UD3.- Desarrollo de aplicaciones de Inteligencia Artificial

PIA

Curso de especialización de Inteligencia Artificial y Big Data (IABD)

1. *Evaluación de los resultados*

La evaluación de un modelo de aprendizaje en Inteligencia Artificial forma parte de su ciclo de vida. Gracias a la evaluación podemos cuantificar la calidad del modelo y su capacidad de predicción. En la mayoría de las ocasiones existe una realimentación entre la fase de evaluación y la de diseño, tal que una evaluación con resultados insatisfactorios da pie a modificaciones sobre el diseño para mejorar la calidad.

Los métodos de evaluación dependen del tipo de modelo; de esta forma, los modelos de aprendizaje automático como, por ejemplo, métodos de regresión lineal y métodos de clustering emplean unos métodos de evaluación distintos a los empleados en modelos de aprendizaje supervisado, principalmente métodos de clasificación a partir de un conjunto de ejemplos etiquetados previamente.

La biblioteca Scikit-learn es una biblioteca, libre, para el lenguaje de programación Python e incluye un amplio repertorio de herramientas con métodos de Machine Learning e Inteligencia Artificial. Dicha biblioteca facilita la evaluación de modelos en todas sus variantes y será la que emplearemos.

En la documentación oficial de Scikit-learn y, en particular, en el apartado de modelos de evaluación (https://scikit-learn.org/stable/modules/model_evaluation.html) se ofrece una descripción detallada de las diferentes métricas soportadas para cuantificar la calidad de las predicciones de un modelo. Dichas métricas están especialmente divididas en tres categorías:

- Métricas de clasificación
- Métricas de clustering
- Métricas de regresión

En concreto las métricas soportadas por Scikit-learn para problemas de clasificación son:

'accuracy'	metrics.accuracy_score
'balanced_accuracy'	metrics.balanced_accuracy_score
'top_k_accuracy'	metrics.top_k_accuracy_score
'average_precision'	metrics.average_precision_score
'neg_brier_score'	metrics.brier_score_loss
'f1'	metrics.f1_score
'f1_micro'	metrics.f1_score
'f1_macro'	metrics.f1_score
'f1_weighted'	metrics.f1_score
'f1_samples'	metrics.f1_score
'neg_log_loss'	metrics.log_loss
'precision' etc.	metrics.precision_score
'recall' etc.	metrics.recall_score
'jaccard' etc.	metrics.jaccard_score
'roc_auc'	metrics.roc_auc_score
'roc_auc_ovr'	metrics.roc_auc_score
'roc_auc_ovo'	metrics.roc_auc_score
'roc_auc_ovr_weighted'	metrics.roc_auc_score
'roc_auc_ovo_weighted'	metrics.roc_auc_score

Con el listado anterior, puedes hacerse una idea general de un conjunto representativo de métricas para la evaluación de modelos de clasificación en aprendizaje supervisado. En Scikit-learn podemos encontrar todas estas métricas dentro de `sklearn.metrics`. Puedes encontrar más información en el siguiente enlace.

https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics

Los modelos de aprendizaje supervisado parten de un conjunto de muestras (Dataset), siendo cada muestra una tupla formada por un conjunto de valores correspondientes a las variables de entrada, y una salida o clase a la que pertenece la muestra. Es decir, el modelo aprende a partir de un conjunto de ejemplos catalogados correctamente a priori. Por ejemplo, un Dataset muy popular (ver Figura 1) con el que los diseñadores prueban sus modelos es *Iris*, disponible en la mayor parte de plataformas de IA (Scikit learn, no es una excepción) y contiene 50 muestras de cada una de tres especies (150 ejemplos en total) de la flor Iris (iris setosa, iris virginica, iris versicolor).

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0

Figura 1. Dataset de Iris como DataFrame

En este conjunto de datos las variables de entrada son: longitud del sépalo, anchura del sépalo, longitud del pétalo y anchura del pétalo. Por tanto, cada tupla está formado por cuatro valores de entrada, uno para cada variable de las mencionadas anteriormente, y la clase a la que pertenece (setosa, virginica, versicolor). A partir de los 150 ejemplos del Dataset, el modelo aplicado debe ser capaz de clasificar una nueva flor a partir de los cuatro valores de entrada. Precisamente, los métodos de evaluación nos ayudarán a medir la calidad de esa predicción.

Podemos descargar este dataset como un DataFrame de la siguiente forma:

```
from sklearn import datasets
iris = datasets.load_iris(as_frame=True)
```

1.1. Validación cruzada

A la hora de validar un modelo es muy habitual emplear una validación cruzada y, sobre ésta, aplicar las diferentes métricas que hemos nombrado anteriormente y que veremos en detalle en las próximas secciones.

En primer lugar, será necesario dividir en conjunto de datos inicial o Dataset en dos subconjuntos: conjunto de datos de entrenamiento (aprendizaje) y conjunto de datos para las pruebas (clasificación). En cuanto al porcentaje de ejemplos que corresponde a cada uno del conjunto inicial hay diversidad de opiniones, pero en la mayoría de los casos, los autores optan por un 70% de ejemplos para el entrenamiento y un 30 % para las pruebas, u 80%-20%.

Los algoritmos de aprendizaje aplicados sobre el conjunto de entrenamiento generan como salida un modelo con capacidad para clasificar los ejemplos del conjunto de pruebas. La evaluación del modelo se lleva a cabo en esta segunda fase, con la aplicación de métricas que determinan la exactitud, precisión o sensibilidad entre otros factores.

La proporción de ejemplos para el entrenamiento y las pruebas es importante, pero también lo es cuáles de ellos son seleccionados. Para evaluar correctamente un modelo es necesario un banco de pruebas con varias iteraciones. En cada una de esas iteraciones se hace una selección diferente de ejemplos para los conjuntos de entrenamiento y pruebas, y se aplican las métricas para obtener los marcadores. Los marcadores finales pueden obtenerse mediante la media aritmética de los obtenidos en cada una de las iteraciones.

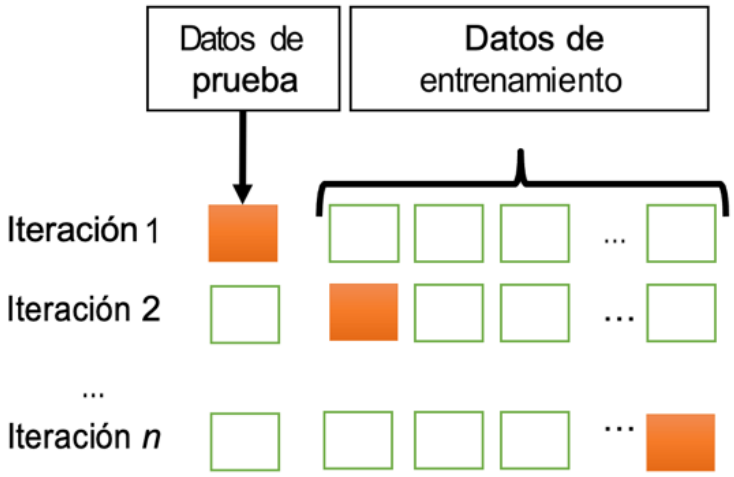


Figura 2. Validación cruzada

La validación cruzada se preocupa por la selección de ejemplos en cada una de estas iteraciones, y hay diversas variantes (se verán en la Sección 1.1.1). Una de ellas, de las más sencillas, es elegir al azar en cada iteración los ejemplos que forman parte de los conjuntos de entrenamiento y prueba. Sin embargo, en otras ocasiones se opta por la no repetición entre iteraciones. Para ello, se trocea el conjunto de datos en tantas partes como iteraciones o pruebas se pretendan realizar y, en cada iteración, se elige un fragmento diferente para el conjunto de pruebas. La Figura 2 muestra gráficamente un ejemplo de validación cruzada, en donde el conjunto de ejemplos empleado para la clasificación varía en cada iteración y sin repetición.

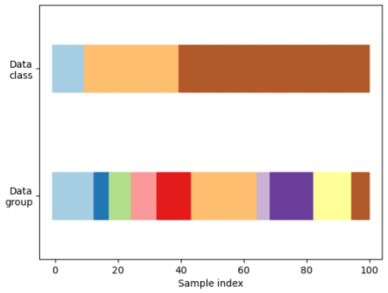
1.1.1. Validación Cruzada con Python

Scikit-learn también incluye herramientas para dividir el conjunto de datos mediante la técnica de validación cruzada. En la Sección "Validación cruzada" que puedes encontrar en el enlace https://colab.research.google.com/drive/1R20d03h4598Z0wHSJU6efpxYt5XNEF38, se verá un ejemplo haciendo uso del conjunto de datos Iris.

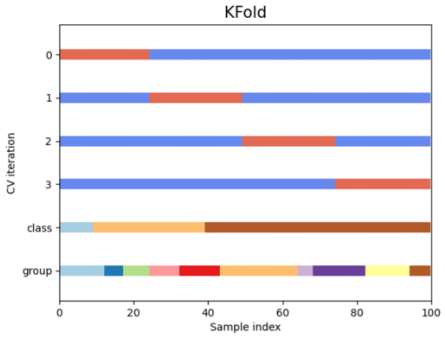
En concreto, la función que utilizaremos para hacer Validación Cruzada será *cross_val_score.*

```
from sklearn.model_selection import cross_val_score

clf = svm.SVC(kernel='linear', C=1, random_state=42)

#cv será el número de subgrupos que se definirán

scores = cross_val_score(clf, X, y, cv=5)

scores
```

Con el fin de ilustrar uno de los problemas de la Validación Cruzada podemos ver un dataset que tiene 100 puntos de datos de entrada generados aleatoriamente, 3 clases divididas de manera desigual en los puntos de datos y 10 "grupos" divididos de manera uniforme en los puntos de datos. Los grupos corresponden por ejemplo al valor de una variable, en este caso habrá diez valores diferentes

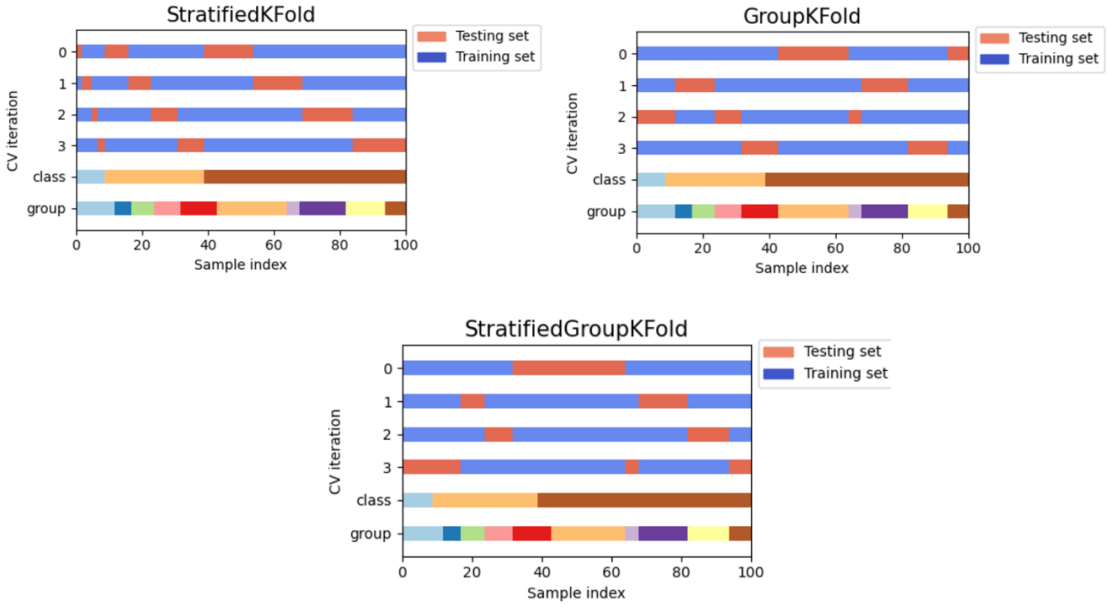


Para la ilustración realizaremos una Validación Cruzada con 4 iteraciones. En cada iteración, se visualizarán los índices elegidos para entrenar (azul) y para testear (rojo).



Como se puede ver, de forma predeterminada con KFold, el iterador de validación cruzada no tiene en cuenta ni la clase ni el grupo de puntos de datos. Podemos cambiar esto usando:

- StratifiedKFold para preservar el porcentaje de muestras para cada clase.
- GroupKFold para asegurarse de que el mismo grupo no aparecerá en dos subconjuntos de test diferentes.
- StratifiedGroupKFold para mantener la restricción de GroupKFold al intentar devolver subconjuntos de test estratificados.



En Python podemos utilizar las diferentes configuraciones del iterador de la siguiente forma.

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold

#KFold
print("KFold")
clf = svm.SVC(kernel='linear', C=1, random_state=42)
k_fold = KFold(n_splits=5) #defino un KFold con 5 iteraciones
scores = cross_val_score(clf, X, y, cv=k_fold)
print(scores)
print("%0.2f accuracy with a standard deviation of %0.2f\n" % (scores.mean(),
scores.std()))

#StratifiedKFold
print("StratifiedKFold")
stratified_k_fold = StratifiedKFold(n_splits=5) #defino un StratifiedKFold
con 5 iteraciones
scores = cross_val_score(clf, X, y, cv=stratified_k_fold)
print(scores)
print("%0.2f accuracy with a standard deviation of %0.2f\n" % (scores.mean(),
scores.std()))

#GroupKFold
print("GroupKFold")
#si tuvieramos un dataframe se podría obtener de la siguiente forma, en
nuestro caso será groups (columna añadida artificialmente)
#groups = X['your_group_name'] # or pass your group another way
groups =  groups # or pass your group another way
group_k_fold = GroupKFold(n_splits=5) #defino un StratifiedKFold con 5
iteraciones
scores = cross_val_score(clf, X, y, cv=group_k_fold, groups= groups)
print(scores)
```

```
print("%0.2f accuracy with a standard deviation of %0.2f\n" % (scores.mean(),
scores.std()))
```

1.2. Evaluación de modelos de clasificación

1.2.1. Matriz de confusión

La matriz de confusión no puede ser considerada una métrica como tal, pero es una herramienta fundamental para evaluar y optimizar los modelos de clasificación, ya que ayuda a profundizar en el tipo de error que el modelo está cometiendo y a comprender otras métricas que se emplean.

Una matriz de confusión se representa mediante una tabla en la que las columnas corresponden a los valores reales y las filas a los valores predichos. En el caso particular de un modelo de clasificación binaria, esta tabla se confeccionará de la siguiente forma:

		Valores Reales	
Valores Predichos	Positivo (A)	Positivo (A)	Negativo (B)
		TP	FP
	Negativo (B)	FN	TN

Supongamos un modelo de clasificación binaria, es decir, aquel en el que tan solo se pueden asignar dos clases diferentes (A y B) para cada ejemplo de entrada. Supongamos también un modelo cuyo objetivo es la clasificación de un tipo de cáncer como maligno (A) o Benigno (B). En función del valor real y el valor predicho por el modelo, podemos encontrarnos cuatro situaciones posibles (ver tabla anterior):

- Verdadero Positivo (True Positive, TP): el cáncer realmente es maligno (clase A) y ha sido clasificado como tal.
- Falso Positivo (False Positive, FP): el cáncer realmente es benigno (B) y el modelo lo ha clasificado como maligno (A).
- Falso Negativo (False Negative, FN): el cáncer realmente es maligno (A) y el modelo lo clasifica como benigno (B).
- Verdadero Negativo (True Negative, TN): el cáncer es benigno (B) y el modelo lo clasifica como benigno (B).

Por tanto, los Verdaderos (positivos y negativos) representan casos que han sido correctamente clasificados, mientras que los Falsos (positivos y negativos) representan errores de clasificación. De esta forma, nos interesa que la cantidad de ejemplos contabilizados en la diagonal principal (marcada en color verde en la anterior tabla) sea mucho mayor a los contabilizados en diagonal secundaria (marcados en color rojo en la anterior tabla), lo que querrá decir que nuestro modelo se aproxima a un diagnóstico fiable.

En función de los valores obtenidos, se pueden realizar modificaciones sobre el modelo con la intención de reducir el número de falsos positivos o falsos negativos (o incluso ambos). Dependiendo del contexto en el que esté enmarcado el modelo, será más crítico reducir uno u otro.

Por ejemplo, en el contexto médico que planteábamos anteriormente, los casos especialmente críticos corresponderían con los Falsos Negativos (FN), es decir, casos de tumores malignos diagnosticados como benignos, lo cual repercutiría gravemente en la salud del paciente.

Por otro lado, en un contexto jurídico en el que un sospechoso puede ser clasificado como culpable (A) o inocente (B), los casos más críticos los representan los Falsos Positivos (FP), es decir, una persona inocente que ha sido catalogada como culpable, con el riesgo de terminar en prisión de forma injusta.

El siguiente fragmento de código muestra un ejemplo escrito en Python con la librería Scikit-learn en donde se define un vector básico de clases con valores reales (`y_real`) y otro con los valores predichos (`y_pred`). Ambos vectores son empleados para calcular la matriz de confusión.

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

import matplotlib.pyplot as plt
import seaborn as sns

y_real = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
y_pred = [1, 1, 1, 1, 0, 1, 1, 0, 0, 0]

cm = confusion_matrix(y_real, y_pred)
print (cm)
```

Si has ejecutado el anterior código verás que no es muy ilustrativo, aunque contiene toda la información necesaria. Si queremos generar una matriz de confusión más visual se puede crear un heatmap con esta matriz de confusión.

```
#Creación de Figura
ax= plt.subplot()
sns.heatmap(cm, annot=True, ax=ax, cmap='Greens');

# labels, title and ticks
ax.set_xlabel('Valores Predichos');ax.set_ylabel('Valores Reales');
ax.set_title('Confusion Matrix');ax.xaxis.set_ticklabels([1, 0]);
ax.yaxis.set_ticklabels([1, 0]);
plt.show()
```

Por otro lado, para aquellos modelos de clasificación no binaria, en los que un ejemplo de entrada puede ser clasificado en n clases diferentes, también es posible la representación de un matriz de confusión. En este caso, las columnas representan el valor verdadero o la clase correcta a la que corresponde el ejemplo, y las filas las clases predichas. La siguiente tabla muestra una representación general de este tipo de matrices.

