max\{ Q(S',A) \} - Q(S,A)$$

Y finalmente

$$Q(S,A)(N) = Q(S,A)(N-1) + \text{alpha} * (R(S,A) + \text{gamma} * \max\{ Q(S',A) \} - Q(S,A))$$

In []: