

Informe - Laboratorios [8-12]

Minería de datos

Aingeru García Blas

Índice

0. Presentación	3
1. Laboratorio 8: clustering, clasificación no-supervisada: K-means	4
1.1. Sobre el Dataset Food de Hartigan Clustering Datasets	4
1.2. Naturaleza y Objetivo del Dataset	5
1.3. Descripción de los Datos	5
1.4. Importancia del Clustering	5
1.4.1. Características del Dataset	5
1.4.2. Configuración de SimpleKMeans en WEKA	5
1.5. Resultados del Clustering	6
1.5.1. Interpretación en WEKA	6
1.5.2. Análisis Visual de las Asignaciones de Cluster	6
1.6. Validación y Elección de Clusters	7
1.6.1. Configuración de 2 Clusters	7
1.6.2. Haciendo de experto; Análisis de Variables y Centroides	8
1.6.3. Detección de Outliers	8
1.7. Conclusiones	8
1.8. Comentarios sobre Scripting con R para k-means	8
1.9. Interpretación de la Gráfica	9
1.10. Clustering Jerárquico en R	9
1.11. Análisis de la demo.	9
1.11.1. Selección de los Centroides Iniciales	9
1.11.2. Tipos de Datos para Clustering	10
1.12. Otras demos de K-Means online	10
1.12.1. Demos sugeridas por otros alumnos	10
1.12.2. Otra demo encontrada	10
1.12.3. 'Genetics parallel'	11
2. Laboratorio 9: Machine Learning with R software – scripting – programando con librerías	12
2.1. Respuestas a Preguntas Específicas	14
2.2. R Output	15
3. Laboratorio 10: Redes neuronales	17
3.1. Preparación del entorno	18
3.2. Construyendo el algoritmo	21
3.3. 2.1 Funciones ayudantes	22
3.4. 2.2 Inicializar los parámetros	22
3.5. 2.3 Forward y backward pass	23
3.6. 2.4 Gradient descent	25
3.7. Construye el modelo y el clasificador	28
4. Laboratorio 11: Programando en Python 'from the scratch' – K-Nearest Neighbours	33
4.1. Preparación del entorno	34
4.2. 1. K-NN básico	36
4.3. 2. K-NN ponderado según distancia	39
4.4. 3. Distancias alternativas	42
4.5. 4. Parameter tuning	45
4.6. 5. K-NN con funciones de librería	49
4.7. EXTRA-1. Variables categóricas	50
4.8. EXTRA-2. Información Mutua	54
5. Laboratorio 12: Algoritmos Genéticos para seleccionar variables de forma 'wrapper'	

en un problema de clasificación supervisada	59
5.1. Algoritmo genético	60
5.2. Análisis del Dataset ACB-BasketballGamePrediction-2012-2013	61
5.2.1. Variables predictoras	61
5.3. Análisis de <code>WrapperSubsetEval</code> en Weka	62
5.4. Configuraciones y Resultados	63
5.4.1. Análisis de las pruebas	63
5.4.2. Óptimo Global	64
5.4.3. Otros Metaheurísticos	64
5.5. Implementación feature selection	64
5.6. Aplicaciones con Algoritmos Genéticos	65
5.6.1. Observaciones en Genetic Cars 2	65
5.6.2. Experimentos en Evolution	66
5.6.3. Curiosidad	66

0. Presentación

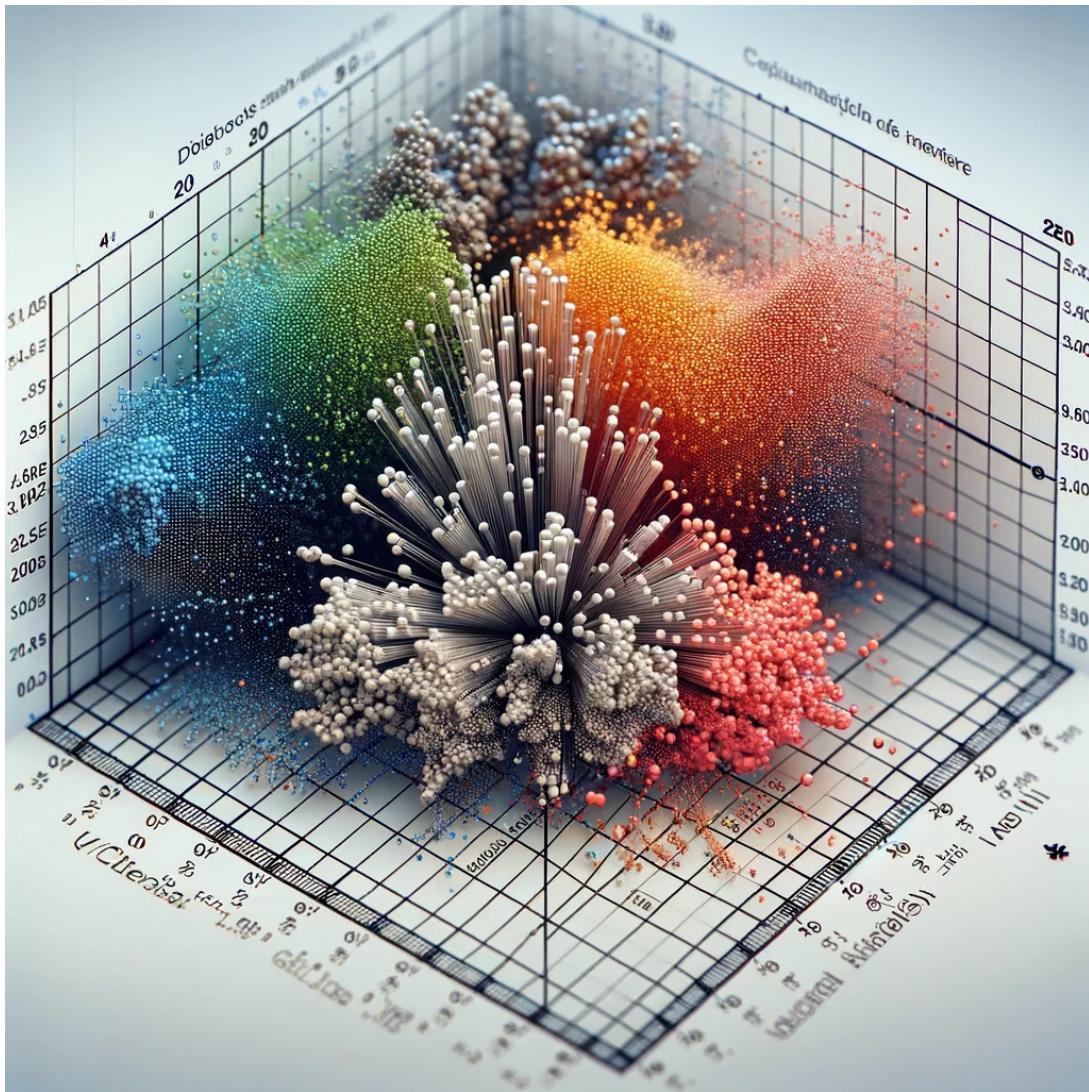


A medida que cerramos este capítulo de nuestra asignatura, nos encontramos en un momento especial, donde el final de un camino señala el comienzo de otro. Hasta ahora, hemos navegado por el fascinante cruce entre la tecnología y el conocimiento humano, con los datos como protagonistas. Este proyecto se ha centrado en cómo nos relacionamos con los algoritmos y en diseñar herramientas para comprender grandes volúmenes de datos; un viaje apasionante.

En los laboratorios anteriores, hemos explorado diferentes técnicas de Machine Learning y reflexionado sobre los aspectos éticos de trabajar con inteligencia artificial. Hemos experimentado y meditado sobre el impacto de los algoritmos en nuestra vida cotidiana, enfrentando retos prácticos en la clasificación y el procesamiento de datos. Ahora, en esta última parte del curso, nos sumergimos en temas avanzados como el clustering con K-means o el intrigante mundo de las redes neuronales y los algoritmos genéticos. Cada laboratorio ha sido diseñado no solo para enseñarnos sobre estas técnicas avanzadas, sino también para estimular nuestra curiosidad.

Aunque este pueda parecer el final de nuestro viaje, en realidad, es solo una introducción. Lo que hemos aprendido hasta ahora es tan solo la punta del iceberg y ha encendido una chispa de pasión y entusiasmo que nos impulsará a seguir explorando. Muchísimas gracias por todo.

1. Laboratorio 8: clustering, clasificación no-supervisada: K-means



1.1. Sobre el Dataset Food de Hartigan Clustering Datasets

El dataset de food, contiene datos nutricionales de distintos alimentos, se tiende a enfocar sobre todo en carnes y pescados. Cada caso o instancia detalla información sobre energía, proteínas, grasas, calcio y hierro.

También podemos encontrar este dataset en el siguiente enlace:

<https://people.sc.fsu.edu/~jburkardt/datasets/hartigan/file06.txt>

1.2. Naturaleza y Objetivo del Dataset

El propósito es clasificar alimentos según su perfil nutricional, esto se puede utilizar para muchas cosas, desde identificar patrones dietéticos, hasta soporte para estudios científicos, por ejemplo.

1.3. Descripción de los Datos

- **Nombre:** Tipo de alimento (ejemplo: *Braised Beef, Canned Tuna*).
- **Energía:** Calorías.
- **Proteínas, Grasas, Calcio, Hierro:** Contenido nutricional en gramos o miligramos.

1.4. Importancia del Clustering

Como hemos visto en clase, el clustering sirve para crear agrupaciones o clústers basados, en este caso, en similitudes nutricionales. Gracias a este método, se nos permite crear una división de las instancias o casos, de manera tanto visual como analítica.

1.4.1. Características del Dataset

Después de haber trabajado en los laboratorios de la primera parte y, como era de esperar por la naturaleza del propio dataset, lo primero que me ha llamado la atención es la ausencia de una variable de clase predefinida, lo que implica que no hay una categorización o etiqueta/clase previa para los alimentos, lo cual lo hace ideal para tareas de clustering no supervisado; de ésta manera podemos buscar patrones intrínsecos en los datos.

1.4.2. Configuración de SimpleKMeans en WEKA

Si clickamos en **SimpleKMeans** en WEKA, podemos ver varios parámetros para ajustar el proceso de clustering:

- **maxIterations:** Número máximo de iteraciones para ejecutar el algoritmo.
- **numClusters:** Número de clusters a formar.
- **distanceFunction:** Función de distancia utilizada para medir la similitud entre instancias (por defecto, distancia Euclídea).
- **displayStdDevs:** Muestra las desviaciones estándar de los atributos numéricos.

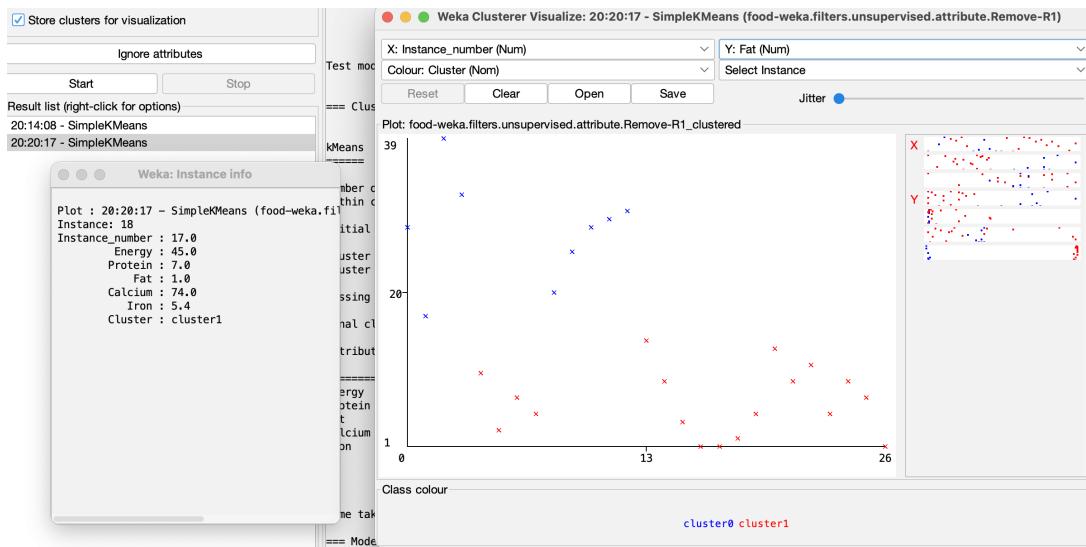
1.5. Resultados del Clustering

1.5.1. Interpretación en WEKA

Después de eliminar los atributos que no tienen sentido a la hora de hacer el clustering, veamos los elementos clave:

- **Number of iterations**: Indica cuántas veces el algoritmo ha iterado hasta converger. En este caso, han sido 2 iteraciones.
- **Initial starting points**: Representan los valores iniciales de los centroides de los clusters. Para el ejemplo, han sido seleccionados de manera totalmente aleatoria.
- **Missing values**: Muestra cómo se manejaron los valores faltantes en el dataset, que en este caso se han reemplazado de manera general con la media/moda.
- **Final cluster centroids**: El concepto clave, el centro del output. Representan el promedio de cada cluster en el espacio de características. Por ejemplo, el Cluster 0 tiene un promedio de 331.1111 en Energía, 19 en Proteínas, etc. Estos valores sirven para interpretar las características dominantes de cada agrupación.
- **Distribución de Instancias**: Muestra la cantidad y el porcentaje de instancias o casos en cada cluster, esto permite apreciar cómo se distribuyen los datos entre los grupos formados.

1.5.2. Análisis Visual de las Asignaciones de Cluster



La gráfica de asignaciones de cluster muestra cada caso del dataset como un punto en un espacio bidimensional; el eje X representa el número de instancia y el eje Y uno de los atributos. Los puntos tienen un color que corresponde al cluster al que han sido asignados, con lo que podemos observar una posible distribución de los datos.

- **Distribución de Datos:** La disposición de los puntos nos puede dar indicios sobre la densidad y la separación de los clusters, así como de la variabilidad dentro de cada cluster, esto es bastante interesante porque, ya no sólo obtenemos las agrupaciones en sí, si no cuán densos pueden ser sus agrupamientos internos, lo cual implicaría una mayor similitud entre sus componentes.
- **Regularidades y Patrones:** Aunque la gráfica muestra solo dos variables, si estas son significativas, pueden revelar patrones, outliers etc.
- **Colores:** Los colores diferencian los clusters. De ésta manera podemos observar cuán similares son los casos, nutricionalmente hablando.
- **Interacción:** Al hacer clic en una instancia, se puede ver información de la misma, lo que ayuda a entender y tener una perspectiva más global de ese caso en concreto.
- **Verificación de la Asignación de Cluster:** Guardando los resultados como un archivo .arff y abriéndolos en WEKA, se puede verificar la asignación de cluster de cada instancia y realizar un análisis más profundo.

1.6. Validación y Elección de Clusters

Hemos visto en clase que la validación de la calidad de un clustering es algo complejo que normalmente requiere un enfoque cualitativo. Ya comentaba el profesor el hecho de que, en muchas ocasiones era necesario recurrir a expertos para clasificar o determinar la calidad de ciertos parámetros de los datasets. Aquí se puede apreciar que efectivamente, el clustering es una técnica que requiere de *visto bueno*.

Después de realizar varias pruebas con diferentes números de clusters:

1.6.1. Configuración de 2 Clusters

- Los dos clusters identificados reflejan dos patrones dietéticos distintos: alimentos ricos en proteínas y bajos en grasas (**Cluster 1**) y alimentos con mayor contenido energético y de grasas (**Cluster 0**).
- Los centroides de cada cluster muestran diferencias curiosas en las variables de energía y grasas, lo que puede llevar a pensar que estos son los factores distintivos en el dataset, aunque claramente están correlacionadas.
- La cohesión dentro de los mismos es alta, y la separación entre ellos es clara, mientras que en algunos *runs* esto no estaba tan matizado. Lo cual se representa en los valores medios de las variables mencionadas, que se alejan de la media general del dataset.

1.6.2. Haciendo de experto; Análisis de Variables y Centroides

El Cluster 0, con altos valores en energía y grasas, podría etiquetarse como **Alimentos Energéticos** (o alimentos ricos), mientras que el Cluster 1, con valores más bajos en estas variables, se etiquetaría como **Alimentos Ligeros** (o alimentos, mucho menos ricos).

1.6.3. Detección de Outliers

Algunos casos, están relativamente alejados de los centroides de sus clusters, con lo que podrían considerarse outliers un poco tímidos. No obstante, para determinar si efectivamente son outliers, un experto en el tema/contexto debería de analizar los casos en detalle.

1.7. Conclusiones

Al fin y al cabo, el análisis que hemos hecho ha sido de tipo cualitativo - es decir, nos basamos en la interpretación de los centroides y la distribución de las instancias - de tal manera que podemos ver las cosas desde una perspectiva a más alto nivel sobre la estructura nutricional del dataset. Como se ha mencionado, cabe destacar que, como en todos los métodos de clasificación, el clustering se beneficia también de una validación apropiada y, en estos casos, no es algo sencillo ya que dependeremos en gran medida de los ojos de un o una experto/a.

1.8. Comentarios sobre Scripting con R para k-means

```
# Carga el dataset food (en un servidor remoto) y lo carga en el dataframe food
food <- read.csv(file='http://.../food.csv', header=TRUE, sep=',', ')

# Muestra los nombres de las columnas
colnames(food)

# Crea un nuevo dataframe foodNumeric quitando la primera columna
# (como hemos hecho nosotros en Weka)
foodNumeric <- food[,-1]

# Carga la librería stats, para poder utilizar k-means.
library(stats)

# Ejecuta y guarda el resultado de el algoritmo k-means en los
# datos numéricos del dataframe con 2 centroides/clusters
kmeans.res <- kmeans(foodNumeric, centers=2)

# Imprime los resultados anteriores
print(kmeans.res)

# Muestra la asignación de cada instancia a un cluster.
kmeans.res$cluster

# Añade la columna de asignaciones anterior al dataframe original
foodWithCluster <- cbind(food, cluster=kmeans.res$cluster)
# Muestra el resultado
```

```

foodWithCluster

# Instala el paquete factoextra para visualización avanzada de clustering.
install.packages('factoextra')
# Carga dicho paquete
library(factoextra)

# Visualiza el resultado del clustering k-means
fviz_cluster(kmeans.res, data=foodNumeric)

```

1.9. Interpretación de la Gráfica

Los ejes de la gráfica representan las dos primeras componentes principales del PCA realizado a los datos, que sirven para visualizar los clusters en dos dimensiones. Los porcentajes indican la varianza para cada componente; esto nos permite tener una idea de cuánta información de los datos originales se conserva en la visualización.