		Clases verdaderas			
		A	B	...	X
Clases Predichas	A				
	B				
	...				
	X				

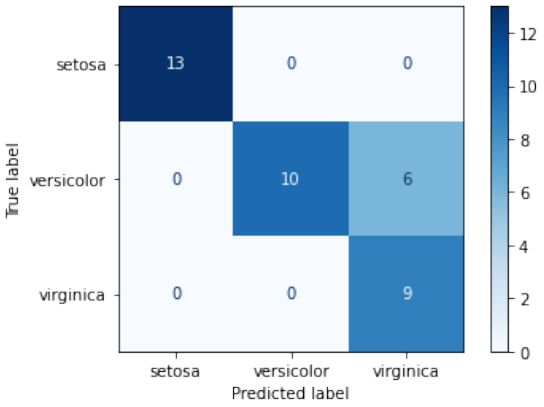
La diagonal principal representa los casos que han sido correctamente clasificados, es decir, la clase predicha corresponde con la clase real del ejemplo de entrada. El resto de las celdas representan casos de errores en la clasificación.

Un fragmento de código en el que se lleva a cabo una clasificación de estas características es el que se muestra a continuación. También podrás ejecutarlo en la Sección "Ejemplo de uso de matriz de confusión multiclase" que puedes encontrar en el documento compartido: https://colab.research.google.com/drive/1R20d03h4598Z0wHSJU6efpxYt5XNEF38:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay

#Importación de Iris Dataset
iris = datasets.load_iris()
#Conjunto de ejemplos
X = iris.data
#Clases
y = iris.target
#nombre de las clases
class_names = iris.target_names

# División del DataSet en dos cojuntos: entrenamiento y test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Usamos como clasificador Support Vector Classification
classifier = svm.SVC(kernel='linear', C=0.01).fit(X_train, y_train)

disp = ConfusionMatrixDisplay.from_estimator(
      classifier,
      X_test,
      y_test,
```

```
        display_labels=class_names,
        cmap=plt.cm.Blues
)
plt.show()
```

Como resultado de esta ejecución obtenemos una matriz de confusión con el siguiente aspecto.



1.2.2. Métricas para la evaluación de los resultados

Teniendo en cuenta los valores de una matriz de confusión, se pueden establecer diferentes métricas de evaluación.

		Valores Reales	
Valores Predichos	Positivo (A)	Positivo (A)	Negativo (B)
		TP (VP)	FP (FP)
	Negativo (B)	FN (FN)	TN (VN)

Recordamos el significado de las siglas:

- TP/VP: True Positive/Verdadero Positivo
- FP/FP: False Positive/Falso Positivo
- FN/FN: False Negative/Falso Negativo
- TN/VN: True Negative/Verdadero Negativo

El código correspondiente a las diferentes métricas de evaluación que vamos a ver a continuación lo puedes encontrar en la sección "Métricas de evaluación" en el documento compartido: https://colab.research.google.com/drive/1R20d03h4598Z0wHSJU6efpxYt5XNEF38

Exactitud (accuracy)

La exactitud es el ratio entre las predicciones correctas: suma de verdaderos (positivos y negativos) y las predicciones totales. Dicho de otra forma, de todos los ejemplos ¿cuántos se predijeron correctamente? En el caso de una clasificación binomial el cálculo es tan sencillo como:

$$\frac{VP+VN}{TOTAL}$$

En el caso planteado en la siguiente tabla, el cálculo de la exactitud del modelo, sería el siguiente:

		Valores Reales	
Valores Predichos	Positivo (A)	Positivo (A)	Negativo (B)
		8	2
	Positivo (B)	1	9

$$\text{Accuracy} = \frac{8+9}{8+9+2+1} = 0{,}85$$

De forma general, la exactitud de un modelo multiclase podría representarse de la siguiente forma:

$$accuracy\_score(y, \hat{y}) = \frac{1}{n}\Sigma_{i=0}^{n-1}(\hat{y} = \hat{y}_i)$$

Donde n representa el número de muestras (o ejemplos) totales, y la clase real e ŷ la clase predicha. Veamos cómo representar el ejemplo anterior en Python y realizar el cálculo de la exactitud en mediante *sklearn*:

```
from sklearn.metrics import accuracy_score

y_pred=['A','A','A','A','A','A','A','A','B','A','A','B','B','B','B','B','B'
,'B','B','B']

y_true=['A','A','A','A','A','A','A','A','A','B','B','B','B','B','B','B','B'
,'B','B','B']

print(accuracy_score(y_true, y_pred))
```

Precisión

La precisión determina el grado de acierto en la clase relevante. Suponiendo que la clase relevante sea A (tumor maligno en los ejemplos planteados anteriormente), la precisión sería la proporción entre los verdaderos positivos (TP) y el total de casos positivos (primera fila de la tabla). Con ello podemos saber cuántas de las muestras que fueron catalogadas como casos positivos eran realmente positivas:

$$\frac{VP}{VP+FP}$$

El cálculo para el caso planteado en la tabla de ejemplo sería:

$$\text{precission} = \frac{8}{8+2} = 0{,}8$$

```
from sklearn.metrics import precision_score

y_pred
=['A','A','A','A','A','A','A','A','B','A','A','B','B','B','B','B','B','B','B','B']

y_true
=['A','A','A','A','A','A','A','A','A','B','B','B','B','B','B','B','B','B','B','B']

print(precision_score(y_true, y_pred, labels=['A', 'B'], pos_label='A', average="binary"))
```

La precisión también podría calcularse para la clase secundaria mediante la relación entre verdaderos negativos (VN) y la suma de VN + FN:

$$\frac{VN}{VN+FN}$$

La interpretación del dato en este caso es similar al anterior: cuántas de las muestras negativas reales han sido realmente detectadas. En el ejemplo de la clasificación de tumores, este valor serviría para conocer la proporción de tumores benignos detectados.

Exhaustividad (Recall)

Métrica también conocida como sensibilidad (sensitivity) o Ratio de Verdaderos Positivos (TPR, True Positive Rate), es un ratio entre los verdaderos positivos y la suma de los verdaderos positivos y los falsos negativos. El valor obtenido nos ofrece una idea clara sobre cuántos de los casos catalogados como positivos fueron predichos correctamente. De vuelta al ejemplo anterior, la exhaustividad determinaría cuántos de los tumores clasificados como malignos, son realmente malignos. ¿Cuál sería el número real de tumores malignos? La suma de TP (tumores malignos detectados correctamente) y los FN (tumores malignos no detectados).

$$\frac{VP}{VP+FN}$$

El cálculo para el caso planteado en la tabla de ejemplo sería:

$$\text{recall} = \frac{8}{8+1} = 0{,}88$$

```
from sklearn.metrics import recall_score

y_pred=['A','A','A','A','A','A','A','A','B','A','A','B','B','B','B','B','B','B','B','B']
```

```
y_true=['A','A','A','A','A','A','A','A','A','B','B','B','B','B','B','B','B'
,'B','B','B']

print(recall_score(y_true, y_pred, labels=['A', 'B'], pos_label='A',
average="binary"))
```

Pérdida Logarítmica (Log Loss)

La pérdida logarítmica es una puntuación única que representa la ventaja del clasificador sobre una predicción aleatoria. La pérdida logarítmica mide la incertidumbre del modelo comparando las probabilidades de sus resultados con los valores conocidos (verdad de base). El objetivo es minimizar la pérdida logarítmica para el modelo en su conjunto, cuanto menor sea el valor, menor será la incertidumbre y mejor será la capacidad del modelo para clasificar correctamente las muestras de entrada. El cálculo de la pérdida logarítmica sería el siguiente:

$$LogLoss = -\frac{1}{n}\Sigma_{i=1}^{n}[y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)]$$

Donde n representa el número de muestras (o ejemplos) totales, y la clase real e ŷ la clase predicha. En Python y la librería sklearn el cálculo sería tan sencillo como se muestra a continuación:

```
from sklearn.metrics import log_loss
y_pred =[0,0,0,0,0,0,0,0,1,0,0,1,1,1,1,1,1,1,1,1]
y_true =[0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1]
print(log_loss(y_true, y_pred))
```

Es importante recalcar que la función log_loss trabaja únicamente con clases representadas numéricamente. Por ello, en el caso del ejemplo anterior cuyas clases se representan mediante las letras A y B, sería necesario una conversión numérica previa.

Valor-F o Media-F (F1-Score)

El Valor-F es la media ponderada de la precisión y la exhaustividad (*recall*). Dicho valor varía entre 0 y 1, siendo 1 el valor ideal y 0 el caso opuesto. El Valor-F puede interpretarse como una media armónica de la precisión y la exhaustividad. La contribución relativa de la precisión y la exhaustividad al Valor-F son iguales.

$$F - score = \frac{2}{\frac{1}{Precisin} + \frac{1}{Exhaustividad}} = \frac{TP}{TP + \frac{FP+FN}{2}}$$

El siguiente fragmento de código muestra cómo hacer uso de esta métrica en Python y sklearn.