1.10. Clustering Jerárquico en R

```

# Normaliza los datos
range01 <- function(x){(x-min(x))/(max(x)-min(x))}
foodNormalized <- range01(foodNumeric)

# Calcula la distancia euclidiana
distances <- dist(foodNormalized)

# Realiza clustering jerárquico con el método de enlace completo
foodHClust <- hclust(distances, method='complete')

# Crea y muestra la gráfica correspondiente.
plot(foodHClust, labels=food$Name, main='Clustering jerárquico - complete linkage')

```

Aquí se normalizan los datos para que todas las variables contribuyan equitativamente al análisis. Luego, se calculan las distancias euclidianas entre las instancias normalizadas y se realiza un clustering jerárquico usando el método de enlace completo. Se hace el plot del dendrograma con las etiquetas de los alimentos, lo que hace que se pueda observar visualmente cómo se agrupan los alimentos en base a su similitud.

Respecto al uso de **single linkage** en lugar de **complete linkage**; radica en cómo se calcula la distancia entre los clusters en el clustering jerárquico. El **single linkage** toma la **distancia mínima** entre los elementos de dos clusters, mientras que el **complete linkage** utiliza la **distancia máxima**. Así que si, esto puede resultar en dendrogramas muy diferentes.

1.11. Análisis de la demo.

1.11.1. Selección de los Centroides Iniciales

La opción que en una primera instancia tenemos que seleccionar, de *How to pick the initial centroids*, hace referencia a la estrategia para elegir los puntos iniciales que actuarán como centroides de los clusters. Esta elección es determinante, porque puede influir en cómo converge del algoritmo y en la configuración final de los clusters. Las tres opciones ofrecidas son:

- **I'll choose:** Nos permite seleccionar manualmente los centroides iniciales. Esto puede ser útil si tenemos información previa sobre la distribución de los datos.
- **Randomly:** Los centroides son elegidos de manera aleatoria. Nótese que puede llevar a resultados diferentes en cada ejecución.
- **Farthest point:** Selecciona un punto aleatorio como primer centroide y luego elige los siguientes basándose en su distancia al centroide a dicho centroide. Al parecer, esto tiende a mejorar la dispersión inicial de los centroides.

1.11.2. Tipos de Datos para Clustering

La opción '*What kind of data would you like?*' nos permite seleccionar la naturaleza del conjunto de datos sobre el que se ejecutará el k-means. Las diferentes opciones simulan escenarios de datos reales y afectan cómo se agruparán los datos. Por ejemplo:

- **Uniform Points:** Distribuye los puntos de manera uniforme en el espacio. Sirve para ver cómo el algoritmo se comporta en condiciones de ausencia de clusters naturales.
- **Gaussian Mixture:** Genera datos que siguen una mezcla de distribuciones gaussianas.
- **Smiley Face:** Una distribución de puntos con forma de cara sonriente, al parecer se suele utilizar para ver cómo se comporta con patrones y distribuciones raras o no usuales.

Notas personales

Bueno, ha estado bastante bien el poder '*jugar*' y poder observar de una manera sencilla cómo se crean los clúster y clasifican las instancias. Es decir, la posibilidad de interactuar con los datos y los parámetros del algoritmo, es algo muy útil para poder comprender bien e incluso de manera visual su funcionamiento. Es particularmente interesante el ver cómo la selección de centroides iniciales y el tipo de datos influyen en los resultados finales.

1.12. Otras demos de K-Means online

1.12.1. Demos sugeridas por otros alumnos

De las demos sugeridas en el enunciado, la que me ha gustado ha sido <http://alekseynp.com/viz/k-means.html>. Me ha parecido una manera muy intuitiva de manipular los datos e iterar sobre ellos. Además la visualización está bastante bien y se puede apreciar claramente el funcionamiento del algoritmo.

1.12.2. Otra demo encontrada

Por sugerir algo un poco diferente, me gustaría mencionar la famosa herramienta matemática online **Desmos**. Sirve para una infinidad de cosas y, por supuesto, también para K-means. Si vamos al siguiente enlace, podemos configurar los clúster manualmente y visualizarlos:

<https://www.desmos.com/calculator/pb4ewmqdvy>

1.12.3. 'Genetics parallel'

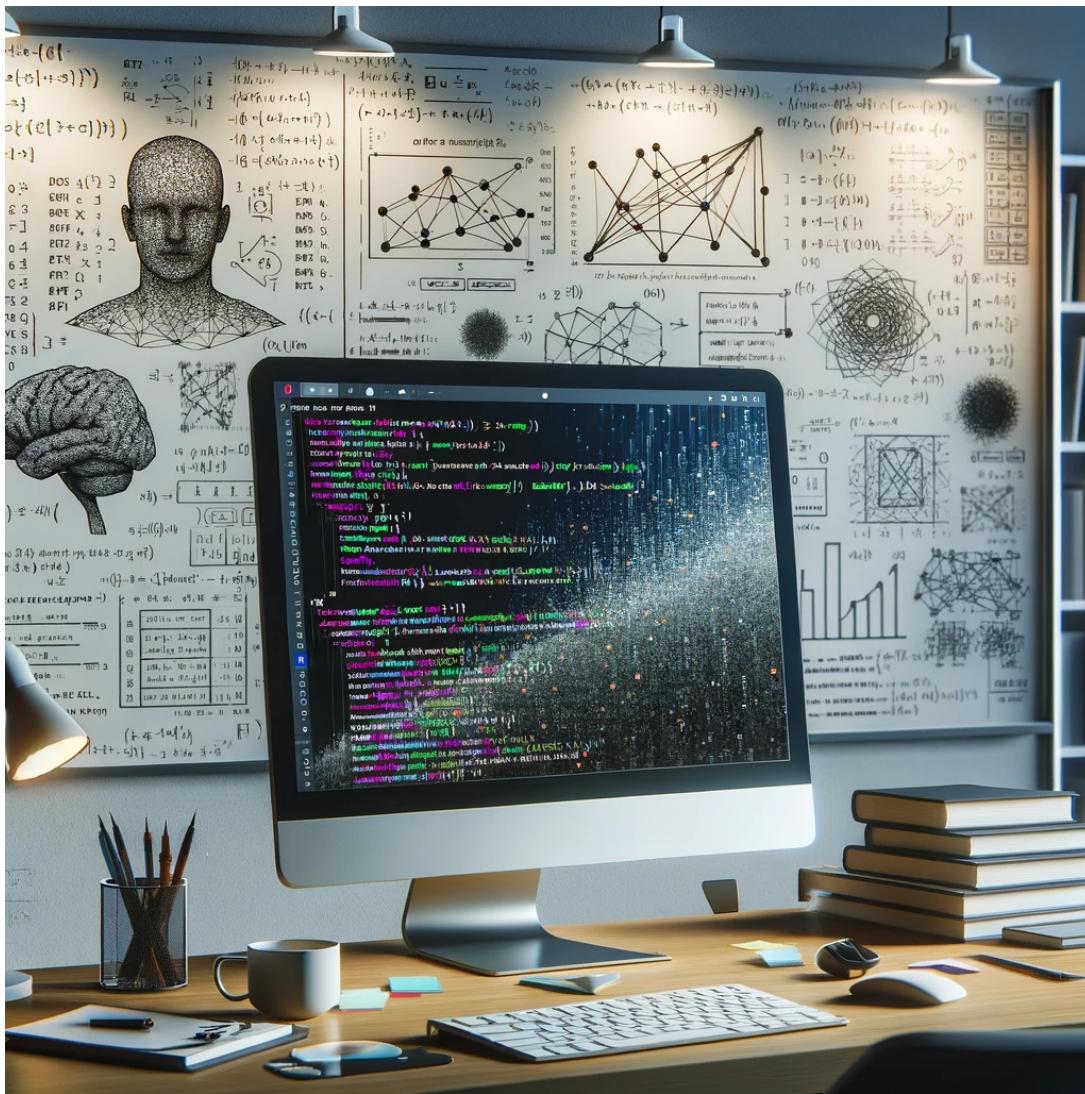
No es una demo en sí, pero me gustaría compartir un proyecto que realizamos el año pasado, en la asignatura de Arquitectura de Computadores. Aunque el proyecto tenía como objetivo principal el trabajar la paralelización de aplicaciones y para ello, clasificábamos distintas instancias en base a la similitud entre sus características genéticas (distancia euclídea), de grandes bases de datos de la **OMS**, utilizábamos el algoritmo de K-means clustering, del cual, unas cuantas partes tuvimos que implementar nosotros:

Proyecto en mi GitHub: <https://github.com/geru-scotland/Genetics-parallel>

El informe/memoria del proyecto se puede encontrar en el repositorio de informes relacionados a mi nuevo viaje universitario:

<https://github.com/geru-scotland/informes-memorias>

2. Laboratorio 9: Machine Learning with R software – scripting – programando con librerías



Comencemos describiendo partes del script:

```
# Establece el directorio de trabajo donde se encuentra el archivo CSV
setwd('/home/geru')

# Carga el dataset iris en formato CSV.
# header = TRUE indica que la primera línea contiene los nombres de las columnas.
# sep = ',' especifica que el separador de campos es una coma.
iris <- read.csv('iris.csv', header = TRUE, sep = ',')

# Resumen estadístico del dataset, incluyendo media, mediana, mínimo, máximo, etc.
summary(iris)
```

```

# Mostrar los nombres de las columnas del dataset.
names(iris)

# Mostar las primeras filas del dataset para obtener una vista preliminar de los datos.
head(iris)

# Convierte la columna variety a un factor para análisis categórico y muestra la tabla.
iris$variety <- as.factor(iris$variety)
table(iris$variety)

# Instalación y carga de la librería caret, usada para entrenamiento de modelos de clasificación
y regresión.
# install.packages('caret', dependencies = T)
library(caret)

# Establece una semilla para asegurar la reproducibilidad de los resultados al generar
números aleatorios.
set.seed('1234567890')

# Utiliza createDataPartition para crear índices para un conjunto de entrenamiento basado
en la columna variety.
# p = .66 indica que el 66% de los datos se usan para training
# list = FALSE hace que la función devuelva un vector de índices en lugar de una lista.
trainSetIndexes <- createDataPartition(y = iris$variety, p = .66, list = FALSE)

# Creación de los conjuntos de entrenamiento y prueba usando los índices generados.
trainSet <- iris[trainSetIndexes,]
testSet <- iris[-trainSetIndexes,]

# Muestra el número de filas en cada conjunto, para entender la distribución de los datos.
nrow(trainSet)
nrow(testSet)

# Configuración del control de entrenamiento con validación cruzada de 10 pliegues.
ctrl <- trainControl('cv', number = 10)

# Entrenamiento de un modelo KNN.
# variety ~ . indica que variety es la variable dependiente y el punto representa todas las demás
columnas como predictoras.
# method = 'knn' especifica que el método de training es k-nearest neighbors.
# tuneLength = 5 indica que se probarán con k = 5
# trControl = ctrl aplica la configuración de control de entrenamiento previamente definida.
# preProc = c('scale') escalado de las variables predictoras para normalizar
KNNModel1 <- train(variety ~ ., data = trainSet, method = 'knn', tuneLength = 5, trControl = ctrl,
preProc = c('scale'))

# Visualización del modelo KNN entrenado
KNNModel1
plot(KNNModel1)

# Uso del modelo para realizar predicciones en el conjunto de test
KNNPredict1 <- predict(KNNModel1, newdata = testSet)

# Generación de una matriz de confusión

```

```

confusionMatrix(KNNPredict1, testSet$variety)

# Para Naive Bayes y Trees
install.packages("e1071")
library(e1071)

# Entrenar el model Naive Bayes
nbModel <- naiveBayes(variety ~ ., data=trainSet)
# Usar el modelo para las predicciones
nbPredictions <- predict(nbModel, testSet)

# Evaluarlo
confusionMatrix(nbPredictions, testSet$variety)

install.packages("rpart")
library(rpart)

# Settings para el training
trainControl <- trainControl(method="none") # Sin validación cruzada

# Training
treeModel <- train(variety ~ ., data=trainSet, method="rpart", trControl=trainControl)

# Usar el modelo para las predicción
treePredictions <- predict(treeModel, testSet)

# Evaluar el modelo
confusionMatrix(treePredictions, testSet$variety)

```

2.1. Respuestas a Preguntas Específicas

set.seed('1234567890'): Esta función se usa para establecer una semilla para la generación de números aleatorios, esto hace que los resultados sean reproducibles.

tuneLength en train(): Especifica el número de valores diferentes de tuning (como el número de vecinos en KNN) que se probarán en el modelo. Afecta a cómo el modelo ajusta sus parámetros internos para optimizar su rendimiento.

tuneGrid: permite especificar una cuadrícula o grid de valores de parámetros para el *tuning* del modelo. Esto da un control más fino sobre el proceso de ajuste del modelo, permitiendo especificar exactamente qué valores se deben probar.

¿Qué implica 'escalar' una variable numérica, tal y como hemos pedido en las opciones de preprocesso?

Escalar una variable numérica significa ajustar su rango o distribución, se suele hacer para mejorar el rendimiento y es algo que hemos ya hemos ido haciendo bastantes veces, por ejemplo normalizando ciertas variables en el preprocessamiento - también es algo básico en gráficos por computador, por ejemplo, donde se normalizan vectores constantemente.

¿qué diferencia principal hay entre el de KNNModel1 y KNNModel2? ¿Qué parámetro se ha tuneado en el proceso de aprendizaje? ¿Con qué valor definitivo se ha quedado, cada modelo? La diferencia principal ha sido el cómo se ha tuneado el hyperparámetro `k`, en modelo 1, se ha utilizado el `tuneLength`, mientras que en el modelo 2, ya que se utiliza `tunegrid` `tuneGrid = expand.grid(k = c(1, 3, 5, 15, 19))`

Para el modelo 1, el `k` final es 9 y para el 2, el `k` final es el 3. Aunque he ejecutado varias veces y esto ha ido cambiando, para ambos.

Veamos el output:

2.2. R Output

```
> summary(iris)
```

sepal.length	sepal.width	petal.length	petal.width	variety
Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100	Length:150
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300	Class :character
Median :5.800	Median :3.000	Median :4.350	Median :1.300	Mode :character
Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199	
3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800	
Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500	

```
> names(iris)
```

Names
sepal.length
sepal.width
petal.length
petal.width
variety

```
> head(iris)
```

sepal.length	sepal.width	petal.length	petal.width	variety
5.1	3.5	1.4	0.2	Setosa
4.9	3.0	1.4	0.2	Setosa
4.7	3.2	1.3	0.2	Setosa
4.6	3.1	1.5	0.2	Setosa
5.0	3.6	1.4	0.2	Setosa
5.4	3.9	1.7	0.4	Setosa

```
> iris$variety <- as.factor(iris$variety)
```

```
> table(iris$variety)
```

Variety	Count
Setosa	50
Versicolor	50
Virginica	50

```
> library(caret)
```

```
> set.seed("1234567890")
```

```
> trainSetIndexes <- createDataPartition(y=iris$variety,p=.66,list=FALSE)
```

```
> trainSet <- iris[trainSetIndexes,]
```

```
> testSet <- iris[-trainSetIndexes,]
```

```
> nrow(trainSet)
```

```
[1] 99
> nrow(testSet)
[1] 51

> ctrl <- trainControl("cv", number=10)

> KNNModel1 <- train(variety ~ ., data=trainSet, method="knn", tuneLength=5, trControl=ctrl,
preProc=c("scale"))
> KNNModel1



| k  | Accuracy  | Kappa     |
|----|-----------|-----------|
| 5  | 0.9386869 | 0.9073918 |
| 7  | 0.9509091 | 0.9255736 |
| 9  | 0.9609091 | 0.9407251 |
| 11 | 0.9497980 | 0.9240585 |
| 13 | 0.9275758 | 0.8907251 |



> KNNModel1
Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 9.

> confusionMatrix(KNNPredict1, testSet$variety)



| Prediction/Reference | Setosa | Versicolor | Virginica |
|----------------------|--------|------------|-----------|
| Setosa               | 17     | 0          | 0         |
| Versicolor           | 0      | 17         | 1         |
| Virginica            | 0      | 0          | 16        |



Overall Statistics
Accuracy : 0.9804
95% CI : (0.8955, 0.9995)
No Information Rate : 0.3333
P-Value [Acc > NIR] : < 2.2e-16
Kappa : 0.9706
McNemar's Test P-Value : NA

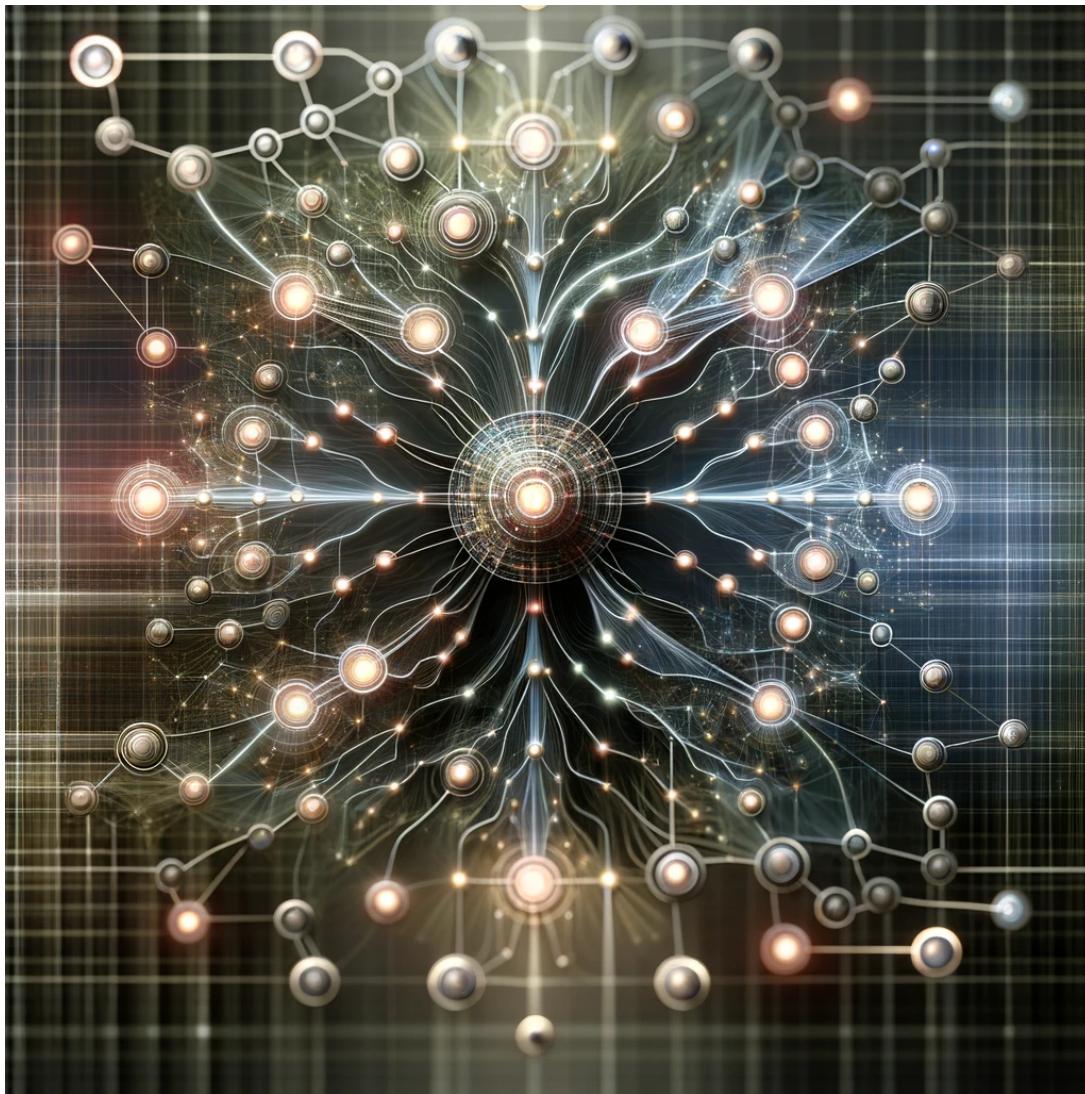


| Statistics           | Class: Setosa | Class: Versicolor | Class: Virginica |
|----------------------|---------------|-------------------|------------------|
| Sensitivity          | 1.0000        | 1.0000            | 0.9412           |
| Specificity          | 1.0000        | 0.9706            | 1.0000           |
| Pos Pred Value       | 1.0000        | 0.9444            | 1.0000           |
| Neg Pred Value       | 1.0000        | 1.0000            | 0.9714           |
| Prevalence           | 0.3333        | 0.3333            | 0.3333           |
| Detection Rate       | 0.3333        | 0.3333            | 0.3137           |
| Detection Prevalence | 0.3333        | 0.3529            | 0.3137           |
| Balanced Accuracy    | 1.0000        | 0.9853            | 0.9706           |



> KNNModel2
k-Nearest Neighbors
99 samples
4 predictor
3 classes: 'Setosa', 'Versicolor', 'Virginica'
Pre-processing: scaled (4)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 89, 90, 89, 87, 90, 89, ...
```

3. Laboratorio 10: Redes neuronales



Para éste laboratorio se nos ha asignado la tarea de completar en Collab. El laboratorio está muy bien preparado y guiado, gracias por el trabajo.

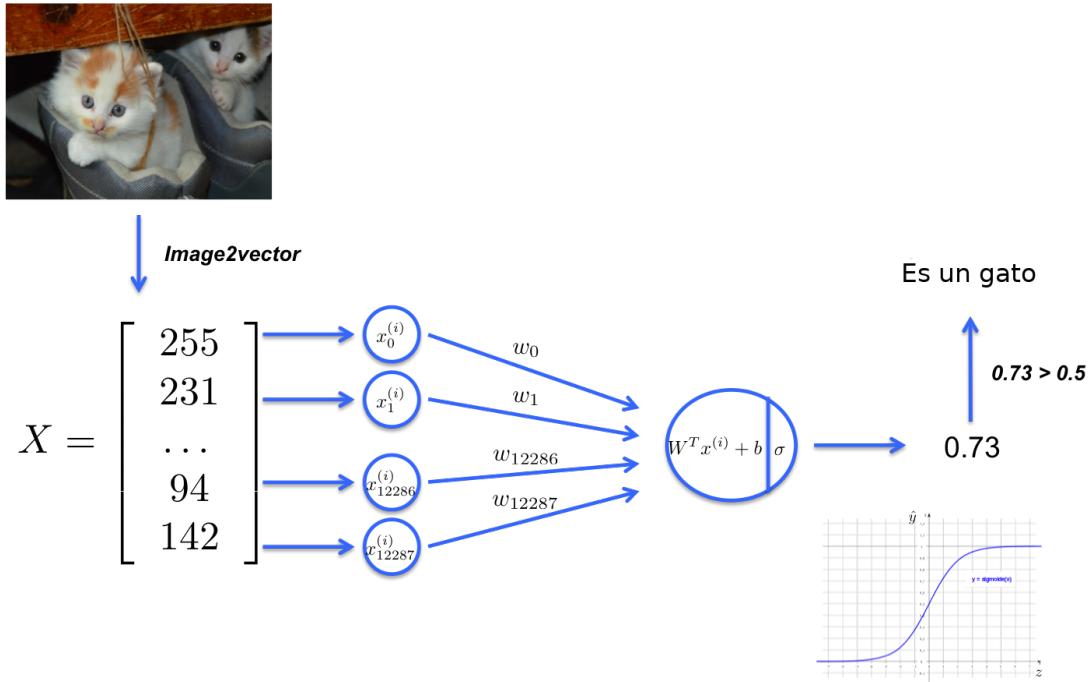
Bien, veamos el laboratorio ya completado de Collab, lo expongo aqui enteramente:

```
# Código original
import numpy as np

# Código añadido por mí (en negrita)
textbf{import pandas as pd}

# Código modificado (con comentario)
np.array([1, 2, 3]) # Modificado para demostración
```

En el ejercicio anterior cargamos los datos de unos ficheros con imágenes de gatos. En este segundo ejercicio, aprenderemos un modelo muy simple de regresión logística. Recuerda que ese modelo se puede considerar una versión muy simple de una red neuronal. A continuación, el diagrama que describe el modelo que vamos a implementar:



La expresión matemática del algoritmo

Para una muestra de datos $x^{(i)}$:

$$z^{(i)} = W^T x^{(i)} + b$$

$$\hat{y} = a^{(i)} = \sigma(z^{(i)})$$

$$(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

La función de coste se consigue calculando sobre todas las muestras de train:

$$J = \frac{1}{m} \sum_{i=1}^m (a^{(i)}, y^{(i)})$$

Estas son las tareas principales a realizar en este ejercicio:

- Inicializar los parámetros del modelo.
- Aprender los parámetros óptimos del modelo minimizando la función de coste.
- Clasificar las muestras de test con los parámetros aprendidos.
- Analizar los resultados y extraer conclusiones.

3.1. Preparación del entorno

Para comenzar el ejercicio, tenemos que importar las librerías, cargar los datasets y estandarizar los datos. Ya hicimos eso en el ejercicio anterior, así que nos limitaremos a pegar el código y a ejecutarlo.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
```



```
[2]: !wget -O train_catvnoncat.h5 https://ehubox.ehu.eus/s/62gBFyzGpDD7rei/download
!wget -O test_catvnoncat.h5 https://ehubox.ehu.eus/s/eBa6kaBjyK3nwSf/download
!ls
```

--2023-12-18 17:25:29-- https://ehubox.ehu.eus/s/62gBFyzGpDD7rei/download
Resolving ehubox.ehu.eus (ehubox.ehu.eus)... 158.227.0.95
Connecting to ehubox.ehu.eus (ehubox.ehu.eus)|158.227.0.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2572022 (2.5M) [application/octet-stream]
Saving to: 'train_catvnoncat.h5'

train_catvnoncat.h5 100%[=====] 2.45M 956KB/s in 2.6s

2023-12-18 17:25:32 (956 KB/s) - 'train_catvnoncat.h5' saved [2572022/2572022]

--2023-12-18 17:25:32-- https://ehubox.ehu.eus/s/eBa6kaBjyK3nwSf/download
Resolving ehubox.ehu.eus (ehubox.ehu.eus)... 158.227.0.95
Connecting to ehubox.ehu.eus (ehubox.ehu.eus)|158.227.0.95|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 616958 (602K) [application/octet-stream]
Saving to: 'test_catvnoncat.h5'

test_catvnoncat.h5 100%[=====] 602.50K 413KB/s in 1.5s

2023-12-18 17:25:35 (413 KB/s) - 'test_catvnoncat.h5' saved [616958/616958]

sample_data test_catvnoncat.h5 train_catvnoncat.h5


```
[3]: def load_dataset():
    train_dataset = h5py.File('train_catvnoncat.h5', 'r')
    train_set_x_orig = np.array(train_dataset['train_set_x'][:]) # your train set
    ↵ features
    train_set_y_orig = np.array(train_dataset['train_set_y'][:]) # your train set
    ↵ labels

    test_dataset = h5py.File('test_catvnoncat.h5', 'r')
    test_set_x_orig = np.array(test_dataset['test_set_x'][:]) # your test set features
    test_set_y_orig = np.array(test_dataset['test_set_y'][:]) # your test set labels

    classes = np.array(test_dataset['list_classes'])[:] # the list of classes

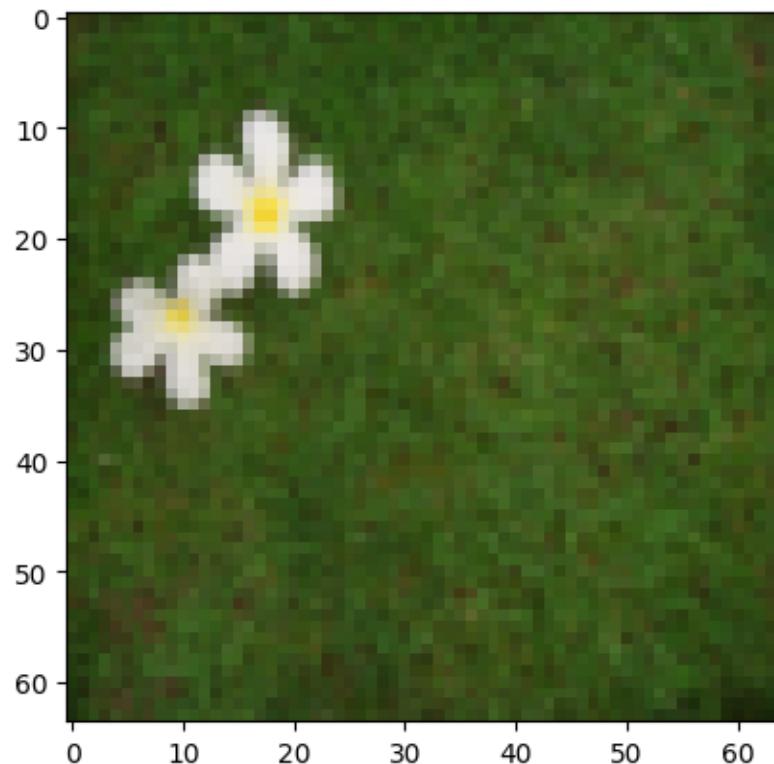
    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig,
    ↵ classes
```

```
[4]: # Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()

# Example of a picture
index = 6
plt.imshow(train_set_x_orig[index])
print ('y = ' + str(train_set_y[:, index]) + ', it\'s a \'' + classes[np.
    -squeeze(train_set_y[:, index])].decode('utf-8') + '\' picture.')
```

y = [0], it's a 'non-cat' picture.



```
[5]: m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px_height = train_set_x_orig.shape[1]
num_px_width = train_set_x_orig.shape[2]
num_px = num_px_width

print ('Cantidad de muestras de train: m_train = ' + str(m_train))
print ('Cantidad de muestras de test: m_test = ' + str(m_test))
print ('Altura de las imágenes: num_px_height = ' + str(num_px_height))
print ('Anchura de las imágenes: num_px_width = ' + str(num_px_width))
print ('Tamaño de cada imagen: (' + str(num_px_height) + ', ' + str(num_px_width) + '
    ↵, 3)')
print ('Estructura de las muestras de train X: ' + str(train_set_x_orig.shape))
print ('Estructura de las etiquetas de train Y: ' + str(train_set_y.shape))
print ('Estructura de las muestras de test X: ' + str(test_set_x_orig.shape))
print ('Estructura de las etiquetas de test Y: ' + str(test_set_y.shape))

# Vamos a reestructurar los datos, convirtiendo el cubo que forma una imagen
# ↵[height, width, channels] a un vector de height*width*channel elementos
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

print ('X_train.shape: ' + str(train_set_x_flatten.shape))
print ('Y_train.shape: ' + str(train_set_y.shape))
print ('X_test.shape: ' + str(test_set_x_flatten.shape))
print ('Y_test.shape: ' + str(test_set_y.shape))

print ('Verifica que los datos se han reestructurado correctamente: ' +
    ↵str(train_set_x_flatten[0:5,0]))
```

Para estandarizar los datos, dividiremos el valor de cada pixel por el valor
 ↵maximo (255 en este caso).

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

Cantidad de muestras de train: m_train = 209
 Cantidad de muestras de test: m_test = 50
 Altura de las imágenes: num_px_height = 64
 Anchura de las imágenes: num_px_width = 64
 Tamaño de cada imagen: (64, 64, 3)
 Estructura de las muestras de train X: (209, 64, 64, 3)
 Estructura de las etiquetas de train Y: (1, 209)
 Estructura de las muestras de test X: (50, 64, 64, 3)
 Estructura de las etiquetas de test Y: (1, 50)
 X_train.shape: (12288, 209)
 Y_train.shape: (1, 209)
 X_test.shape: (12288, 50)
 Y_test.shape: (1, 50)
 Verifica que los datos se han reestructurado correctamente: [17 31 56 22 33]

3.2. Construyendo el algoritmo

Para entrenar una red neuronal, tenemos que implementar las siguientes líneas:

1. Definir la estructura del modelo (¿cuántas variables de entrada?).
2. Inicializar los parámetros del modelo.
3. Loop:
 - Calcular el valor de la función de coste (forward pass).
 - Calcular el valor de los gradientes (backward pass).
 - Actualizar los parámetros (gradient descent).

A continuación, vamos a implementar cada parte de forma separada para juntar todas al final, implementando una red neuronal.

3.3. 2.1 Funciones ayudantes

Implementa la función sigmoid() (σ). Más adelante la necesitaremos, para calcular las predicciones $\sigma(W^T x + b)$. Recuerda que $\sigma(z) = \frac{1}{1+e^{-z}}$

```
[6]: # sigmoid funtzioa

def sigmoid(z):
    """
    Calcula el sigmoide de z

    Input:
    z -- Un número real o un numpy-array de números reales.

    Output:
    s -- sigmoid(z)
    """

    # Recuerda, numpy hace broadcasting
    s = 1 / (1 + np.exp(-z))

    #####
    return s

print ('sigmoid([0, 2]) = ' + str(sigmoid(np.array([0,2]))))
```

sigmoid([0, 2]) = [0.5 0.88079708]

3.4. 2.2 Inicializar los parámetros

Ejercicio: Tienes que inicializar los parámetros en el siguiente código. Para eso, crea el array w con ceros. **Pista:** mira la función np.zeros().

```
[7]: def initialize_with_zeros(dim):
    """
    Esta funcion crea un vector de ceros de dimensiones (dim, 1), para inicializar w
    →y b.

    Argument:
    dim -- tamaño del vector w.

    Returns:
    w -- vector w de ceros con dimension (dim, 1).
    b -- parametro b inicializado a 0.
    """

    w = np.zeros((dim, 1))
    b = 0

    #####
    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b

dim = 2
w, b = initialize_with_zeros(dim)
print ('w = ' + str(w))
print ('b = ' + str(b))
```

```
w = [[0.]
      [0.]]
b = 0
```

Gure datuak irudiak direnean, w ($\text{num_px} \times \text{num_px} \times 3, 1$) dimentsioko bektorea izango da.

3.5. 2.3 Forward y backward pass

Una vez inicializados los parámetros del modelo, dado un dataset de train, tenemos que implementar las funciones forward y backward que nos permitan aprender los parámetros.

Ejercicio: Implementa la función de coste y su gradiente en la función propagate().

Pistas:

Forward pass:

- Consigue X .
- Calcula $A = \sigma(W^T X + b) = (a^{(1)}, a^{(2)}, \dots, a^{(m)})$
- Calcula la función de coste $J = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Backward pass:

- $\frac{\delta J}{\delta w} = \frac{1}{m} X(A - Y)^T$
- $\frac{\delta J}{\delta b} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)})$

```
[8]: def propagate(w, b, X, Y):
    """
    Calcula la funcion de coste y su gradiente ejecutando el forward y el backward pass.

    Input:
    w -- pesos, numpy-array de tamaño (num_px * num_px * 3, 1).
    b -- parametro b (bias).
    X -- Conjunto de muestras X de train. Dimension: (num_px * num_px * 3, number of examples)
    Y -- Conjunto de etiquetas Y de train (0 -> no gato, 1 -> gato). Dimension: (1, number of examples)

    Output:
    cost -- Coste de la regresion logistica.
    dw -- gradiente de la funcion de coste respecto a w.
    db -- gradiente de la funcion de coste respecto a b.
    """

    Pista:
    - Escribe tu codigo paso a paso para la propagacion. np.log(), np.dot()
    ,,
```

```
m = X.shape[1]

# FORWARD PASS (empezando desde X hasta calcular el coste)
A = sigmoid(np.dot(w.T, X) + b)
cost = (-1/m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))

#####
# BACKWARD PASS (calcula el gradiente)
dw = (1/m) * np.dot(X, (A - Y).T)
db = (1/m) * np.sum(A - Y)

#####
assert(dw.shape == w.shape)
assert(db.dtype == float)
cost = np.squeeze(cost)
assert(cost.shape == ())

grads = {'dw': dw,
         'db': db}

return grads, cost
```

```
w, b, X, Y = np.array([[1.], [2.]]), 2., np.array([[1., 2., -1.], [3., 4., -3.2]]), np.array([[1, 0, 1]])
grads, cost = propagate(w, b, X, Y)
print ('dw = ' + str(grads['dw']))
print ('db = ' + str(grads['db']))
print ('cost = ' + str(cost))
```

```
dw = [[0.99845601]
      [2.39507239]]
db = 0.001455578136784208
cost = 5.801545319394553
```

3.6. 2.4 Gradient descent

En este punto ya:

- has inicializado los parámetros,
- eres capaz de calcular la función de coste y el gradiente,
- vas a actualizar los parámetros usando el algoritmo “gradient descent” (descenso de gradiente)

Ejercicio: implementa el algoritmo de descenso de gradiente en la función `optimize()`. Recuerda que el objetivo es aprender los parámetros w y b que minimicen la función J . Para cualquier parámetro θ , éste se actualiza segun la regla $\theta = \theta - \alpha\delta\theta$, donde α es la tasa de aprendizaje (learning rate).

```
[9]: def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    Esta función optimiza los parámetros w y b para minimizar la función de coste
    →siguiendo
    el algoritmo de gradient descent.

    Input:
    w -- pesos, numpy-array de tamaño (num_px * num_px * 3, 1).
    b -- parámetro b (bias).
    X -- Conjunto de muestras X de train. Dimension: (num_px * num_px * 3, number of
    →examples)
    Y -- Conjunto de etiquetas Y de train (0 -> no gato, 1 -> gato). Dimension: (1,
    →number of examples)
    num_iterations -- numero de iteraciones del algoritmo.
    learning_rate -- tasa de aprendizaje de la regla de actualización.
    print_cost -- cuando es True, imprime el valor del gradiente cada 100
    →iteraciones.