```
from sklearn.metrics import f1_score
```

```
y_pred=['A','A','A','A','A','A','A','A','B','A','A','B','B','B','B','B','B'
,'B','B','B']
y_true=['A','A','A','A','A','A','A','A','A','B','B','B','B','B','B','B','B'
,'B','B','B']

print(f1_score(y_true, y_pred, average="weighted"))
```

Entre los atributos de la función f1_score merece especial mención **average**, cuyo valor repercute directamente en el cálculo del Valor-F. Veamos las alternativas que ofrece dicho atributo:

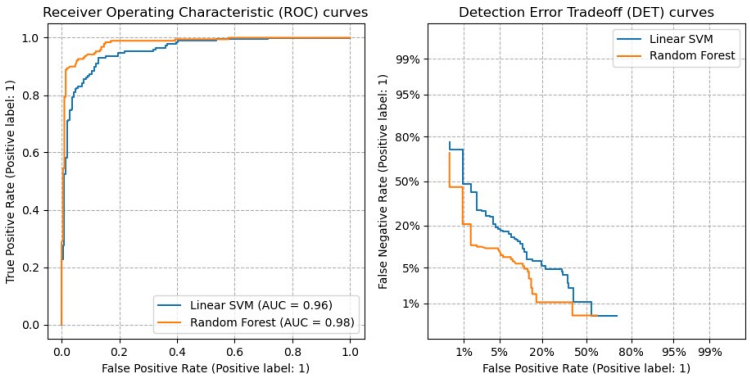
- **None**: se devuelven las puntuaciones de cada clase. En caso contrario, el tipo de promedio viene determinado por alguna de las siguientes opciones:
  - **binary**: sólo informa de los resultados de la clase especificada por el atributo pos_label. Esto es aplicable sólo si los objetivos (y_true, y_pred) son binarios.
  - **micro**: calcula las métricas globalmente contando el total de verdaderos positivos, falsos negativos y falsos positivos.
  - **macro**: calcula las métricas de cada etiqueta y encuentra su media no ponderada. Esto no tiene en cuenta el desequilibrio de las etiquetas.
  - **weighted**: calcula las métricas para cada etiqueta y encuentra su media ponderada por soporte (el número de instancias verdaderas para cada etiqueta). Esto altera la "macro" para tener en cuenta el desequilibrio de etiquetas; puede dar lugar a una puntuación F que no esté entre la precisión y la recuperación.
  - **samples**: calcula las métricas para cada instancia, y encuentra su promedio (sólo tiene sentido para la clasificación multietiqueta donde esto difiere de accuracy_score).

Curvas ROC

Una curva ROC (curva de característica operativa del recepto) es un gráfico que muestra el rendimiento de un modelo en base a la tasa de verdaderos positivos, y la tasa de falsos positivos. La forma de calcular dichas tasas es la siguiente:

- Tasa de verdaderos positivos (TPR) = VP / VP + FN (misma fórmula de la Exhaustividad)
- Tasa de falsos positivos (FPR) = FP / FP + VN

No solo es un tipo de gráfico con el que se puede apreciar a simple vista el rendimiento de un modelo, sino también la comparación entre modelos. Por ejemplo, modelos de aprendizaje y clasificación diferentes aplicados sobre un mismo *Dataset*, como muestra la siguiente figura.



Cuanto más cerca esté la curva al área de verdaderos positivos (cuanto más alta sea) mejor clasificará el modelo los datos de entrada.

AUC: área bajo la curva ROC

El área bajo la curva ROC toma un valor entre 0 y 1, en donde 1 representa una clasificación perfecta por parte del modelo. En tal caso, hay que prestar especial atención a que dicho resultado no sea consecuencia de un sobreajuste del modelo (overfitting). El sobreajuste de un modelo se produce cuando éste se ajusta casi a la perfección a los ejemplos del conjunto de entrenamiento (tiene capacidad para clasificarlos correctamente), pero no tiene capacidad para clasificar la mayoría de nuevos ejemplos (de los que no ha aprendido). En otras palabras, la capacidad de generalización del modelo es baja.

A medida que la curva ROC se acerque al área de verdaderos positivos, el área bajo la curva aumentará. El valor AUC de área representa la probabilidad de que el modelo clasifique correctamente un modelo de entrada. Por ejemplo, si AUC = 0.7, significa que el modelo ofrece un 70% de probabilidad de clasificar correctamente una muestra de entrada. Si queremos ver la curva ROC de una determinada predicción podemos utilizar la función RocCurveDisplay.

```
#Curva ROC y área AUC
import matplotlib.pyplot as plt
import numpy as np
from sklearn import metrics
y = y_test
pred = y_pred_1

#calculo el ratio de verdaderos positivos y falsos positivos.
fpr, tpr, _ = metrics.roc_curve(y, pred)

#con estos valores obtengo el valor para la AUC
roc_auc = metrics.auc(fpr, tpr)

#genero el gráfico
display = metrics.RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc,
                                  estimator_name='nombre clasificador')
display.plot()

plt.show()
```

Si queremos mostrar varios clasificadores en un mismo gráfico podemos hacerlo de la siguiente forma:

```
#Curva ROC y área AUC
import matplotlib.pyplot as plt
import numpy as np
from sklearn import metrics

#primer clasificador
y = y_test
pred = y_pred_1
fpr, tpr, _ = metrics.roc_curve(y, pred)
roc_auc = round(metrics.auc(fpr, tpr),2)
plt.plot(fpr,tpr,label="SVC(C= 0.1, gamma= 1, kernel= 'linear') ="+str(roc_auc))

#segundo clasificador
y = y_test
```

```
pred = y_pred_2
fpr, tpr, _ = metrics.roc_curve(y, pred)
roc_auc = round(metrics.auc(fpr, tpr),2)
plt.plot(fpr,tpr,label="SVC(kernel='rbf', gamma=0.5, C=0.1) ="+str(roc_auc))

#tercer clasificador
y = y_test
pred = y_pred_3
fpr, tpr, _ = metrics.roc_curve(y, pred)
roc_auc = round(metrics.auc(fpr, tpr),2)
plt.plot(fpr,tpr,label="LGBMClassifier() ="+str(roc_auc))

plt.legend()
```

Informe global con Sklearn

Mediante la métrica `classification_report` de sklearn existe la posibilidad de elaborar un informe con información relevante, sin la necesidad de cálculo de métricas independientes. Con este informe podríamos obtener una primera idea general de la efectividad de nuestro modelo.

```
from sklearn.metrics import classification_report
y_pred=['A','A','A','A','A','A','A','A','B','A','A','B','B','B','B','B','B'
,'B','B','B']
y_true=['A','A','A','A','A','A','A','A','A','B','B','B','B','B','B','B','B'
,'B','B','B']
print(classification_report(y_true, y_pred, labels=['A', 'B']))
```

La salida obtenida será:

	precision	recall	f1-score	support
A	0.80	0.89	0.84	9
B	0.90	0.82	0.86	11
accuracy			0.85	20
macro avg	0.85	0.85	0.85	20
weighted avg	0.86	0.85	0.85	20

Los promedios indicados incluyen la **macro media** (que promedia la media no ponderada por etiqueta), la media ponderada (que promedia la media ponderada por soporte por etiqueta) y **la media de la muestra** (sólo para la clasificación multietiqueta). El **micromedio** (que promedia el total de verdaderos positivos, falsos negativos y falsos positivos) sólo se muestra para la clasificación multietiqueta o multiclase con un subconjunto de clases, porque de lo contrario corresponde a la exactitud y sería la misma para todas las métricas. Por último, **support** representa el número de ocurrencias de cada clase en y_true.

Una función que podemos definir para mostrar los diferentes valores obtenidos para la clasificación utilizando un clasificador concreto puede ser:

```
#También podemos utilizar una función como la siguiente
def print_score(clf, X_train, y_train, X_test, y_test):
  pred = clf.predict(X_test)
```

```
    clf_report = pd.DataFrame(classification_report(y_test,
                                                    pred,
                                                    output_dict=True))
    print("Test Result:\n================================================")
    print(f"Accuracy Score: {accuracy_score(y_test, pred) * 100:.2f}%")
    print("_______________________________________________")
    print(f"CLASSIFICATION REPORT:\n{clf_report}")
    print("_______________________________________________")
    print(f"Confusion Matrix: \n {confusion_matrix(y_test, pred)}\n")
    print('\n')
```

Podemos utilizar esta función por ejemplo si queremos obtener las métricas de un determinado modelo. En el siguiente código se muestra cómo se llamaría para evaluar la clasificación de flores del dataset Iris.