    Output:
    params -- diccionario que guarda los parametros w y b.
    grads -- diccionario que guarda los gradientes de los parametros w y b respecto
    →a la funcion de coste.
    costs -- lista donde se guardan los valores de coste. Los usaremos para generar
    →una grafica.

    Pistas:
    Tienes que implementar dos pasos e iterar sobre ellos:
        1) Calcular el coste y el gradiente para los parametros actuales. Usa
    →propagate()
        2) Actualizar los parametros w y b utilizando la regla de actualizacion del
    →descenso de gradiente .
    """

    costs = []
    params = {'w':w,
              'b':b}
    m = X.shape[1]

    for i in range(num_iterations):

        # Calculo del coste y del gradiente ( 1 linea de codigo)
        grads, cost = propagate(w, b, X, Y)

        #####
        # Retrieve derivatives from grads
        dw = grads['dw']
        db = grads['db']

        # Regla de actualización
        w = w - learning_rate * dw
        b = b - learning_rate * db
```

[9] :

```
#####
# Guarda los costes.
if i % 100 == 0:
    costs.append(cost)

# Imprime el coste cada 100 iteraciones.
if print_cost and i % 100 == 0:
    print ('Cost after iteration %i: %f' %(i, cost))

params = {'w': w,
          'b': b}

grads = {'dw': dw,
          'db': db}

return params, grads, costs

params, grads, costs = optimize(w, b, X, Y, num_iterations= 100, learning_rate = 0.
→009, print_cost = False)

print ('w = ' + str(params['w']))
print ('b = ' + str(params['b']))
print ('dw = ' + str(grads['dw']))
print ('db = ' + str(grads['db']))
```

```
w = [[0.19033591]
      [0.12259159]]
b = 1.9253598300845747
dw = [[0.67752042]
      [1.41625495]]
db = 0.21919450454067657
```

Ejercicio: La función anterior calcula los parámetros w y b . La última función que necesitamos es la función predict(). Esta función será capaz de predecir la clase de unas muestras de test X , dados los parámetros w y b . Para ello, tenemos que dar dos pasos:

1. Calcula $\hat{y} = A = \sigma(W^T X + b)$.
2. Convierte la predicción a 0 (si $\hat{y} < 0,5$) o a 1 (si $\hat{y} \geq 0,5$). Guarda todas las predicciones en el array Y_prediction.

```
[10]: def predict(w, b, X):
    """
    Dados los parametros de la regresion logistica (w, b), predice las clases (0 o 1) de las muestras.

    Input:
    w -- pesos, numpy-array de tamaño (num_px * num_px * 3, 1).
    b -- parametro b (bias).
    X -- Conjunto de muestras X de test. Dimension: (num_px * num_px * 3, number of examples)

    Output:
    Y_prediction -- numpy-array con todas las predicciones obtenidas para las muestras en X (0/1).
    """

    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    # Calcula el vector 'A', donde tendremos las probabilidades de que cada foto contenga un gato.
    ## PON TU CODIGO AQUI ## ( 1 linea)
    A = sigmoid(np.dot(w.T, X) + b)

    #####
    for i in range(A.shape[1]):
        # Convierte las probabilidades A[0, i] a predicciones (0/1)
        Y_prediction[0, i] = 1 if A[0, i] >= 0.5 else 0

    #####
    assert(Y_prediction.shape == (1, m))

    return Y_prediction

w = np.array([[0.1124579],[0.23106775]])
b = -0.3
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]])
print ('predicciones = ' + str(predict(w, b, X)))

predicciones = [[1. 1. 0.]]
```

3.7. Construye el modelo y el clasificador

Ya tenemos todos los ingredientes. Ahora usaremos todas las funciones anteriores para aplicar un clasificador de regresión logística a nuestro dataset.

Ejercicio: Implementa el modelo usando las funciones anteriores. Utiliza los siguientes nombres para declarar las variables:

- Y_prediction_test: predicciones hechas sobre el dataset de test.
- Y_prediction_train: predicciones hechas sobre el dataset de train.

```
[12]: def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
    """
        Implementa el modelo de regresion logisitca usando las funciones implementadas anteriormente.

    Input:
        X_train -- numpy-array con el conjunto de muestras X de train. Dimension: (num_px * num_px * 3, m_train)
        Y_train -- numpy-array con el conjunto de etiquetas Y de train (0 -> no gato, 1 -> gato). Dimension: (1, m_train)
        X_test -- numpy-array con el conjunto de muestras X de test. Dimension: (num_px * num_px * 3, m_test)
        Y_test -- numpy-array con el conjunto de etiquetas Y de test (0 -> no gato, 1 -> gato). Dimension: (1, m_test)
        num_iterations -- Numero de iteraciones del algoritmo de gradient descent.
        learning_rate -- hyperparametro que representa la tasa de aprendizaje (learning_rate) de la regla de actualizacion en la funcion optimize().
        print_cost -- True para imprimir el coste cada 100 iteraciones.

    Output:
        d -- diccionario con la informacion sobre el modelo (coste, predicciones de test, predicciones de train, w, b, tasa de aprendizaje y numero de iteraciones).
    """

    # Inicializa los parametros a 0 ( 1 linea)
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent ( 1 linea)
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)

    # Extrae los parametros w y b
    w = parameters['w']
    b = parameters['b']

    # Predice las clases para las muestras de train y de test ( 2 lineas)
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    #####
    # Imprime la metrica de accuracy tanto para train como para test.
    print('train accuracy: {} %'.format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
    print('test accuracy: {} %'.format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))

    d = {'cost': costs,
          'Y_prediction_test': Y_prediction_test,
          'Y_prediction_train' : Y_prediction_train,
          'w' : w,
          'b' : b,
```

```
[12]:      'lr' : learning_rate,
      'iterations': num_iterations}

      return d

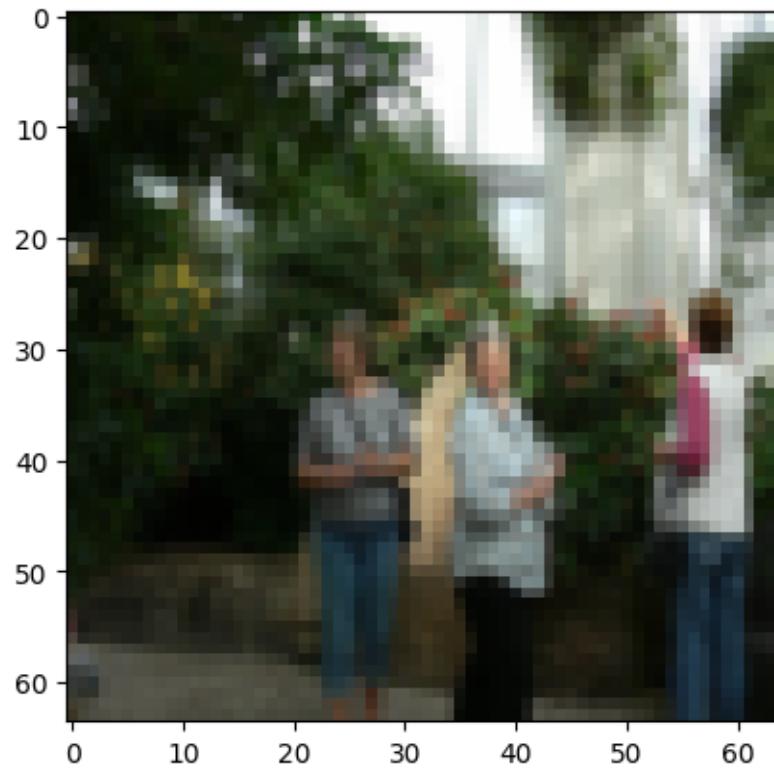
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, ↵
      ↵learning_rate = 0.005, print_cost = True)

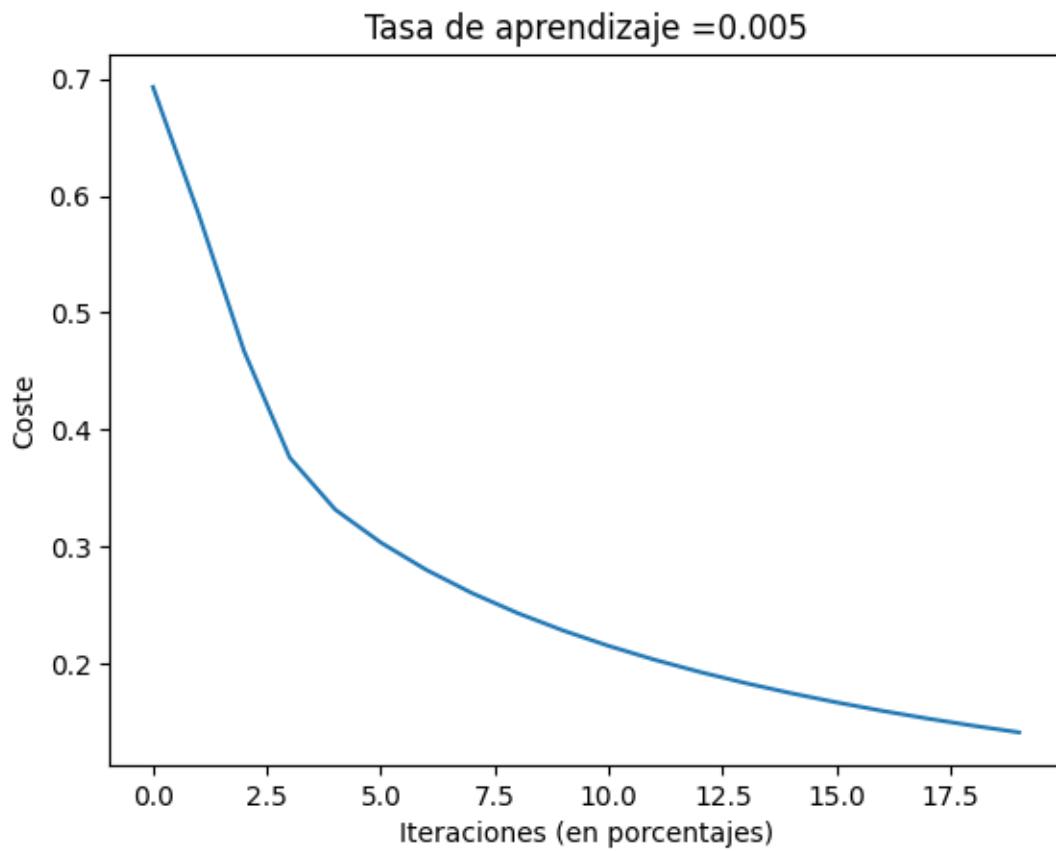
# Vamos a imprimir un ejemplo.
index = 21
plt.figure(1)
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))

print ('y = ' + str(test_set_y[0,index]) + ', you predicted that it is a \\' + ↵
      ↵classes[test_set_y[0,index]].decode('utf-8') + '\\' picture.')

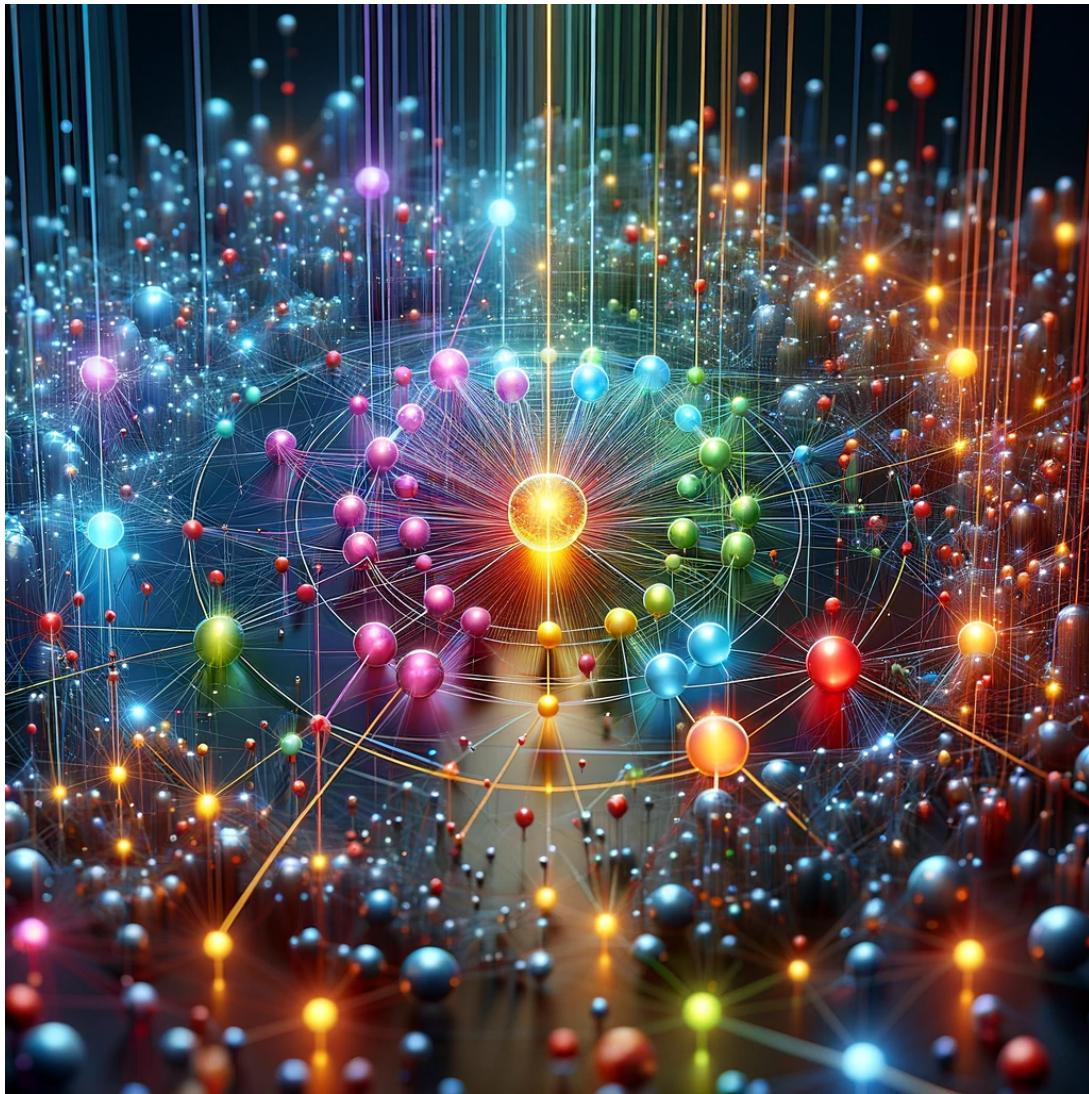
# Imprime la grafica del coste (curva de aprendizaje)
costs = np.squeeze(d['cost'])
plt.figure(2)
plt.plot(costs)
plt.ylabel('Coste')
plt.xlabel('Iteraciones (en porcentajes)')
plt.title('Tasa de aprendizaje =' + str(d['lr']))
plt.show()
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
y = 0, you predicted that it is a 'non-cat' picture.
```





4. Laboratorio 11: Programando en Python 'from the scratch' – K-Nearest Neighbours

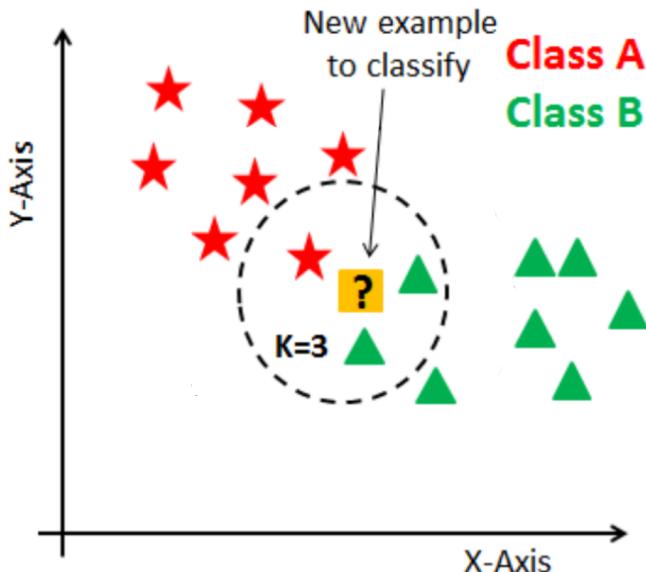


Al igual que en el anterior, expongo el laboratorio de collab ya completo. Una vez más, gracias por semejante trabajo y proyecto guiado.

Si bien existen muchos modelos de clasificación supervisada, uno de los más conocidos (e intuitivos) es el **K-Nearest Neighbors**, también conocido como **K-NN**. Este modelo de clasificación se basa en las distancias entre individuos del conjunto de datos. Dada cualquier muestra a clasificar x , el funcionamiento del **K-NN** es el siguiente:

1. Se calcula la distancia entre la muestra x y todas las muestras del conjunto de entrenamiento. La métrica de distancia utilizada dependerá de la elección del usuario.
2. Entre todas las muestras del conjunto de entrenamiento, se eligen las K más cercanas a x en base a la métrica de distancia utilizada.
3. Se predice la clase de x en base a la clase de los K vecinos más cercanos. La técnica utilizada dependerá

de la elección del usuario.



En la imagen anterior se puede observar un ejemplo de un 3-NN (es decir, un K-NN que solo se fija en los 3 vecinos más cercanos) para un conjunto de datos con solo 2 variables numéricas. En este caso, podemos considerar cada muestra como un punto en un plano de dos dimensiones, y por lo tanto, podemos utilizar la **distancia euclídea** como métrica de distancia. Si nos fijamos en las 3 muestras más cercanas a la muestra a clasificar, podemos observar que encontramos una muestra de la clase *A* y dos muestras de la clase *B*. Por lo tanto, utilizando la técnica del **voto mayoritario**, clasificaremos la nueva muestra como perteneciente a la clase más común entre sus vecinos más cercanos, en este caso, la clase *B*.

Teniendo todo esto en cuenta, en este *notebook* trataremos de programar desde cero un **K-NN** básico, y lo modificaremos iterativamente para mejorar su rendimiento. Finalmente, crearemos un pequeño esquema de **parameter tuning** para seleccionar la mejor configuración de hiper-parámetros para el modelo.

4.1. Preparación del entorno

Lo primero que vamos a hacer será importar los paquetes que necesitaremos para la implementación del *K-NN* (*numpy*, *sklearn*). Además, también importaremos varios paquetes de visualización de datos para poder visualizar mejor los resultados que obtengamos a lo largo de este *notebook* (*seaborn*, *matplotlib*).

```
[1]: import numpy as np
from sklearn import datasets, model_selection
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Una vez importados todos los paquetes, el siguiente paso es cargar y preparar la base de datos sobre la que trabajaremos. En concreto, en este *notebook* consideraremos el **Olivetti Faces data-set** disponible en la librería *sklearn*. Esta base de datos consta de 400 imágenes de caras en blanco y negro distribuidas en 40 clases distintas (numeradas del 0 al 39). Cada imagen tiene un tamaño de 64x64 pixeles, donde el color de cada pixel se representa con un número real entre el 0 (*negro*) y el 1 (*blanco*). Cada una de las clases se asocia con la cara de una persona concreta.

```
[2]: X, y = datasets.fetch_olivetti_faces(return_X_y=True)

fig = plt.figure(figsize=(10, 10))
for i in range(8):
    fig.add_subplot(4, 2, i+1)
    plt.imshow(X[5*i].reshape(64, 64), cmap = mpl.cm.gray, interpolation='nearest')
    plt.title('Class '+str(y[5*i]))
    plt.axis('off')
```

downloading Olivetti faces from <https://ndownloader.figshare.com/files/5976027>
to /root/scikit_learn_data

Class 0



Class 0



Class 1



Class 1



Class 2



Class 2



Class 3



Class 3



A continuación dividiremos el conjunto de datos en un conjunto de entrenamiento (80 %) y un conjunto de test (20 %). Para ello utilizaremos la función `train_test_split` del paquete `sklearn`. Configuraremos la función para que nos devuelva dos conjuntos con una proporción de clases similar (ver parámetro `stratify`).

[4]:

```
#DIVIDE LA BASE DE DATOS EN CONJUNTO DE ENTRENAMIENTO Y TEST
#LA PROPORCIÓN DE CLASES SERÁ SIMILAR EN AMBOS CONJUNTOS
```

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
    ↪stratify=y, test_size=0.2, random_state=8)

n_classes = 40

print ('Cantidad de muestras de train:', len(X_train))
print ('Cantidad de muestras de cada clase en train:', np.
    ↪unique(y_train,return_counts=True)[1])
print ()
print ('Cantidad de muestras de test:', len(X_test))
print ('Cantidad de muestras de cada clase en test:', np.
    ↪unique(y_test,return_counts=True)[1])
```

```
Cantidad de muestras de train: 320
Cantidad de muestras de cada clase en train: [8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8]
```

```
Cantidad de muestras de test: 80
Cantidad de muestras de cada clase en test: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

4.2. 1. K-NN básico

Como hemos mencionado anteriormente, los píxeles de cada imagen en nuestra base de datos se representan con números reales entre el 0 y el 1. Por lo tanto, nos encontramos ante una base de datos numérica. En estos casos, una elección natural para nuestro **K-NN** es utilizar la conocida **distancia euclídea**. Dadas dos muestras de n variables numéricas $x = \{x(1), x(2), \dots, x(n)\}$ y $z = \{z(1), z(2), \dots, z(n)\}$, la distancia euclídea entre ellas se calcula como:

$$d_e(x, z) = \sqrt{\sum_{i=1}^n (x(i) - z(i))^2} \quad (1)$$

Dicho esto, la primera tarea a realizar será crear un primer **K-NN básico** que utilice la distancia euclídea y el voto mayoritario para clasificar las instancias del conjunto de test.

```
[6]: ####CALCULA LA DISTANCIA EUCLIDEA ENTRE LOS VECTORES NUMÉRICOS A Y B
def euclidean(A,B,*args):
    dist = np.sqrt(np.sum((A - B) ** 2))
    return dist

#CALCULA LA CLASE EN FUNCIÓN DE LOS VECINOS (VOTO MAYORITARIO)
def majority(neighbors_classes,*args):
    #PISTA: Cuenta la cantidad de veces que aparece cada clase en los vecinos más cercanos, y quédate con aquella que sea más frecuente.
    (values, counts) = np.unique(neighbors_classes, return_counts=True)
    index = np.argmax(counts)
    selected_class = values[index]

    return selected_class

####DEVUELVE LA CLASE DE LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
def get_neighbors(test_row,X_train,y_train,k):
    dists = np.array([euclidean(test_row,train_row) for train_row in X_train])

    #PISTA: La posición i del array dists guarda la distancia entre la muestra a clasificar y la muestra i-ésima del conjunto de entrenamiento.
    neighbors_indices = np.argsort(dists)[:k]
    neighbors_classes = y_train[neighbors_indices]

    return neighbors_classes

###CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k):
    predictions = np.empty(len(X_test),dtype=y_train.dtype)
    for test_ind in range(len(X_test)):
        ####ENCUENTRA LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
        test_row = X_test[test_ind]
        neighbors_classes = get_neighbors(test_row,X_train,y_train,k)
        ####CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS (VOTO MAYORITARIO)
        predictions[test_ind] = majority(neighbors_classes)
    return predictions
```

Ahora que hemos implementado una primera versión de nuestro **K-NN**, vamos comprobar cómo funciona a la hora de predecir la identidad de las imágenes de test. Para ello, implementaremos dos funciones adicionales:

- Una función simple que nos calcule el **accuracy** del modelo en base a las clases predichas y reales.
- Una función que nos visualice en forma de mapa de calor la **matriz de confusión** de las predicciones realizadas. Para ello, utilizaremos la función *heatmap* del paquete *seaborn*.

Utilizaremos las funciones implementadas para comprobar el rendimiento del modelo con $K = 5$.

```
[7]: from sklearn.metrics import confusion_matrix as sk_confusion_matrix
#DEVUELVE EL ACCURACY EN BASE A LAS PREDICCIONES Y LAS CLASES REALES
def accuracy(pred,real):

    correct = np.sum(pred == real)
    total = len(real)
    accuracy = correct / total

    return accuracy

#MUESTRA LA MATRIZ DE CONFUSIÓN EN FORMA DE HEATMAP
def confusion_matrix(pred,real,n_classes):
    conf_mat = np.zeros((n_classes,n_classes))
    #CONSTRUYE LA MATRIZ DE CONFUSIÓN

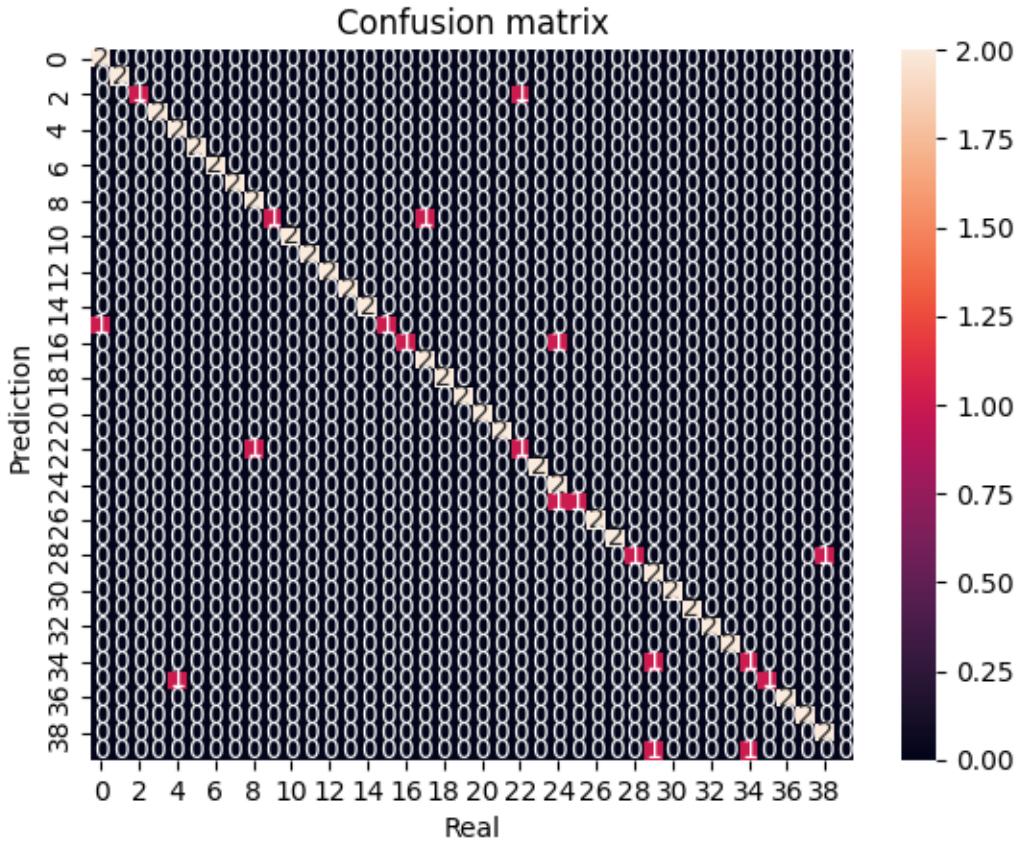
    # Utilizo sklearn
    conf_mat = sk_confusion_matrix(real, pred)

    #PISTA: La posición  $(i,j)$  de la matriz de confusión deberá contener la
    #cantidad de veces que una muestra con clase real  $j$  ha sido clasificada
    #como clase  $i$ .
    #

    #CREA Y MUESTRA EL HEATMAP BASADO EN LA MATRIZ DE CONFUSIÓN
    sns.heatmap(conf_mat,annot=True,fmt='g')
    plt.title('Confusion matrix')
    plt.ylabel('Prediction')
    plt.xlabel('Real')
    plt.show()

pred = k_nearest_neighbors(X_train,X_test,y_train,k=5)

confusion_matrix(pred,y_test,n_classes)
print('Test accuracy:',str(accuracy(pred,y_test)))
```



Test accuracy: 0.8625

Como se puede observar, los resultados obtenidos son bastante buenos para un modelo tan simple. Realizando todo de forma correcta, se debería obtener un accuracy cercano al 0.86. Sin embargo, como veremos a continuación, aún hay espacio para la mejora.

4.3. 2. K-NN ponderado según distancia

En la imagen previa se puede observar una situación en la que se utiliza un 5-NN para predecir la clase de una muestra dada. En este ejemplo, solo dos de los cinco vecinos más cercanos pertenecen a la clase *B*, mientras que los otros tres pertenecen a la clase *A*. Siguiendo la estrategia del **voto mayoritario**, el modelo predeciría la clase *A* para la nueva muestra, ya que es la clase más común entre sus vecinos.

Sin embargo, en la imagen se observa que los dos vecinos más cercanos son aquellos que pertenecen a la clase *B*. En este caso, ¿No deberíamos dar más importancia a aquellas muestras que estén más cerca? ¿Es justo darles la misma importancia a los *K* vecinos más próximos, sin considerar la distancia a la que se encuentran?

Si bien no hay una respuesta universal para estas preguntas, considerar la distancia a los vecinos más cercanos a la hora de hacer la predicción puede ser una buena opción en muchas situaciones, sobre todo cuando las muestras se encuentran cerca de la frontera de decisión entre clases. Una forma de hacer esto es utilizar un **voto mayoritario ponderado** a la hora de seleccionar la clase a predecir. En esta estrategia, la importancia de los vecinos va decreciendo según aumenta la distancia con respecto a la muestra a clasificar. Es decir, dada un muestra x , un conjunto de vecinos más cercanos $\{z_1, z_2, \dots, z_K\}$ y sus respectivas clases $\{y_1, y_2, \dots, y_K\}$, la clase de x se predice como:

$$y' = \underset{c_i \in C}{\operatorname{argmax}} \sum_{j=1}^K \left(\frac{\delta_{c_i y_j}}{d(x, z_j)} \right) \quad (2)$$

donde $\{c_1, c_2, \dots, c_m\}$ es el conjunto de todas las posibles clases y $d(x, z)$ es la distancia entre las muestras x y z . La expresión δ_{cy} es 1 si la condición $c = y$ es cierta, mientras que es 0 en caso contrario.

Si nos fijamos en la anterior ecuación, observaremos que la contribución de cada vecino a la votación es inversamente proporcional a su distancia con respecto a la muestra a clasificar. Por lo tanto, en este tipo de **K-NN ponderado** no todos los vecinos tienen la misma relevancia.

Teniendo esto en cuenta, vamos a implementar un **K-NN** que acepte distintos tipos de estrategias de selección de clase en base a los vecinos más cercanos. Para ello, deberemos modificar el **K-NN** del ejercicio anterior de la siguiente forma:

- Modificar la función `get_neighbors` de forma que nos devuelva no solo los vecinos más cercanos, sino la distancia a la que se encuentran.
- Modificar la función `k_nearest_neighbors` de forma que acepte un parámetro extra llamado `selection`, el cuál nos indicará qué estrategia utilizar para elegir la clase en base a los vecinos más cercanos.
- Añadir una función `weighted_majority` que implemente el voto mayoritario ponderado.

```
[47]: #CALCULA LA CLASE EN FUNCIÓN DE LOS VECINOS (VOTO MAYORITARIO PONDERADO)
def weighted_majority(neighbors_classes,neighbors_dists):
    # PISTA: Fíjate en la clase de cada uno de los vecinos más cercanos,
    # y pondera cada caso según la distancia a la muestra a clasificar.
    votes = {}
    for i in range(len(neighbors_classes)):
        vote = neighbors_classes[i]
        weight = 1 / neighbors_dists[i] if neighbors_dists[i] != 0 else float('inf')
        if vote in votes:
            votes[vote] += weight
        else:
            votes[vote] = weight
    selected_class = max(votes, key=votes.get)

    return selected_class

###DEVUELVE LA CLASE Y DISTANCIA DE LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
def get_neighbors(test_row,X_train,y_train,k):

    distances = [euclidean(test_row, train_row) for train_row in X_train]
    neighbors_indices = np.argsort(distances)[:k]
    neighbors_classes = y_train[neighbors_indices]
    neighbors_dists = np.array(distances)[neighbors_indices]

    #PISTA: Copia y modifica el código implementado en el ejercicio anterior de
    #forma que la función devuelva la distancia a la que se encuentran los vecinos
    #más cercanos.
    #...

    return neighbors_classes, neighbors_dists

###CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k,selection):
    predictions = np.empty(len(X_test),dtype=y_train.dtype)
    for test_ind in range(len(X_test)):
        ###ENCUENTRA LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
        test_row = X_test[test_ind]
        neighbors_classes, neighbors_dists = get_neighbors(test_row,X_train,y_train,k)
        ###CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS
        predictions[test_ind] = selection(neighbors_classes,neighbors_dists)
    return predictions
```

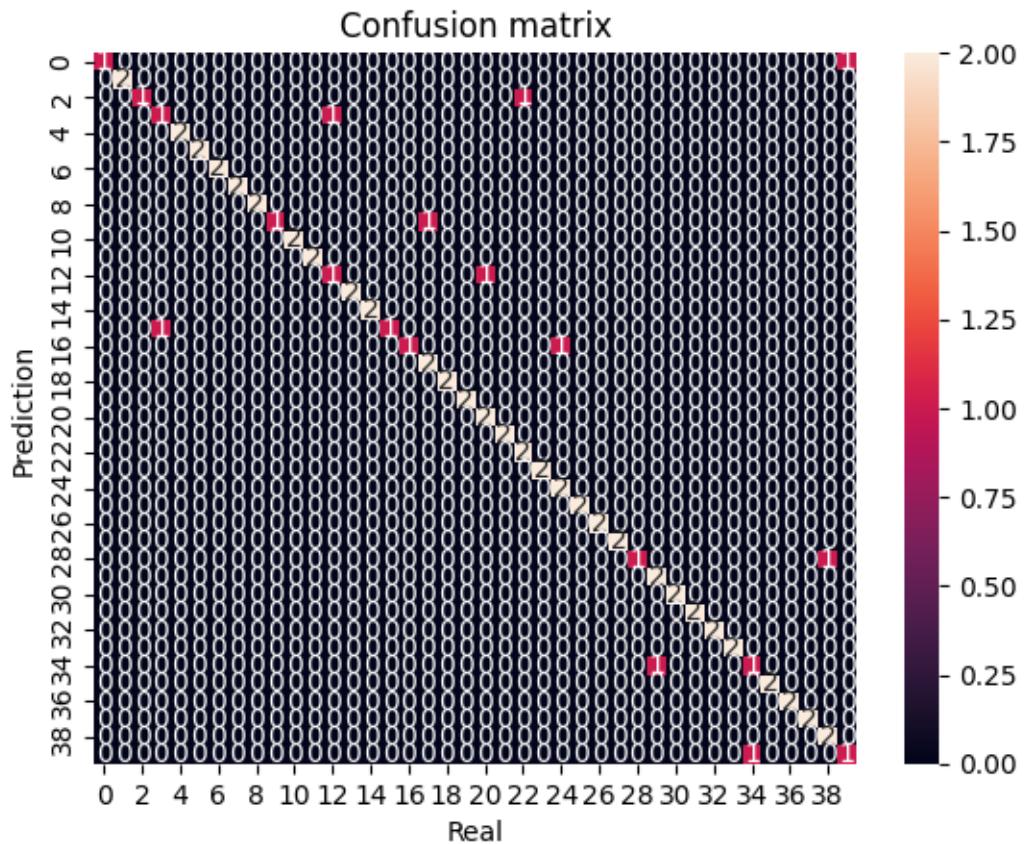
A continuación, comprobaremos los resultados que se obtienen en el conjunto de test si utilizamos la estrategia del voto mayoritario ponderado. Al igual que en el anterior ejercicio, nos fijaremos solo en los 5 vecinos más próximos.

[9]: *#INDICA QUE LA SELECCIÓN DE CLASE EN BASE A LOS VECINOS MÁS CERCANOS SE REALIZARÁ #A TRAVÉS DE UN VOTO MAYORITARIO PONDERADO*

#PISTA: En Python, se pueden pasar funciones como argumentos de otra función.

```
pred = k_nearest_neighbors(X_train,X_test,y_train,k=5,selection=weighted_majority)

confusion_matrix(pred,y_test,n_classes)
print('Test accuracy:',str(accuracy(pred,y_test)))
```



Test accuracy: 0.875

Como se puede observar, los resultados obtenidos son algo mejores que en el caso del voto mayoritario. En concreto, se obtienen valores de accuracy de alrededor de 0.87, lo que implica una subida de 0.01 con respecto al anterior ejercicio.

4.4. 3. Distancias alternativas

Hasta ahora, los **K-NN** que hemos implementado para nuestra base de datos se han basado siempre en la **distancia euclídea**. Sin embargo, esta no es la única opción. Por ejemplo, si los datos fueran categóricos en lugar de numéricos, nos veríamos obligados a utilizar otras métricas de distancia, tales como la distancia de Jaccard o la distancia de Hamming.

Incluso con datos numéricos, existen multitud de alternativas a la distancia euclídea. De hecho, existen otras métricas que pueden ser incluso más adecuadas cuando la dimensionalidad del problema es alta

(como en nuestro caso). Una de las más conocidas y utilizadas es la **distancia de Manhattan**. Dadas dos muestras de n variables numéricas $x = \{x(1), x(2), \dots, x(n)\}$ y $z = \{z(1), z(2), \dots, z(n)\}$, la distancia de Manhattan entre ellas se calcula como:

$$d_m(x, z) = \sum_{i=1}^n |x(i) - z(i)| \quad (3)$$



La imagen anterior muestra visualmente la diferencia entre la distancia euclídea (*verde*) y la de Manhattan (*azul*) para dos puntos en un plano de dos dimensiones (en este ejemplo, un mapa). La distancia euclídea calcula la distancia más corta entre ambos puntos, mientras que la distancia de Manhattan calcula la suma de las diferencias en todas las dimensiones (o variables).