```
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

#cargo dataset iris
iris = datasets.load_iris()
X = iris.data
y = iris.target

#creo conjunto de entrenamiento y test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ra
ndom_state=1, stratify = y)

#normalizo
sc = StandardScaler()
sc.fit(X_train)
X_train = sc.transform(X_train)
X_test = sc.transform(X_test)

# Training a SVM classifier using SVC class
svm = SVC(kernel= 'linear', random_state=1, C=0.1)
svm.fit(X_train, y_train)

print_score(svm, X_train, y_train, X_test, y_test, iris.target_names)
```

Un resultado posible es el siguiente:

```
Test Result:
================================================
Accuracy Score: 97.78%

_______________________________________________
CLASSIFICATION REPORT:
                0          1           2  accuracy  macro avg  weighted avg
precision     1.0   0.937500    1.000000  0.977778   0.979167      0.979167
recall        1.0   1.000000    0.933333  0.977778   0.977778      0.977778
f1-score      1.0   0.967742    0.965517  0.977778   0.977753      0.977753
support      15.0  15.000000   15.000000  0.977778  45.000000     45.000000

_______________________________________________
Confusion Matrix:
 [[15  0  0]
 [ 0 15  0]
 [ 0  1 14]]
```

Ejemplo completo de análisis

En el siguiente enlace (https://colab.research.google.com/drive/1xTxmiZVVLdGn1Zobka-OaD4Zcsts1OxK) existe un documento de Google Colab donde se muestra un ejemplo de análisis completo a partir de un conjunto de datos con el que identificaremos si un caso es considerado un cáncer maligno o no. Puedes encontrar más información del dataset en el siguiente enlace: https://scikit-learn.org/stable/datasets/toy_dataset.html#breast-cancer-dataset.

¡OJO!

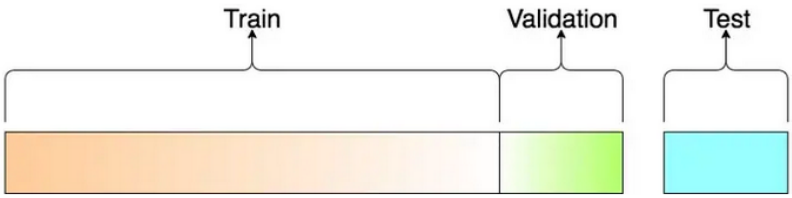
En este ejemplo, además de explicar los nuevos conceptos sobre evaluación, también se explican nuevas técnicas de preprocesamiento, como por ejemplo el generar una **matriz de correlación** entre las variables.

También se hace un repaso de anteriores técnicas del preprocesamiento de datos vistas en la anterior UD. Se pide por favor que se preste atención a estas técnicas ya que son importantes.

1.2.3. Overfitting y underfitting

El área bajo la curva ROC toma un valor entre 0 y 1, en donde 1 representa una clasificación perfecta por parte del modelo. En tal caso, hay que prestar especial atención a que dicho resultado no sea consecuencia de un sobreajuste del modelo (**overfitting**). El sobreajuste de un modelo se produce cuando éste se ajusta casi a la perfección a los ejemplos del conjunto de entrenamiento (tiene capacidad para clasificarlos correctamente), pero no tiene capacidad para clasificar la mayoría de nuevos ejemplos (de los que no ha aprendido). En otras palabras, la capacidad de generalización del modelo es baja. Por el contrario, cuando el modelo entrenado no se ajusta correctamente y no será capaz de identificar patrones, dando unos pésimos resultados.

Una forma de evitar el overfiting es crear un subconjunto de datos de validación; a demás del de entrenamiento y test.



Para ello podemos utilizar la función `train_test_split` que ya hemos utilizado previamente. Un ejemplo de como obtener este subconjunto es el siguiente:

```
# para separar los datos

from sklearn.model_selection import train_test_split

# Separamos conjunto de entrenamiento (90%) y de test (10%)

X_train, X_test, y_train, y_test = train_test_split(Xdf, ydf, test_size=0.1
0, random_state=1)

# Del conjunto de entrenamiento inicial, volvemos a separar en el conjunto
de entrenamiento definitivo (80%) y de validación (20%)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_si
ze=0.20, random_state=1)