En este ejercicio, vamos a mejorar nuestro **K-NN** para que acepte distintas métricas de distancia. Para ello, haremos lo siguiente:

- Modificar las funciones `get_neighbors` y `k_nearest_neighbors` para que acepten un nuevo argumento `distance` que nos indicará la métrica de distancia a utilizar.
- Añadir una nueva función `manhattan` que implemente la distancia de Manhattan. Recordemos que la distancia euclídea ya la implementamos en el primer ejercicio (función `euclidean`).

```
[14]: #CALCULA LA DISTANCIA DE MANHATTAN ENTRE LOS VECTORES NUMÉRICOS A Y B
def manhattan(A,B,*args):
    dist = np.sum(np.abs(A - B))
    return dist

###DEVUELVE LA CLASE Y DISTANCIA DE LOS K VECINOS MÁS CERCANOS
def get_neighbors(test_row,X_train,y_train,k,dist_func):

    distances = [dist_func(test_row, train_row) for train_row in X_train]
    neighbors_indices = np.argsort(distances)[:k]
    neighbors_classes = y_train[neighbors_indices]
    neighbors_dists = np.array(distances)[neighbors_indices]

    #PISTA: Copia y modifica el código implementado en el ejercicio anterior de
    #forma que la función admita distintas métricas de distancia.
    #...

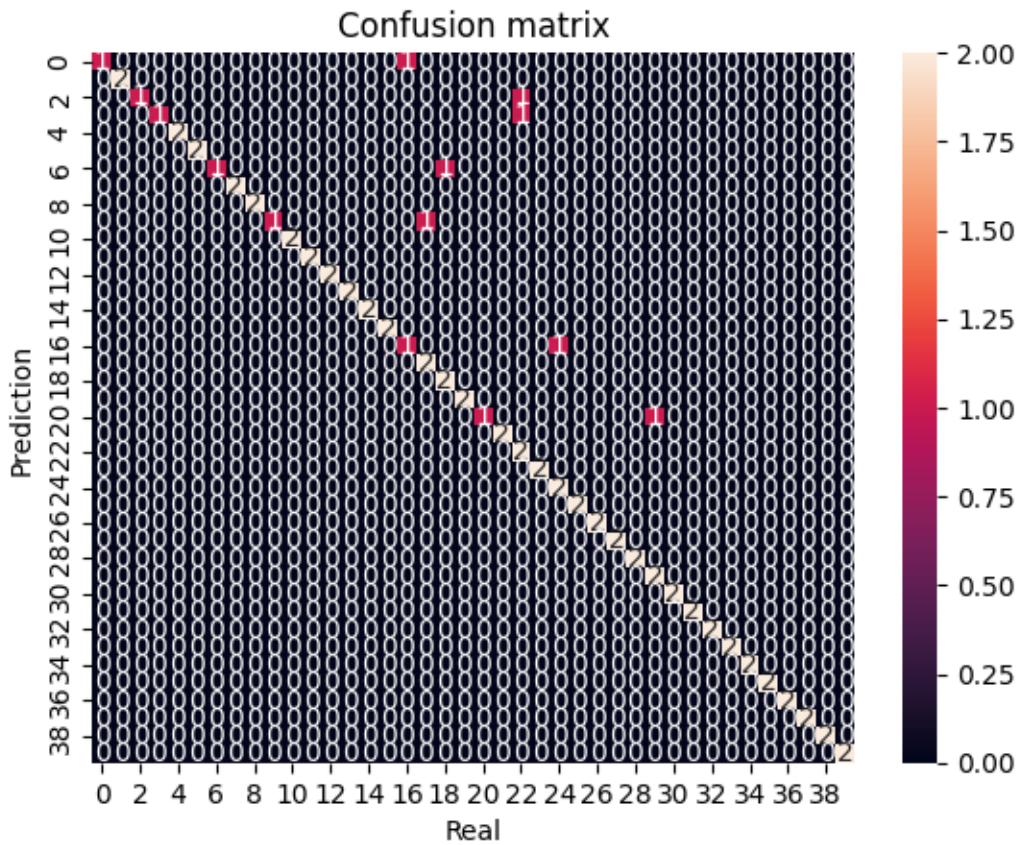
    return neighbors_classes, neighbors_dists

###CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k,selection,distance):
    predictions = np.empty(len(X_test),dtype=y_train.dtype)
    for test_ind in range(len(X_test)):
        ###ENCUENTRA LOS K VECINOS MÁS CERCANOS
        test_row = X_test[test_ind]
        neighbors_classes, neighbors_dists = get_neighbors(test_row,X_train,y_train,k,distance)
        ###CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS
        predictions[test_ind] = selection(neighbors_classes,neighbors_dists)
    return predictions
```

Finalmente, probemos cómo funciona nuestro **K-NN** con la distancia de Manhattan y el voto mayoritario ponderado implementado en el ejercicio anterior. Como hasta ahora, consideraremos $K = 5$.

```
[15]: pred =_
    k_nearest_neighbors(X_train,X_test,y_train,k=5,selection=weighted_majority,distance=manhattan)

confusion_matrix(pred,y_test,n_classes)
print('Test accuracy:',str(accuracy(pred,y_test)))
```



Test accuracy: 0.9125

Con solo cambiar la métrica de distancia, hemos conseguido aumentar el accuracy del modelo hasta alrededor de 0.91. Por lo tanto, elegir la distancia correcta para cada problema es crucial a la hora de crear un buen modelo **K-NN**.

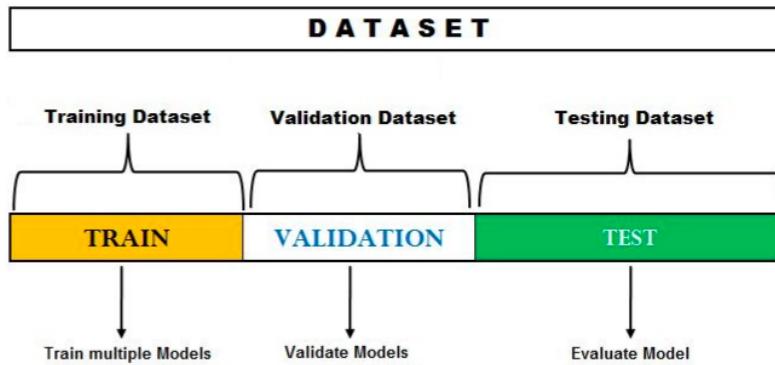
4.5. 4. Parameter tuning

Como hemos visto hasta el momento, parece que elegir una buena configuración de hiper-parámetros (distancia, método de selección de clase...) resulta vital para obtener un modelo con un buen rendimiento. Esto no ocurre solo en el **K-NN**, sino que cualquier modelo de clasificación con hiper-parámetros necesita ser ajustado de antemano. Este proceso de elegir la mejor configuración posible para un modelo es lo que se conoce como **parameter tuning**.

Si bien hasta ahora hemos estado probando las distintas versiones del **K-NN** sobre el conjunto de test, esto no debe hacerse a la hora de realizar el parameter tuning. El conjunto de test solo debe de utilizarse para comprobar el rendimiento del modelo final, lo que nos dará una idea de cómo *generaliza* nuestro clasificador al presentarle datos nunca antes vistos. Si utilizásemos el conjunto de test para hacer el parameter tuning, los resultados que obtendríamos sobre dicho conjunto estarían sesgados debido a nuestras decisiones, por lo que las métricas de generalización no serían fiables. Este fenómeno es lo que se conoce como *data leaking*. Por lo tanto, la elección de los hiper-parámetros debe de ser realizada tomando en cuenta solo el conjunto de entrenamiento.

Para ello, en este ejercicio dividiremos el conjunto de entrenamiento en dos porciones diferenciadas, y utilizaremos una de ellas como **conjunto de validación**. Es decir, crearemos los modelos considerando

solo una porción de las muestras de entrenamiento, y utilizaremos el resto para probar el rendimiento de las distintas configuraciones de hiper-parámetros. Por lo tanto, nuestra base de datos quedará particionada de la siguiente forma.



El parameter tuning que implementaremos se centrará en optimizar la cantidad de vecinos a considerar (K), y supondremos que tanto la distancia como el método de selección de clase ya han sido definidos de antemano por un experto. En concreto, utilizaremos la **distanzia de Manhattan** y el **voto mayoritario ponderado** implementados en ejercicios anteriores.

El parámetro K será elegido mediante una estrategia **grid search**. Este método se basa en definir un conjunto de configuraciones de hiper-parámetros que se evalúan una por una sobre el conjunto de validación. Después, se comparan los resultados obtenidos por cada configuración, y se seleccionan aquellos hiper-parámetros que obtengan los mejores resultados. Es importante destacar que esta estrategia no prueba todas las posibles combinaciones de los hiper-parámetros, sino que tendremos que decidir de antemano qué valores queremos probar.

Teniendo todo esto en cuenta, vamos a implementar un parameter tuning basado en grid search para optimizar el parámetro K de nuestro **K-NN**. También incluiremos una visualización de los valores de accuracy obtenidos en el conjunto de validación por las distintas configuraciones del hiper-parámetro K (función *plot* del paquete *matplotlib*).

```
[18]: #CREA Y MUESTRA LA GRÁFICA QUE REPRESENTA EL ACCURACY EN VALIDACIÓN OBTENIDO POR LOS DISTINTOS VALORES DE K
def plot_tuning(k_values,k_acc):
    plt.plot(k_values,k_acc,'o-')
    plt.ylabel('Validation Accuracy')
    plt.xlabel('k')
    plt.xticks(k_values)
    plt.show()

#REALIZA EL PARAMETER TUNING DEL PARÁMETRO K DEL K-NN
def parameter_tuning(X_train,y_train,validation_split,k_values):
    #DIVIDE EL CONJUNTO DE ENTRENAMIENTO PARA OBTENER EL CONJUNTO DE VALIDACIÓN
    X_train_split, X_validation, y_train_split, y_validation = model_selection.train_test_split(X_train, y_train, stratify=y_train, test_size=validation_split, random_state=8)
    #EVALUA LOS DISTINTOS MODELOS EN EL CONJUNTO DE VALIDACIÓN
    k_acc = np.zeros(len(k_values))

    for i, k in enumerate(k_values):
        pred = k_nearest_neighbors(X_train_split, X_validation, y_train_split, k=k, selection=weighted_majority, distance=manhattan)
        k_acc[i] = accuracy(pred, y_validation)

    #PISTA: La posición i del array k_acc deberá contener el accuracy de validación del modelo K-NN con el valor de K indicado en la posición i del array k_values.
    #Recuerda que los modelos deberán utilizar la distancia de manhattan y el voto mayoritario ponderado.
    #...

    plot_tuning(k_values,k_acc)
    #ELIGE EL VALOR DE K CON EL QUE SE HAN OBTENIDO MEJORES RESULTADOS

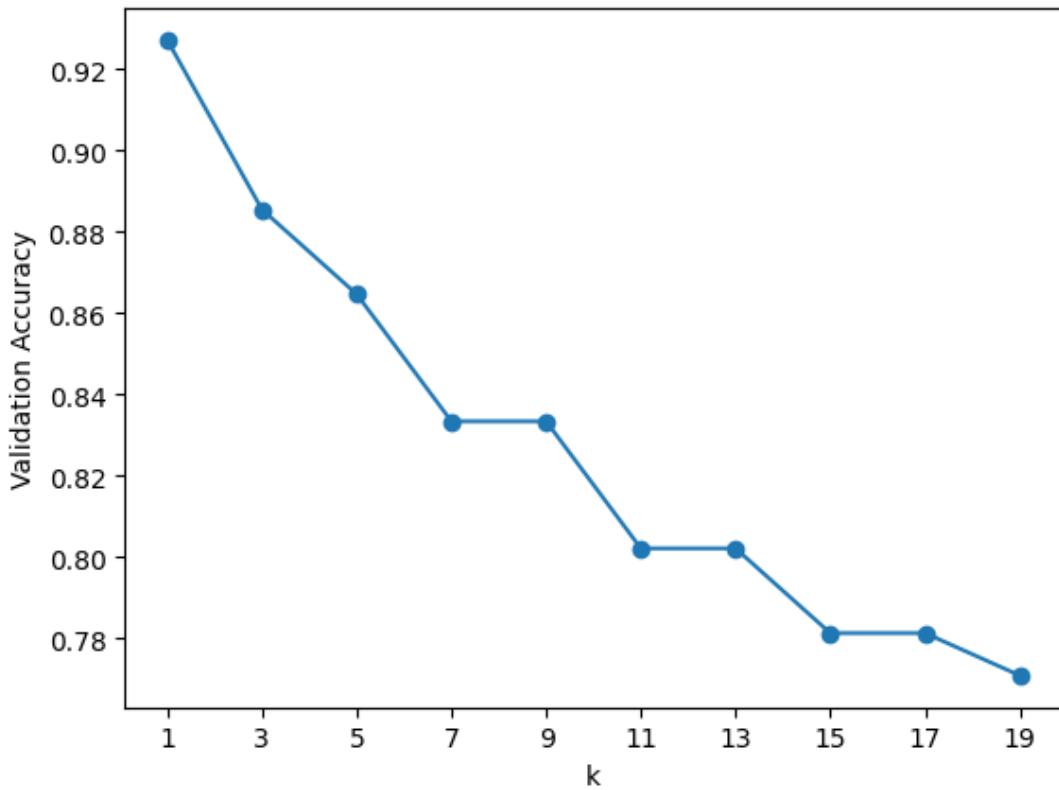
    max_acc_index = np.argmax(k_acc)
    max_acc = k_acc[max_acc_index]
    best_k = k_values[max_acc_index]

    #PISTA: Deberás devolver tanto el mejor accuracy de validación como el valor de K para el que se haya obtenido dicho resultado.
    #...

    return max_acc,best_k
```

A continuación, probemos nuestro parameter tuning. Para ello, utilizaremos un conjunto de validación compuesto por el 30 % de las muestras de entrenamiento. En cuanto a los valores de K a considerar, probaremos con: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19].

```
[19]: k_values = np.array([1,3,5,7,9,11,13,15,17,19])
max_acc, best_k = parameter_tuning(X_train,y_train,0.3,k_values)
print('Best validation accuracy:',max_acc,', Best k:',best_k)
```

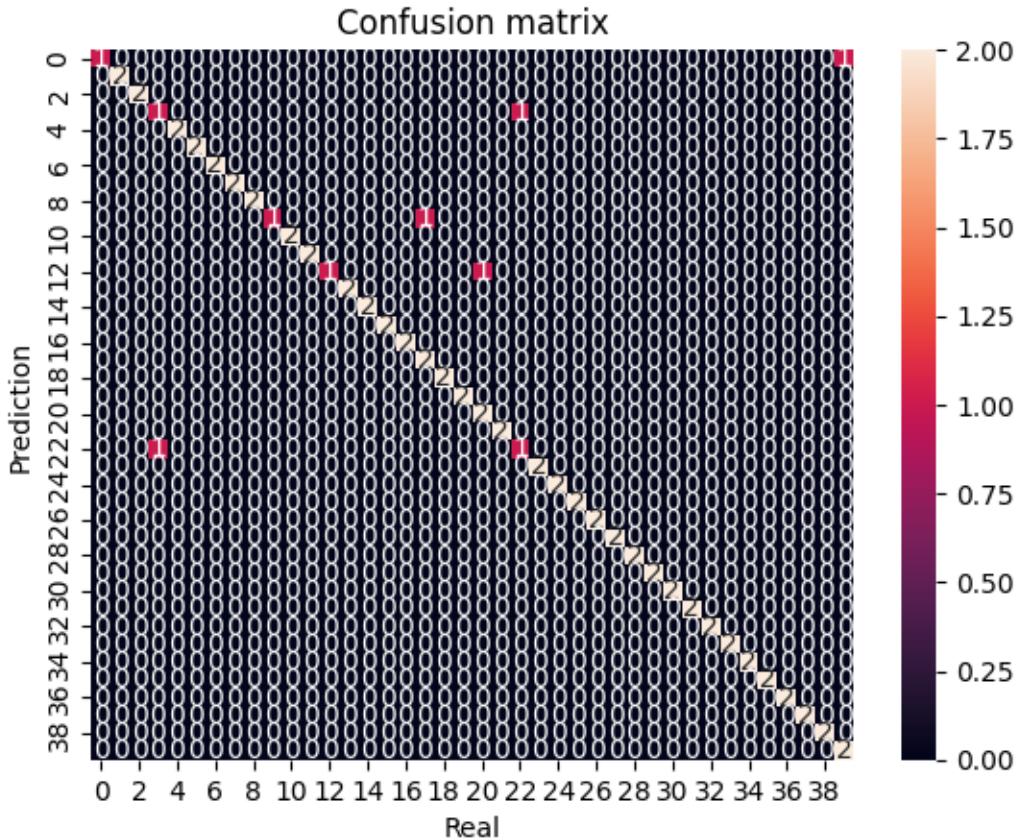


Best validation accuracy: 0.9270833333333334 , Best k: 1

Al parecer, el modelo obtiene los mejores resultados con $K = 1$. De hecho, la gráfica debería mostrar que el accuracy en el conjunto de validación decrece según aumenta el valor de K . Esto tiene sentido ya que la cantidad de muestras por cada clase en nuestra base de datos es muy pequeña, por lo que considerar muchos vecinos podría llevar a error.

Ahora sí, una vez que hemos decidido la configuración de los hiper-parámetros para el modelo, podemos comprobar su rendimiento sobre el conjunto de test. En este punto consideraremos el conjunto de entrenamiento entero para construir el modelo, incluyendo las instancias que utilizamos para la validación durante el parameter tuning.

```
[20]: pred =  
    k_nearest_neighbors(X_train,X_test,y_train,k=best_k,selection=weighted_majority,distance=manhattan  
  
    confusion_matrix(pred,y_test,n_classes)  
    print('Test accuracy:',str(accuracy(pred,y_test)))
```



Test accuracy: 0.9375

Con el parámetro K optimizado el accuracy en el conjunto de test aumenta hasta alrededor de 0.94. Este accuracy resulta muy similar al obtenido durante la validación, lo que demuestra que nuestro **K-NN** es capaz de generalizar de forma correcta a muestras nunca antes vistas.

4.6. 5. K-NN con funciones de librería

A lo largo de este *notebook* hemos implementado un **K-NN** que hemos mejorado iterativamente, añadiendo distintas métricas de distancia, métodos de selección de clase, e incluso un parameter tuning para decidir el valor de K . Todo esto ha sido hecho a mano, utilizando solo funciones de librería básicas como apoyo. Sin embargo, a la hora de trabajar con este tipo de modelos en el mundo real no hace falta que implementemos todo desde cero. Existen decenas de paquetes que incluyen métodos de clasificación como el **K-NN**, siendo un ejemplo claro el conocido *sklearn*.

Este ejercicio consiste en ejecutar un **K-NN** con las mismas características que el modelo final del anterior apartado utilizando solo funciones del paquete *sklearn*. Para ello, será necesario investigar la documentación de dicho paquete para entender las distintas funciones y sus argumentos. Si todo se realiza correctamente, los resultados en el conjunto de test deberían ser idénticos a los obtenidos por nuestro modelo.

```
[21]: from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

#DEFINE, CREA Y EVALUA EL MODELO
neigh = KNeighborsClassifier(n_neighbors=best_k, weights='distance',  

    metric='manhattan', algorithm='brute')

# Entrenamiento
neigh.fit(X_train, y_train)

pred = neigh.predict(X_test)

#PISTA: Crea el modelo utilizando las muestras de train y evalúalo sobre el conjunto  

#de test. Utiliza solo funciones de la librería sklearn.
#...

print('Test accuracy:', str(accuracy_score(pred,y_test)))
```

Test accuracy: 0.9375

4.7. EXTRA-1. Variables categóricas

Si bien hasta el momento hemos considerado una base de datos con variables numéricas, el **K-NN** puede ser utilizado para bases de datos categóricas o incluso mixtas. La principal diferencia en estos casos es la métrica de distancia a utilizar, la cuál deberá ser una métrica válida para el tipo de datos sobre el que trabajemos.

En este ejercicio, implementaremos desde cero dos métricas distintas para conjuntos de datos con variables *binarias*: la **distancia de Hamming** y la **distancia de Jaccard-Needham**. El muy conocido paquete *scipy* incluye este tipo de métricas, por lo que sería una buena idea investigar en su documentación para entender cómo se definen dichas distancias.

```
[49]: import scipy

#CALCULA LA DISTANCIA DE HAMMING ENTRE LOS VECTORES BINARIOS A Y B
def hamming(A,B,*args):
    # https://en.wikipedia.org/wiki/Hamming_distance
    dist = np.sum(A != B) / len(A)
    return dist

#CALCULA LA DISTANCIA DE JACCARD-NEEDHAM ENTRE LOS VECTORES BINARIOS A Y B
#CONSIDERA QUE 1=TRUE y 0=FALSE
def jaccard(A,B,*args):
    # Distancia jaccard: https://docs.scipy.org/doc/scipy/reference/generated/scipy.  

    # spatial.distance.jaccard.html
    # No consigo implementarlo, me rindo y tiro de scipy...
    intersection = np.sum(np.logical_and(A, B))
    union = np.sum(np.logical_or(A, B))
    dist = 1 - intersection / union if union != 0 else 0
    return scipy.spatial.distance.jaccard(A, B)
```

Para probar las métricas que acabamos de implementar, vamos a **discretizar** nuestra base de datos

numérica (**Olivetti Faces data-set**) de forma que cada muestra sea un vector binario. Para ello, lo que haremos será convertir todos los valores menores o iguales que 0.5 a 0, y todos los valores mayores que 0.5 a 1. La variable clase no deberá ser modificada.

```
[34]: #DISCRETIZA LA BASE DE DATOS SIN MODIFICAR LA VARIABLE CLASE (TANTO EN TRAIN COMO EN TEST)

X_train_binary = np.where(X_train > 0.5, 1, 0)
X_test_binary = np.where(X_test > 0.5, 1, 0)

fig = plt.figure(figsize=(10, 10))
for i in range(8):
    fig.add_subplot(4, 2, i+1)
    plt.imshow(X_train[5*i].reshape(64, 64), cmap = mpl.cm.gray, interpolation='nearest')
    plt.title('Class ' + str(y_train[5*i]))
    plt.axis('off')
```

Class 32



Class 39



Class 3



Class 11



Class 9



Class 8



Class 0

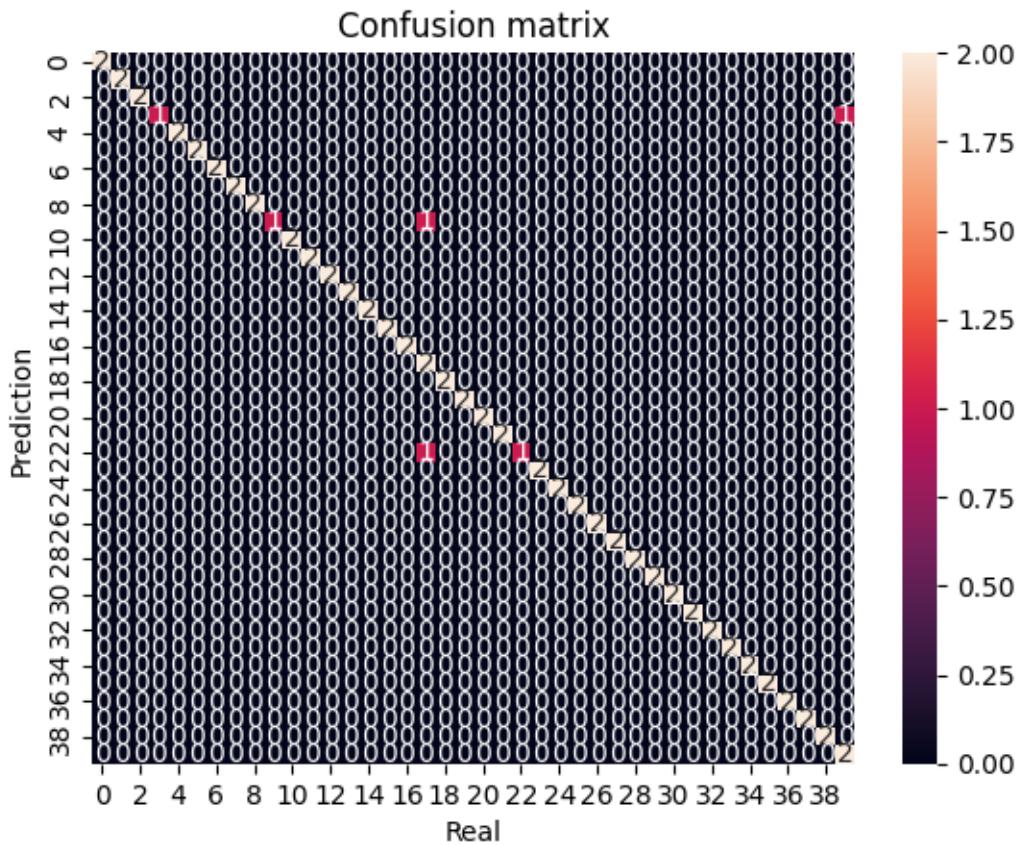


Class 34



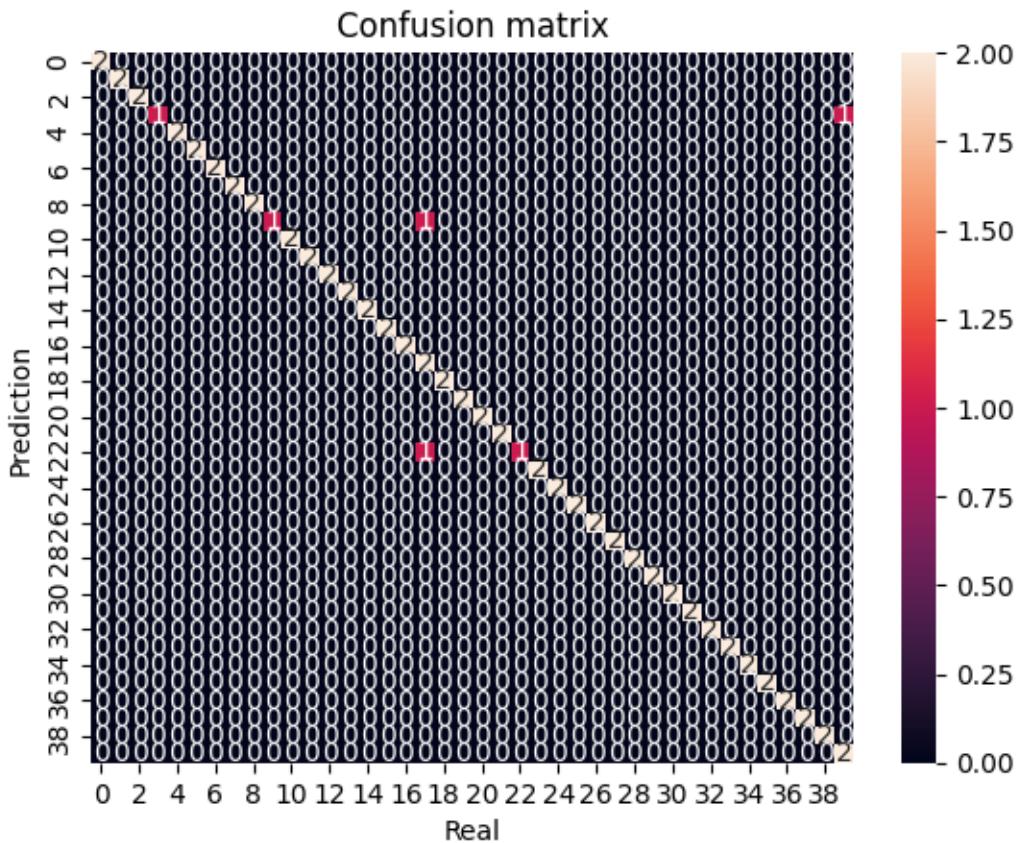
A continuación utilizaremos nuestro **K-NN** con las distancias de Hamming y Jaccard-Needham para clasificar las instancias de test usando la base de datos discretizada. Nos fijaremos únicamente en el vecino más cercano.

```
[35]: pred =  
    k_nearest_neighbors(X_train,X_test,y_train,k=1,selection=majority,distance=hamming)  
  
confusion_matrix(pred,y_test,n_classes)  
print('Hamming test accuracy:',str(accuracy(pred,y_test)))
```



Hamming test accuracy: 0.9625

```
[40]: pred =  
    k_nearest_neighbors(X_train,X_test,y_train,k=1,selection=majority,distance=jaccard)  
  
confusion_matrix(pred,y_test,n_classes)  
print('Jaccard test accuracy:',str(accuracy(pred,y_test)))
```



Jaccard test accuracy: 0.9625

Como se puede observar, obtenemos resultados bastante buenos incluso con la pérdida de información derivada de la discretización. En caso de haber realizado todo correctamente, se deberían obtener valores de accuracy de alrededor de 0.86 en el caso de la distancia de Hamming y de alrededor de 0.84 en el caso de la distancia de Jaccard-Needham.

4.8. EXTRA-2. Información Mutua

Una de las métricas más utilizadas para medir la correlación entre variables categóricas es la **información mutua**. La información mutua nos indica la *cantidad de información* sobre una variable que nos proporciona el conocer el valor de otra variable. Este tipo de medidas son muy útiles para conocer qué variables predictoras nos dan más información sobre la variable clase a predecir.

Lo primero que vamos a hacer en este ejercicio es crear una función que nos permita calcular la información mutua $I(X; Y)$ entre dos variables categóricas X e Y . Para ello, será necesario buscar y entender la fórmula de esta popular métrica de correlación.

```
[41]: #CALCULA LA ENTROPIA DE SHANNON EN FUNCIÓN DE LAS PROBABILIDADES DE ENTRADA
def shannon_entropy(P_x):
    P_x = P_x[P_x > 0]
    entropy = -np.sum(P_x * np.log2(P_x))
    return entropy

#CALCULA LA INFORMACIÓN MUTUA ENTRE LAS VARIABLES CATEGÓRICAS X E Y
def mutual_information(X,Y):
    length = len(X)

    P_x = np.array([np.sum(X == x) for x in np.unique(X)]) / length
    P_y = np.array([np.sum(Y == y) for y in np.unique(Y)]) / length
    P_xy = np.array([[np.sum((X == x) & (Y == y)) for y in np.unique(Y)] for x in np.unique(X)]) / length

    # Calcula las entropías H(X), H(Y) y H(X, Y)
    H_x = shannon_entropy(P_x)
    H_y = shannon_entropy(P_y)
    H_xy = shannon_entropy(P_xy.flatten())

    #PISTA: Dederás calcular las probabilidades necesarias para el cálculo de las
    #entropias H(X), H(Y) y H(X, Y).
    #...

    return H_x + H_y - H_xy
```

Como hemos mencionado anteriormente, la información mutua puede ser muy útil para descubrir cuales son las variables predictoras más relevantes en una base de datos. Esta información puede ser crucial a la hora de calcular las distancias entre muestras en el **K-NN**, ya que nos permite dar más relevancia a aquellas variables con una gran relación con el valor de clase, e ignorar aquellas que solo añadan ruido al proceso de predicción.

Por este motivo, el último paso a realizar será añadir una nueva métrica de distancia para variables categóricas llamada *weighted_hamming*, la cuál será una versión modificada de la **distancia de Hamming**. La principal particularidad de esta métrica es que incluirá un *pesado de variables predictoras*. En nuestro caso, dicho peso será proporcional a la **información mutua** con respecto a la variable clase. Es decir, el peso de cada variable X_i se calculará de la siguiente forma:

$$w_i = \frac{I(X_i; Y)}{\sum_{j=1}^n I(X_j; Y)} \quad (4)$$

donde $\{X_1, X_2, \dots, X_n\}$ es el conjunto de todas las variables predictoras e Y es la variable clase. Los pesos a utilizar en la métrica de distancia se especificarán mediante un nuevo argumento *weights* en la función *k_nearest_neighbors*.

```
[42]: #CALCULA LA DISTANCIA DE HAMMING ENTRE LOS VECTORES BINARIOS A Y B PONDERADA EN BASE A LOS PESOS EN W
def weighted_hamming(A,B,W):
    weighted_dist = np.sum(W * (A != B))
    return weighted_dist

###DEVUELVE LA CLASE Y DISTANCIA DE LOS K VECINOS MÁS CERCANOS
def get_neighbors(test_row,X_train,y_train,k,distance,weights):

    if weights is not None:
        dists = [distance(test_row, train_row, weights) for train_row in X_train]
    else:
        dists = [distance(test_row, train_row) for train_row in X_train]

    neighbors_indices = np.argsort(dists)[:k]
    neighbors_classes = y_train[neighbors_indices]
    neighbors_dists = np.array(dists)[neighbors_indices]

    #PISTA: Copia y modifica la función del ejercicio 3 de forma que acepte métricas
    #de distancia con pesado de variables.
    #...

    return neighbors_classes, neighbors_dists

#CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k,selection,distance,weights=None):
    predictions = np.empty(len(X_test),dtype=y_train.dtype)
    for test_ind in range(len(X_test)):
        ###ENCUENTRA LOS K VECINOS MÁS CERCANOS
        test_row = X_test[test_ind]
        neighbors_classes, neighbors_dists = get_neighbors(test_row,X_train,y_train,k,distance,weights)
        ###CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS
        predictions[test_ind] = selection(neighbors_classes,neighbors_dists)
    return predictions
```

Finalmente, vamos a probar la nueva métrica de distancia con pesado de variables utilizando nuestra base de datos discretizada. Es importante tener en cuenta que el peso de las variables se debe calcular tomando en cuenta **SOLO** las instancias de entrenamiento, ya que de lo contrario sufriríamos de *data leaking*. Al igual que en el anterior ejercicio, nos fijaremos solo en el vecino más cercano para clasificar las instancias de test.

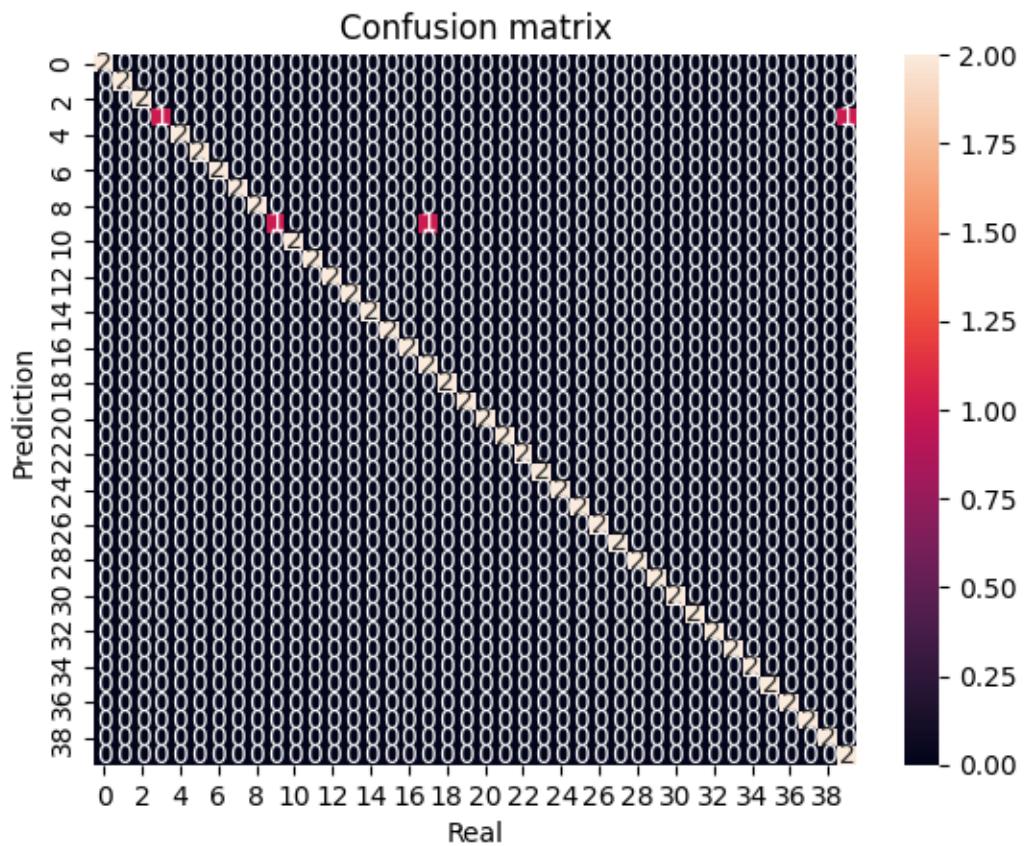
```
[43]: #CALCULA EL PESADO DE VARIABLES EN BASE A LA INFORMACIÓN MUTUA
weights = np.zeros(len(X_train[0]))

for i in range(len(weights)):
    weights[i] = mutual_information(X_train_binary[:, i], y_train)

#PISTA: La posición i del array weights debe contener el peso de la i-ésima variable. Los pesos se calculan en base a la información mutua con respecto al valor de clase.
#...

pred = k_nearest_neighbors(X_train,X_test,y_train,k=1,selection=majority,distance=weighted_hamming,weights=weights)

confusion_matrix(pred,y_test,n_classes)
print('Test accuracy:',str(accuracy(pred,y_test)))
```



Test accuracy: 0.975

[]:

Como se puede observar, obtenemos resultados de accuracy ligeramente superiores a los obtenidos por la distancia de Hamming sin pesado de variables.