# De esta manera se obtiene el 72% de los datos para entrenar, el 18% de da
tos para validar, y el 10% de los datos para testear.
```

Más adelante veremos cómo podemos utilizar este nuevo subconjunto para la creación de un tipo concreto de modelos.

2. *Modelado de redes neuronales*

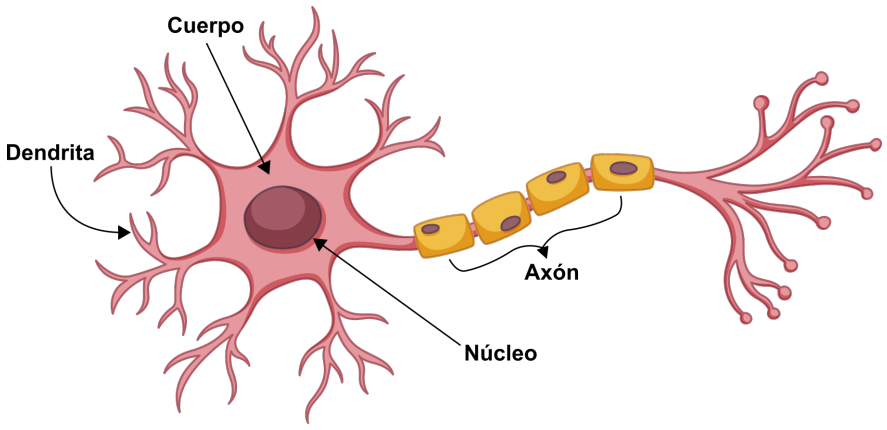
En esta sección veremos estos tres conceptos clave respecto a la programación de redes neuronales:

- Modelos de tipo Sequential, sus tipos de capas y los parámetros de éstas.
- Principales protagonistas del entrenamiento del modelo: función de coste y optimizadores.
- Otros parámetros de gobierno del entrenamiento como las iteraciones epochs o el número de muestras de entrenamiento (batch_size).

2.1. Fundamentos básicos

Las redes neuronales, independientemente de que su origen sea biológico o artificial, se constituyen como un sistema compuesto de neuronas conectadas entre sí que reciben señales y las transmiten entre ellas. Las neuronas son la unidad básica de este esquema, constituidas por un cuerpo celular y ramificaciones. Solo como curiosidad, esta unidad simple de procesamiento se encuentra la mayor parte del tiempo a la espera de señales procedentes de las ramificaciones que conectan a una con otra neurona.

En esencia, la neurona está formada por un cuerpo o soma donde se encuentra el núcleo. Las ramificaciones que sirven de entrada de información a las neuronas se denomina dendritas. Por el contrario, la señal saliente de la red neuronal proviene del axón, una ramificación más alargada que la dendrita. Las conexiones entre neuronas es un mecanismo que se denomina sinapsis y se realiza uniendo el axón a una o varias dendritas.



Las neuronas emiten señales que se propagan de una a otra mediante una compleja reacción electroquímica, las cuales controlan la actividad del cerebro a corto plazo. Por el contrario, estas señales también habilitan la conectividad de las neuronas, a largo plazo. Estos mecanismos muy elaborados son la base del aprendizaje. La mayor parte del procesamiento de la información tiene lugar en la corteza cerebral, la capa externa del cerebro.

Una red neuronal artificial no busca imitar el comportamiento de su equivalente biológica. Al contrario, se ignoran la mayoría de los mecanismos internos de las neuronas porque replicar su comportamiento no es en absoluto necesario para satisfacer el objetivo deseado. A modo de ejemplo, cualquier artefacto capaz de navegar, ya sea bajo o sobre el agua, no está construido siguiendo fielmente la estructura de

un pez, es decir, aletas y branquias. Algo similar ocurre con las redes neuronales artificiales, dado que el objetivo no es simular el sistema biológico, sino imitar, de algún modo, el procesamiento del cerebro humano para añadir comportamiento inteligente a sistemas con cerebros artificiales.

Obviamente, los cerebros biológicos y los ordenadores presentan propiedades totalmente diferentes. Por un lado, una red neuronal, cuyo origen no sea artificial, es capaz de procesar, y simultáneamente, información en cada una de las unidades básicas de su estructura, es decir, las neuronas. Por el contrario, los computado- res disponen de dos componentes para replicar este comportamiento, haciendo uso de un cerebro artificial, denominado CPU que procesa información, y de una memoria que guarda dicho resultado. Por lo tanto, la gran diferencia existente es que las neuronas biológicas son capaces de procesar información por su cuenta. En otras palabras, las neuronas tratan grandes cantidades de información paralelamente, mientras que una CPU realiza este cometido de uno en uno.

No obstante, inspirado en el procesamiento paralelo, las siguientes secciones presentan la arquitectura de una red neuronal artificial capaz de procesar simultáneamente numerosos datos, replicando a alto nivel el comportamiento de una red neuronal biológica.

2.2. Elementos básicos de una Red Neuronal Artificial

En el siguiente vídeo, veremos una introducción a lo que son las redes neuronales donde se comienzan explicando las redes neuronales naturales y su analogía con las redes neuronales artificiales.

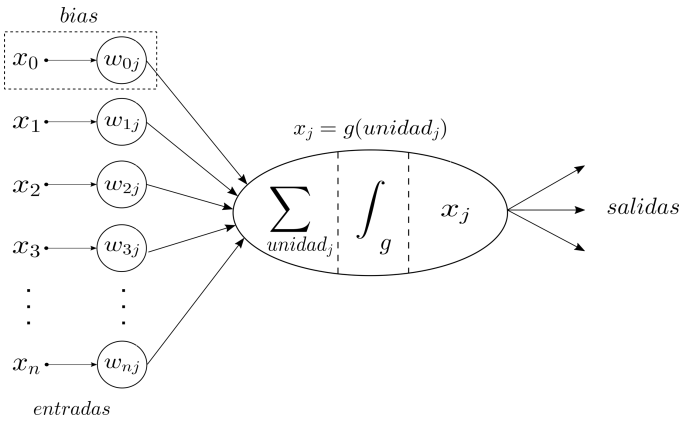
https://www.youtube.com/watch?v=CU24iC3grq8

2.2.1. El perceptrón

Una red neuronal artificial, al igual que una red biológica real, está compuesta de nodos o unidades denominadas neuronas artificiales o **perceptrones**. Cada una está conectada por un enlace directo al cual se le asigna un peso numérico que determina la influencia de la conexión sináptica entre las neuronas. Estos pesos se utilizan como multiplicadores de las entradas de la red neuronal, resultando en una suma pondera- da del producto de los pesos y entradas.

Por ejemplo, si contamos con una red neuronal compuesta de 5 entradas x1, x2, x3, x4, x5, también disponemos de 5 pesos asociados a cada entrada w1, w2, w3, w4, w5. Además, se suele añadir un término extra denominado sesgo o «bias», una medida que indica lo fácil que es disparar la salida de la neurona. Como resultado, la salida en este paso consiste realizando la siguiente suma ponderada: $x0 \times w0 + x1 \times w1 + x2 \times w2 \times \cdots \times x5 + w5$. Una vez computado el resultado, la neurona efectúa una operación más, aplicando una función de activación cuyo término es prestado de la neurociencia, la cual determine la salida de la neurona. En otras palabras, una predicción o decisión a partir de las entradas proporcionadas a la red.

Para dotar de inteligencia a esta estructura, utilizando el símil de la red neuronal biológica, el aprendizaje resulta de la modificación de la influencia de las conexiones sinápticas entre neuronas. Esto se consigue visualizando la red neuronal artificial como un sistema dinámico que modifica los valores de sus pesos. De un modo simplificado, este es el funcionamiento de una red neuronal artificial. Veamos a continuación los detalles técnicos de su estructura.



La figura muestra el esquema general de un elemento mínimo de procesamiento en una red neuronal (el perceptrón). En esencia, una red cuenta con numerosas entradas (neuronas) y, en base a su simplicidad o complejidad, una o varias salidas. La salida x_j representa la activación de la entrada x_i y el peso w_{ij} en el enlace desde la unidad i hasta la unidad j. El valor de la red a la unidad j se puede definir como:

$$unidad_j = \sum_{i=0}^{n} x_i w_{ij}$$

Nótese que el número de interconexiones en una red determina indudablemente la velocidad en la cual se computa esta operación.

Una vez resuelto el valor de la combinación entre los valores de la entrada y los pesos, se le aplica una función de activación *g* para resolver la salida, es decir, la predicción o decisión como consecuencia de las entradas introducidas en la red. Esta función se denota como:

$$x_j = g(unidad_j) = g(\sum_{i=0}^{n} x_i w_{ij})$$

La función de activación *g* trata de imitar el mecanismo de los impulsos eléctricos en la comunicación entre dos o más neuronas. Este comportamiento se intenta replicar, de algún modo, con un umbral mediante la **función escalón** (véase *parte a* de la siguiente Figura) o con una versión suavizada usando la **función sigmoide** (véase *parte b* de la siguiente Figura). En ocasiones poco frecuentes se utiliza también la función identidad, cuyo objetivo es no hacer nada y producir la combinación lineal. Es por ello que rara vez se utiliza esta función en redes neuronales, dado que su comportamiento no conduce a nada nuevo.

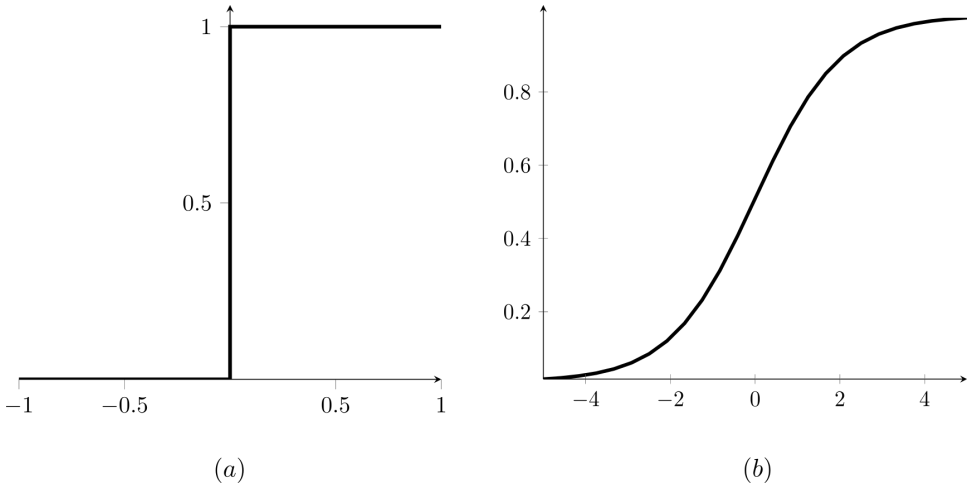


Figura 3. Funcion escalón y sigmoide

La función **escalón** (Parte a de Figura 3) produce una salida de 1 cuando el resultado de la combinación de entradas y pesos es mayor de 0. De lo contrario, su salida será 0. Como es de esperar, una salida con valor 1 significa que la unidad de salida se activa.

$$\theta(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

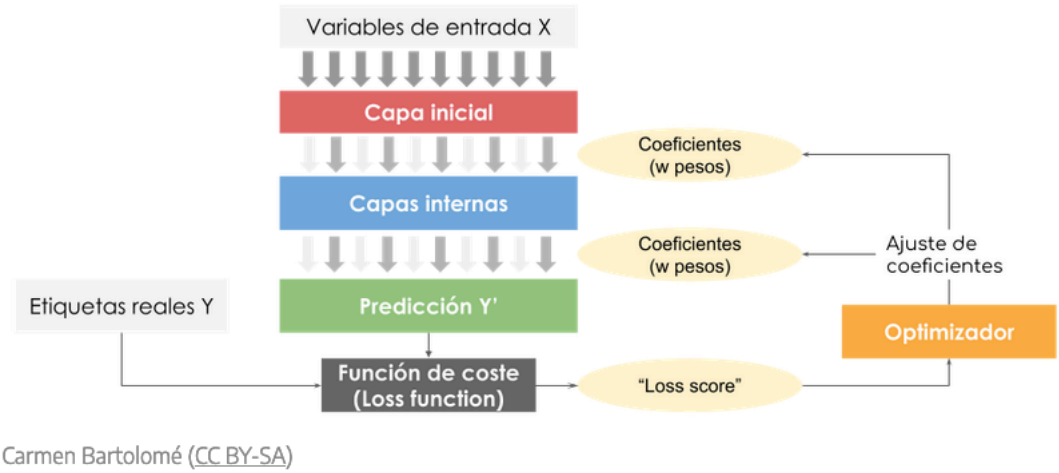
Por el contrario, la función **sigmoide** (Parte b de Figura 3) es una modificación suavizada de la función escalón que produce salidas entre 0 y 1. El resultado de las salidas se puede interpretar como probabilidades, lo cual es útil para predecir cómo de probable es que ocurra algo, en lugar de utilizar valores numéricos enteros. El uso de esta función no lineal se emplea para mejorar el aprendizaje de los pesos y el sesgo de la red. Al realizar pequeñas modificaciones en ambos elementos se produce un ligero cambio en la salida, resolviendo los valores adecuados para una tarea concreta, a diferencia de valores enteros que pueden cambiar el comportamiento del resto de la red.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

No obstante, es imprescindible destacar que la convención a usar (salidas con números enteros o reales) viene determinada por el tipo de problema a resolver. En otras palabras, no existe una función que sea la «panacea» ante la resolución de un problema cualquiera.

2.2.2. Las capas

Una red neuronal va a estar compuesta de varios perceptrones conectados entre sí, apilados en capas. En Keras, la clase a la que tenemos que llamar para crear un modelo con varias capas será "**Sequential**". A lo largo de esta sección, siempre empezaremos así la construcción del modelo.



Carmen Bartolomé (CC BY-SA)

La capa o "layer" de tipo Dense, es la que representa verdaderamente una red neuronal con todos los nodos conectados con las diferentes variables de entrada, en mayor o menor medida según el valor del coeficiente de cada conexión. En la imagen superior, hay dos capas (con sus correspondientes perceptrones) que operan con combinaciones de las variables de entrada. A esto, se le llama red "densamente" conectada.

La clase Dense, crea un objeto que implementa la operación

output = activation(dot(input, kernel) + bias)

donde:

- *activation* es la función de activación que actúa sobre el resultado que se da en cada neurona.
- *kernel* es la matriz de coeficientes o pesos "w" que se generan de forma aleatoria
- *bias* es el vector del sesgo que solo es distinto de cero si se pasa como argumento True.

2.3. Definición de redes neuronales utilizando Keras y TensorFlow

Toda la información que aparece en esta sección se ilustra con la implementación en Python utilizando la librería Keras de Tensorflow. Todo el código lo puedes encontrar en el documento compartido del siguiente enlace: https://colab.research.google.com/drive/1xF4Xcc7XKGfYNGTwLPbxrHXWk4T6yx-c

La clase Dense implementa la operación `output = activation(dot(input, kernel) + bias)` que hemos visto anteriormente. Pero además presenta números parámetros, de los que se explicarán los que se consideran más importantes en este momento del aprendizaje. Estos parámetros, entre otros, se pueden encontrar en el esquema:

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
```

```
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Units

El parámetro **units** se refiere al número de neuronas que debe tener la capa. Debe ser un número entero y positivo.

Activation

La función de **activación**, en realidad, es opcional. Si no se indica una, no se aplicará activación y el resultado de la neurona será directamente el cálculo lineal. Para problemas lineales, no hay inconveniente, pero si necesitamos que el modelo genere una solución de tipo no lineal (curvas o superficies curvas), es imprescindible aplicar funciones de activación, que irán "dibujando" esa superficie a costa de encender y apagar los nodos de cálculo que son las neuronas según vayan dando valores que ayuden a tener el resultado final correcto.

Por ejemplo, para definir un modelo de tipo DNN (Deep Neural Network) o red neuronal profunda, con dos capas de red neuronal, el código podría ser:

```
model= keras.models.Sequential()

model.add(keras.layers.Dense(32, activation='relu', input_shape=(12,)))
model.add(keras.layers.Dense(32))
```

En este ejemplo, hay 12 variables de entrada. Hay dos capas tipo red neuronal, ambas con 32 neuronas. La primera, tiene como función de activación, la ReLu, mientras que la segunda, no tiene función de activación.

Input_shape

La capa de entrada debe tener la misma forma que tus datos de entrenamiento. Si tienes 30 imágenes de 50x50 píxeles en RGB (3 canales), la forma de los datos de entrada es (30,50,50,3). Por lo tanto, la capa de entrada debe tener esta forma. En el caso de capas Dense requiere una entrada de la forma (batch_size, input_size).

En algunas ocasiones, se recurre a **la capa de tipo** Flatten. Es una capa que "aplana" una estructura de datos de entrada de más de una dimensión, para que tengamos un vector, o array de una dimensión, que es lo que aceptan las capas de redes neuronales. En el caso de trabajar con imágenes, lo normal es tener, como datos de entrada, una serie de matrices o arrays de N x N pixels. Por ejemplo, si tenemos un dataset con 1000 imagenes de 12 x 12 pixels, la estructura de datos de entrada o dataset.shape sería: (1000, 12, 12). Al aplicar la capa Flatten, obtenemos una estructura de salida (1000, 144). Para este ejemplo, el código sería:

```
import keras

model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape = (12,12)))
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(keras.layers.Dense(1,activation = 'sigmoid'))
```

Inicialización del kernel

El algoritmo de retropropagación requiere un punto de partida en el espacio de posibles valores de los pesos. La idea de utilizar un enfoque no determinista frente a uno determinista se resume en el rendimiento del algoritmo, fundamentalmente porque la segunda opción resulta ineficiente para problemas complejos.

Por normal general, los pesos no deben inicializarse al valor 0, ya que esto imposibilita que el algoritmo de ajuste busque de forma efectiva los valores correctos para realizar adecuadas predicciones. Históricamente, la inicialización de pesos se ha venido definiendo mediante heurísticas simples, como por ejemplo:

- Pequeños valores aleatorios entre [0, 1]
- Pequeños valores aleatorios entre [-1, 1]
- Pequeños valores aleatorios entre [-0.1, 0.1]
- Otras similares.

En general, la inicialización de los pesos se realiza siguiendo una distribución gaussiana o uniforme. Sin embargo, la escala de la distribución inicial tiene un gran efecto tanto en el resultado del procedimiento de optimización como en la capacidad de generalización de la red. En definitiva, la inicialización de los pesos afecta considerablemente en el proceso de aprendizaje de la red. Estas técnicas continúan funcionando en general. Sin embargo, se han desarrollado nuevos enfoques que se han convertido en la norma de facto, dado que pueden dar lugar a un entrenamiento del modelo muchos más optimizado.

En esencia, estas nuevas heurísticas se clasifican en función del tipo de función de activación utilizada, principalmente destinadas a la función sigmoide, tangente hiperbólica y ReLU.

Debes saber…

El método utilizado para inicializar los pesos de una red con funciones de activación sigmoide o tangente hiperbólica se denomina **inicialización de Glorot** o **inicialización de Xavier.**

Por otro lado, la estrategia de inicialización para la función de activación ReLU (y sus variantes, incluida la activación ELU) se denomina **inicialización He**.

Estos inicializadores los puedes encontrar dentro del modulo tf.keras.initializers. También puedes acceder a este enlace para ver más información sobre los inicializadores.

https://faroit.com/keras-docs/2.0.6/initializers/

Se puede definir de dos formas, (1) a través de su nombre y (2) mediante un objeto de esa clase

```
(1)   Dense(64, kernel_initializer='he_uniform')
(2)   Dense(64, kernel_initializer=tf.keras.initializers.he_uniform)
(3)   Dense(64, kernel_initializer= tf.keras.initializers.HeUniform())
```

2.3.1. Número de neuronas

Una de las dudas más corrientes es sobre el número de neuronas a definir en cada capa. En los problemas de clasificación, hay un criterio muy claro:

- Si es clasificación binaria, la capa de salida tendrá una única neurona
- Si es clasificación múltiple, la capa de salida debe tener tantas neuronas como clases o categorías de clasificación tenga el problema.

Pero no hay un criterio claro para las capas internas. Hay algunos planteamientos matemáticos que pueden ayudar, pero la tendencia es, precisamente, probar mucho e ir adquiriendo experiencia que aporte la intuición necesaria para hacer una primera estimación que nos aporte un buen modelo lo antes posible.

Te recomendamos que empieces por configuraciones muy sencillas, con un número de neuronas dentro del orden de magnitud de las variables de entrada. Después, podrás ir probando a aumentar capas y neuronas en diferentes configuraciones.

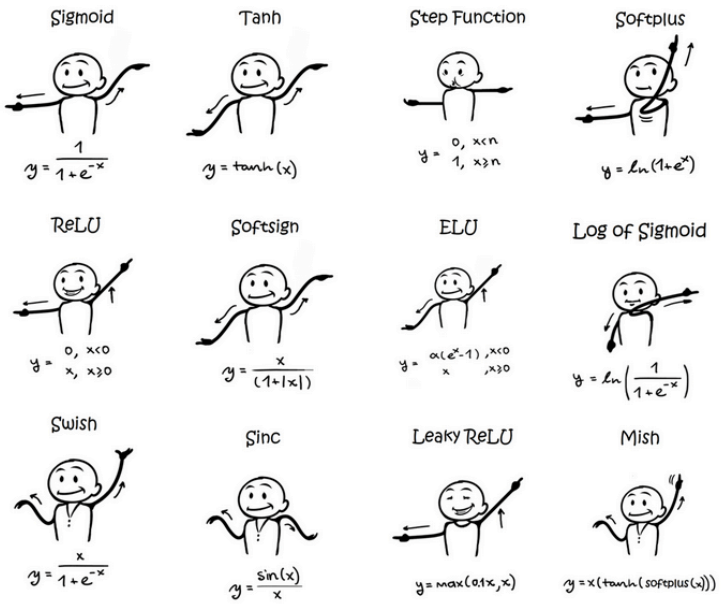
En todo caso, lo más útil es aprender de los modelos que otros ya han creado y probado.

2.3.2. Funciones de activación

Como introducción a esta sección, puedes visualizar el siguiente vídeo sobre las funciones de activación: https://www.youtube.com/watch?v=_0wdproot34 En este video puedes ver las características principales de estas funciones y sus ventajas e inconvenientes.

Hay muchas posibles funciones de activación, pero se suele utilizar siempre una de estas:

- ReLu
- Sigmoid
- Softmax
- Tanh



El criterio para aplicar una u otra, reside en su comportamiento matemático, pero no hay un criterio absoluto. En general, lo recomendable es partir de una configuración básica e ir probando con cambios controlados. Algunos consejos sobre dicha configuración básica serían:

- Utiliza ReLu para capas internas u ocultas
- En la capa de salida, si estás en un problema de clasificación binaria, utiliza Sigmoid
- En la capa de salida de un problema de clasificación múltiple, utiliza Softmax

Para saber más…

Sobre las funciones de activación, puedes leer el análisis que hace B. Chen en este artículo. Y sobre la función Softmax, también tienes un artículo muy descriptivo y completo que te puede orientar mejor.

2.3.3. Recomendaciones para la definición de la arquitectura

La definición de la arquitectura de una red neuronal es una tarea que requiere mucha experiencia, pero se pueden dar ciertas recomendaciones como la que aparecen en XXXX y que se muestran en la siguiente tabla:

Hiperparámetro	Clasificación binaria	Clasificación binaria multietiqueta	Clasificación multiclase
N.º de neuronas de entrada	Una por característica de entrada	Una por característica de entrada	Una por característica de entrada
N.º de capas ocultas	Normalmente entre 1 y 5	Normalmente entre 1 y 5	Normalmente entre 1 y 5
N.º de neuronas por capa oculta	Normalmente entre 10 y 100	Normalmente entre 10 y 100	Normalmente entre 10 y 100
Activación capa interna u oculta	ReLu	ReLu	ReLu
N.º de neuronas de salida	1	1 por etiqueta	1 por clase
Activación capa de salida	Sigmoid	Sigmoid	Sofmax
Función de pérdida	Entropía cruzada	Entropía cruzada	Entropía cruzada

2.4. Compilar Red Neuronal Artificial

La parte de nuestro algoritmo que se encarga de configurar cómo va a ser el entrenamiento, viene controlada por la función "compile". Echemos un vistazo a la línea de código de un posible algoritmo para clasificación:

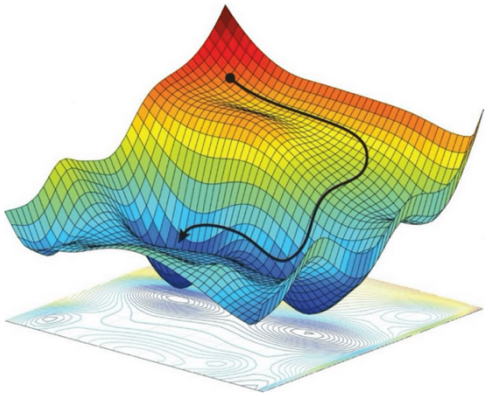
```
model.compile(optimizer= 'Adam',
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

A continuación veremos los diferentes parámetros que existen a la hora de configurar la compilación de una red neuronal, y que fijarán la estrategia a seguir durante su entrenamiento.