Por curiosidad, antes de finalizar vamos a representar en forma de imagen en blanco y negro los pesos utilizados para las variables predictoras en el modelo anterior. Dado que cada variable en nuestra base de datos representa un píxel de una imagen, esta visualización nos dará una idea sobre qué partes de las imágenes son más relevantes a la hora de clasificar los rostros del **Olivetti Faces data-set**. Teniendo esto en cuenta, el color blanco representa píxeles con una información mutua más alta con respecto a la variable clase, mientras que lo contrario ocurre con el color negro.

```
[44]: plt.imshow(weights.reshape(64, 64), cmap = mpl.cm.gray, interpolation='nearest')
plt.title('Mutual information with respect to Class label')
plt.axis('off')
plt.show()
```

Mutual information with respect to Class label



5. Laboratorio 12: Algoritmos Genéticos para seleccionar variables de forma 'wrapper' en un problema de clasificación supervisada



Me gustaría comentar, que el siguiente enlace me ha ayudado bastante y parecido muy interesante; creo que explica los conceptos de manera muy clara y expone ejemplos que ayudan bastante a comprender las ideas:

https://www.neuraldesigner.com/blog/genetic_algorithms_for_feature_selection/

5.1. Algoritmo genético

1: Representación de los individuos

¿Cómo representarás-codificarás un individuo de tu algoritmo genético para nuestro problema, su longitud y codificación?

- Un individuo en nuestro algoritmo genético (y todos, realmente) representa un subconjunto de variables predictoras en un problema de clasificación. Voy a usar una **codificación binaria**, donde cada bit representa la presencia (1) o ausencia (0) de una variable en el subconjunto. La longitud del individuo será igual al número total de variables disponibles. Por ejemplo, si tengo 10 variables predictoras, cada individuo será un vector binario de 10 elementos.

2: Evaluación de los individuos

¿Cuál es la función objetivo a optimizar y cómo evaluarás cada individuo?

- La función objetivo a optimizar puede variar, pero puede ser bastante acertado el medir la precisión de la propia clasificación del subconjunto de variables que representan al individuo o solución. Es decir, se evaluaría cada individuo entrenando un modelo de clasificación con las variables seleccionadas y se calcularía la precisión del modelo en un conjunto de datos de prueba.

3: Operador de cruce

Define un operador de cruce y garantiza que todas las nuevas soluciones sean válidas.

- Esto me recuerda bastante a la asignatura de Investigación Operativa, donde explorábamos diferentes espacios de búsqueda para distintos algoritmos y, en ese proceso, íbamos verificando si las soluciones que íbamos obteniendo eran válidas o no. Bueno, realmente lo hacíamos incluso con la solución aleatoria inicial.

Bien, como operador de cruce, he investigado un poco y he visto muchas técnicas diferentes, pero me ha gustado el de utilizar el cruce de un punto. Simplemente sería seleccionar un punto al azar a lo largo de la longitud del individuo e intercambiar todos los bits después de ese punto entre dos individuos (algo similar existe en las transparencias también). Esto debería de garantizar que las nuevas soluciones sean combinaciones válidas de las variables predictoras.

4: Operador de mutación

Propón un operador de mutación que garantice que cada solución mutada sea válida.

- Simple, pero interesante y eficaz puede ser el operador de la mutación bit a bit. Con una probabilidad baja, cada bit en el individuo puede hacer swap (de 0 a 1 o de 1 a 0). Esto asegura que el individuo mutado siga siendo un subconjunto válido de variables predictoras, ergo la solución seguiría siendo válida.

5: Criterio de parada del algoritmo genético

Define un criterio de parada para el algoritmo genético.

El algoritmo genético se detendrá cuando se cumpla uno de los siguientes criterios:

- 1) Se alcanza un número máximo de generaciones.
- 2) La mejora en la función objetivo entre generaciones consecutivas es menor a un umbral predefinido, indicando convergencia. Como nota, este tipo de umbrales o *tolerancias*, hemos estudiado bastante en la asignatura de Computación Científica, en el contexto de algoritmos iterativos (Newton, métodos del punto fijo...etc)
- 3) Implosione el Universo.

5.2. Análisis del Dataset ACB-BasketballGamePrediction-2012-2013

¿Cuál es el problema de clasificación que recoge el dataset ACB-BasketballGamePrediction-2012-2013.arff, su origen, la variable clase y sus valores, y el origen de las variables predictoras?

El dataset `ACB-BasketballGamePrediction-2012-2013.arff` recoge datos estadísticos de partidos de baloncesto de la liga ACB durante la temporada 2012-2013. Cada partido se transforma en dos instancias-casos, una por cada equipo participante, etiquetadas como **victoria** o **derrota**.

- **Origen del Problema:** El dataset fue creado por Gorka Elgezabal, ex-alumno de la misma facultad donde estudiamos; Facultad de Informática de Donostia, UPV-EHU, como parte de un proyecto de fin de grado. Expone casos de partidos de la liga regular de la ACB, sin incluir playoffs.
- **Variable Clase y sus Valores:** La variable de clase es el resultado del partido, con dos posibles valores: victoria o derrota.
- **Origen de las Variables Predictoras:** Las variables predictoras son estadísticas de cada equipo por partido; puntos anotados, porcentajes de tiro, rebotes, asistencias...etc. Estos datos probablemente fueron recopilados a través de métodos como el web scraping o el uso de APIs de estadísticas deportivas (no se indica).
- **Generación de Casos en la Base de Datos:** Cada instancia, expresa estadísticas de un equipo en un partido específico. Cada partido genera dos instancias (una por equipo - esto es nuevo hasta ahora, donde la relación era 1 a 1), estas instancias aparecen de forma contigua en el fichero.

5.2.1. Variables predictoras

Después de eliminar las variables código temporada, código partido, fecha, hora, y teniendo en cuenta que una de las 25 variables restantes es la variable de clase, ¿cuántos subconjuntos distintos de variables predictoras existen?

Tras eliminar las variables, nos quedamos con 24 variables predictoras. El número de subconjuntos

distintos de estas variables predictoras que se pueden formar es 2^{24} . Podemos calcular el total para n variables es 2^n . Así que, $2^{24} = 16,777,216$, lo que significa que existen 16,777,216 posibles combinaciones o soluciones en el espacio de búsqueda de éste problema de selección de variables.

5.3. Análisis de WrapperSubsetEval en Weka

Parámetros de WrapperSubsetEval y elección del clasificador

Describe brevemente los parámetros de WrapperSubsetEval en Weka y discute la posibilidad de usar un clasificador con más coste de CPU como las redes neuronales.

WrapperSubsetEval en Weka evalúa subconjuntos de atributos mediante un clasificador y la validación cruzada o Cross Validation. Si analizamos los parámetros clave:

- **seed:** Semilla para las divisiones del Cross Validation, asegura reproducibilidad.
- **folds:** Número de pliegues en la Cross Validation para estimar precisión.
- **classifier:** Bastante 'self-explanatory'; el clasificador que se usará, como podría ser Naive Bayes etc.
- **evaluationMeasure:** Medida que se utiliza para evaluar el rendimiento al combinar atributos.
- **Otros parámetros** incluyen **IRClassValue**, **doNotCheckCapabilities**, y **threshold**, cada uno ajustando aspectos específicos del proceso de evaluación.

Sobre el uso de clasificadores más costosos como las redes neuronales: aunque pueden ofrecer una evaluación más precisa y, tal y como hemos visto en clase y el cual ha sido el motivo del 'delay' histórico en su uso; tienen un alto coste de CPU que puede hacer que el proceso sea más lento, especialmente cuando hay muchos subconjuntos a evaluar. Naive Bayes es una opción más eficiente si lo que buscamos es rapidez.

5.4. Configuraciones y Resultados

He realizado varias pruebas con el método de genetic-search en WEKA, variando los parámetros de tamaño de población, número de generaciones, y probabilidades de cruce y mutación. Veamos algunos de ellos:

1. Primera configuración:

- Tamaño de población: 20
- Número de generaciones: 20
- Probabilidad de cruce: 0.6
- Probabilidad de mutación: 0.033

2. Segunda configuración:

- Tamaño de población: 20
- Número de generaciones: 20
- Probabilidad de cruce: 0.6
- Probabilidad de mutación: 0.1

3. Tercera configuración:

- Tamaño de población: 10
- Número de generaciones: 5
- Probabilidad de cruce: 0.6
- Probabilidad de mutación: 0.1

5.4.1. Análisis de las pruebas

-
1. **Primera Ejecución (Probabilidad de Mutación: 0.033):** La baja probabilidad de mutación ha producido una convergencia lenta, que no tiene porqué ser malo, ya que a su vez también ha promovido una exploración más exhaustiva. Claro, esto también conlleva riesgo de converger a óptimos locales y 'quedarnos atascados ahí hasta que 'le peguemos una patada a la pelota y caiga en otra zona" (El año pasado en Investigación Operativa, el profesor nos ponía este tipo de similes para comprender cómo la pelota iba cayendo a los mínimos, cómo teníamos que perturbar la solución en la que nos habíamos atascado si queríamos saltar a otra zona etc - me gustó bastante y se me han quedado grabados.
 2. **Segunda Ejecución (Probabilidad de Mutación: 0.1):** Una mayor probabilidad de mutación ha hecho que se mantenga la diversidad por más tiempo, lo cual favorece la exploración y evita óptimos locales - 'cada poco, pam, patada a la pelota y a buscar en esa zona'.
 3. **Tercera Ejecución (Tamaño de Población: 10, Generaciones: 5):** Una población menor, ergo menos generaciones, hacen claramente que se limite mucho más la exploración del espacio de búsqueda. Lo cual, lógicamente nos puede dar soluciones que, pese a ser válidas (por los sistemas y operadores definidos), pueden estar lejos de ser óptimas.

5.4.2. Óptimo Global

No hay garantías de que la mejor solución encontrada sea el óptimo global, y éste es un poco el conundrum de los problemas que requieren de heurísticos avanzados, se debe a la naturaleza heurística de los algoritmos genéticos y la posibilidad de converger a óptimos locales, sobre todo con poblaciones pequeñas o mutaciones limitadas. Encontrar el óptimo global en problemas complejos con grandes espacios de búsqueda, puede resultar imposible en un tiempo 'razonable'.

5.4.3. Otros Metaheurísticos

Aparte de los genéticos, he visto que WEKA ofrece otros metaheurísticos como la búsqueda de tabú y la optimización de enjambres de partículas. Otros conocidos incluyen la búsqueda de hormigas y el recocido simulado, que al parecer resultan muy útiles para espacios de búsqueda complejos.

Nota personal

Tengo muchísimas ganas de aprender más sobre ésto, que algoritmos que simulan o se basan en obsevación de patrones recurrentes en la naturaleza puedan llegar a ser tremadamente útiles... me parece fascinante. Por ejemplo el algoritmo de la colonia de hormigas; que utiliza agentes, como hormigas virtuales, que dejan feromonas en sus caminos en búsqueda de la comida para marcar buenas soluciones. Las rutas con más feromonas atraen a más hormigas y así se genera un 'pathing', ayudando a encontrar las mejores soluciones con el tiempo. Es sencillamente maravilloso.

5.5. Implementación feature selection

Tras cotillear un poco, he visto el siguiente script: https://github.com/prakhargurawa/Feature-Selection-Using-Genetic-Algorithm/blob/main/genetic_algorithm.py

```
# -*- coding: utf-8 -*-
'''
Created on Tue Feb  2 12:57:22 2021

@author: prakh
'''

# import necessary libraries
import numpy as np
import random
import matplotlib.pyplot as plt

# GA for minimisation
def GA(f, init, nbr, crossover, select, popsize, ngens, pmut):
    history = []
    # make initial population, evaluate fitness, print stats
    pop = [init() for _ in range(popsize)]
```

```

popfit = [f(x) for x in pop]
history.append(stats(0, popfit))
for gen in range(1, ngens):
    # make an empty new population
    newpop = []
    # elitism : directly select the best candidate to next population as it is
    bestidx = min(range(popszie), key=lambda i: popfit[i])
    best = pop[bestidx]
    newpop.append(best)
    while len(newpop) < popszie:
        # select and crossover
        p1 = select(pop, popfit)
        p2 = select(pop, popfit)
        c1, c2 = crossover(p1, p2)
        # apply mutation to only a fraction of individuals : pmut is ↴
        ↪hyperparameter
        if random.random() < pmut:
            c1 = nbr(c1)
        if random.random() < pmut:
            c2 = nbr(c2)
        # add the new individuals to the population
        newpop.append(c1)
        # ensure we don't make newpop of size (popszie+1) -
        # elitism could cause this since it copies 1
        if len(newpop) < popszie:
            newpop.append(c2)
    # overwrite old population with new, evaluate, do stats
    pop = newpop
    popfit = [f(x) for x in pop]
    history.append(stats(gen, popfit))
    bestidx = np.argmin(popfit)
return popfit[bestidx], pop[bestidx], history

def stats(gen, popfit):
    # let's return the generation number and the number
    # of individuals which have been evaluated
    return gen, (gen+1) * len(popfit), np.min(popfit), np.mean(popfit), np.
    ↪median(popfit), np.max(popfit), np.std(popfit)

```

5.6. Aplicaciones con Algoritmos Genéticos

He estado mirando un poco las aplicaciones de algoritmos genéticos en el diseño y optimización de prototipos virtuales que se han mencionado, como los simuladores de **Genetic Cars 2** y **Evolution** de Keiwan.

5.6.1. Observaciones en Genetic Cars 2

En **Genetic Cars 2**, se puede apreciar cómo variaciones genéticas en la forma y tamaño de las ruedas, así como en la densidad del chasis, afectan la capacidad de los coches para atravesar terrenos más irregulares. La aplicación nos permite ajustar sus parámetros como tasas de mutación y clones de élite, es impresionante ver cómo algunos individuos evolucionan hacia diseños más eficientes. ¡Parece magia!

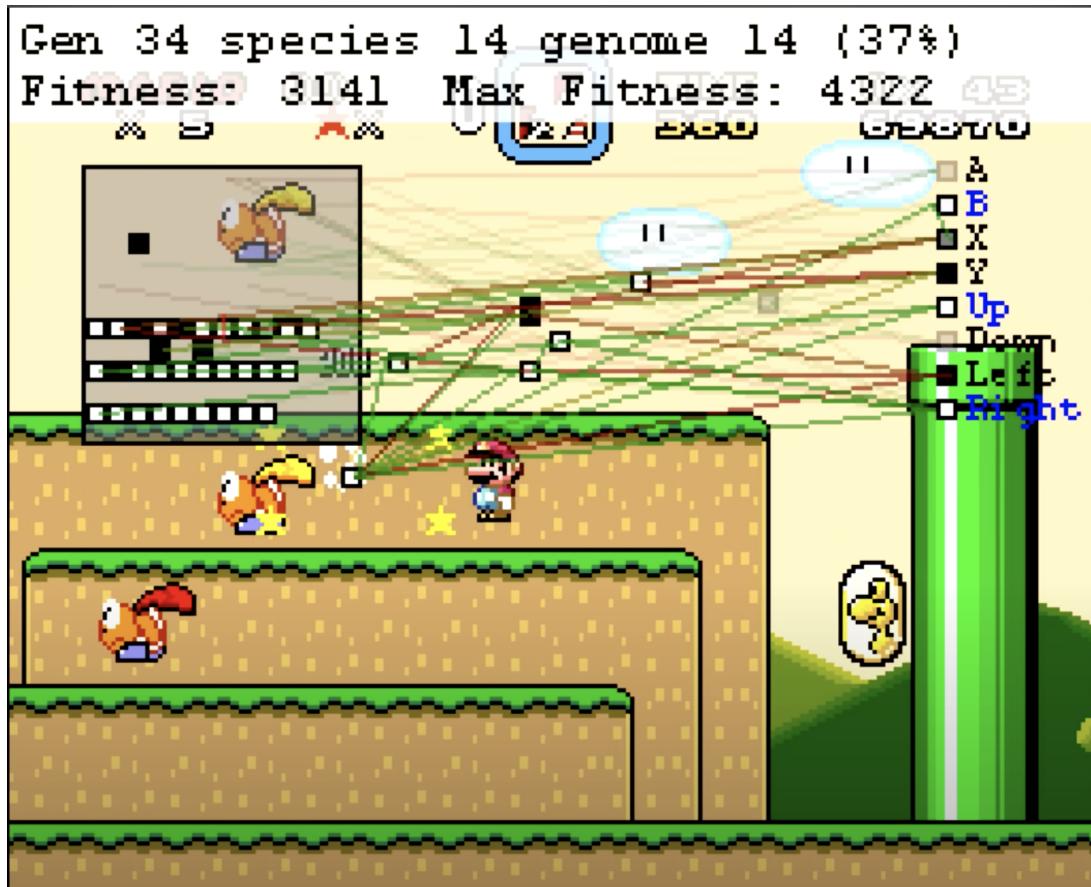
5.6.2. Experimentos en Evolution

Por otro lado, **Evolution** es una experiencia diferente; nos permite crear criaturas con articulaciones y músculos. La evolución está más orientada en este caso hacia acciones / habilidades específicas como correr o saltar. La aplicación de redes neuronales junto a algoritmos genéticos resulta en una evolución mucho más compleja y difícil de comprender, la verdad.

Estos experimentos demuestran la versatilidad y potencia de los algoritmos genéticos en la resolución de problemas de diseño complejos en entornos virtuales.

5.6.3. Curiosidad

MarI/O



Por supuesto, cómo no, he de aprovechar y hacer un pequeño hincapié en algo relacionado a los videojuegos; en este caso un proyecto llamado MarI/O. Lo conocía desde ya tiempo y se puede ver un video cortito que está accesible en YouTube, titulado [MarI/O - Machine Learning for Video Games](#) de SethBling. Este video muestra una red neuronal que aprende a jugar a **Super Mario World** (buenísimo juego). El sistema, utilizando algoritmos genéticos, evoluciona sus estrategias y movimientos en el juego sin necesidad de una programación pre-definida para cada situación, el personaje va aprendiendo a moverse en el mundo a través de las generaciones. Es increíble ver cómo el algoritmo mejora y adapta sus tácticas a medida que avanza el progreso. Para la gente que le gusten los videojuegos, es una muestra bastante impresionante de la capacidad y flexibilidad del Machine Learning.

*«By far, the greatest danger of Artificial Intelligence
is that people conclude too early that they understand
it.»*

–Eliezer Yudkowsky.

26 de Diciembre de 2023