2.4.1. Tipos de función de coste (Loss)

El parámetro loss, representa lo que llamamos "función de coste" o "función de pérdida" y es una métrica necesaria para que el proceso de ajuste de los coeficientes o pesos de las redes neuronales en las capas de nuestro modelo, se vayan actualizando adecuadamente hacia el modelo definitivo ya entrenado.

Esta figura, representa lo que podría ser la superficie generada por la función Loss en un problema de dos variables. Los distintos puntos de esta superficie dependen del estado del modelo, en función de los valores de los pesos o coeficientes que vamos recorriendo en los ejes. El entrenamiento del modelo, básicamente, consiste en buscar los valores de los pesos que corresponden al valor mínimo de esta superficie. Con cada iteración, nos vamos moviendo por ella, calculando los distintos valores del error hasta encontrar el menor de ellos.



Arizan & Assibi (CC BY-SA)

En keras, tenemos disponibles varias funciones de coste o "loss functions" adecuadas para nuestros modelos. Vamos a repasar las más utilizadas en deep learning, identificando los casos en los que utilizar cada una de ellas. En general, para los modelos basados en neuronas, es necesario utilizar lo que se conoce como cálculo de la "entropía cruzada", en contraposición al error cuadrático medio o MSE que se venía utilizando en otros modelos.

Las siguientes tres funciones hacen un cálculo del índice de error entre las clases reales dadas por las etiquetas y las predicciones que va dando el modelo. En realidad, se proporcionará una media de la pérdida o error de todas las instancias de la muestra en cada iteración (epoch) del proceso de entrenamiento. Pero cada una de ellas está programada para un tipo de problema de clasificación:

- **Binary Crossentropy**: es la función de coste indicada para trabajar con problemas de clasificación binaria, junto a la función de activación sigmoide.
- **Categorical Crossentropy**: adecuada para problemas de clasificación múltiple, pero con etiquetas o variables de salida de tipo categórico en formato one-hot encoding.
- **Sparse Categorical Crossentropy**: es la función de coste para problemas tanto binarios como múltiples, pero con las etiquetas o clases de salida dadas como números enteros.

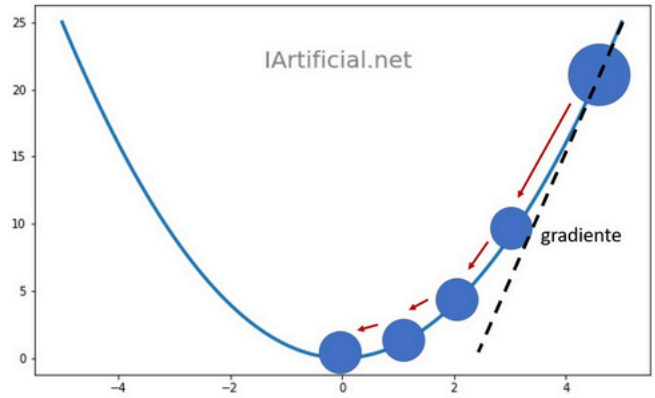
En Keras puedes encontrar estas funciones en el módulo `tf.keras.losses`

2.4.2. Optimizadores

Para entrenar el modelo, necesitamos alcanzar la configuración de los pesos w en toda la red que hace que el error sea mínimo. El método matemático que se utiliza para buscar el valor mínimo de una función es la derivada, y, en un caso multidimensional, el gradiente, que se compone de las derivadas parciales de la función respecto a cada una de las variables. Cuando nos acercamos a un mínimo, la derivada, que nos da el valor de la pendiente de la tangente a la curva, se va haciendo más negativa.

Esta técnica se denomina "descenso del gradiente", y los métodos de optimización se basan en favorecer esa evolución hacia el mínimo.

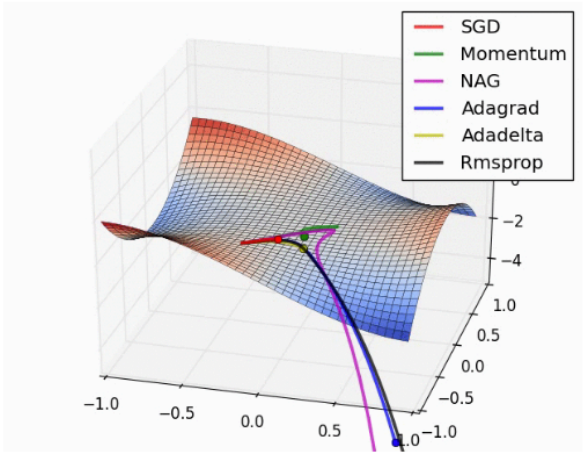
$$w_{k+1} = w_k - \eta \frac{1}{n} \sum_{i=t}^{t+n-1} \nabla f_{w_k}(x^i)$$



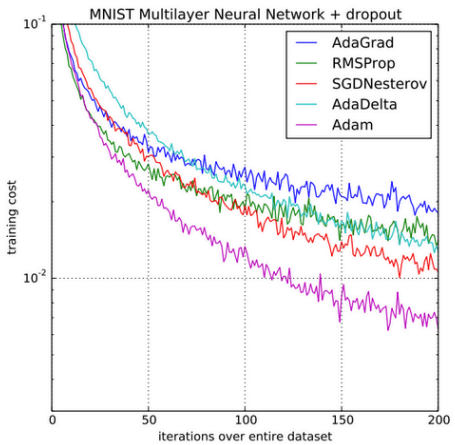
IArtificial.net (CC BY-SA)

Como aproximación inicial al mundo de los optimizadores, te recomendamos que utilices, de momento, uno de estos dos: **RMSprop** o **Adam**. A medida que vayas aprendiendo más y adquiriendo experiencia, podrás ir explorando las posibilidades particulares que te ofrecen los demás. En estos gráficos, puedes comprobar el buen desempeño, en general, que tiene cada uno, pero si tenemos que caracterizar de alguna manera sus ventajas, puedes considerar que:

- **RMSprop** presenta una convergencia hacia el mínimo más rápida
- **Adam** presenta un mejor comportamiento general. Es una buena opción en tus primeros entrenamientos.



Optimization Open Textbook (CC0)

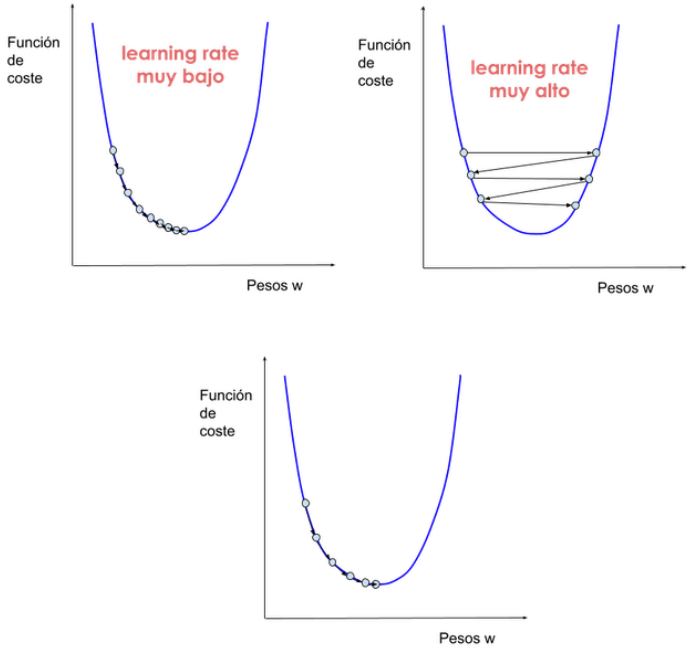


Optimization Open Textbook (CC0)

En Keras puedes encontrar estas funciones en el módulo `optimizers`

Ratio de aprendizaje (Learning rate)

Es el parámetro que controla la magnitud con la que modificamos los pesos en función de la pendiente de la función de coste. Podríamos decir que es un factor que amplifica el efecto de la pendiente de la función de coste. Si el ratio de aprendizaje es pequeño, aunque estemos en una zona de fuerte pendiente, se avanzará a pasos pequeños, y en la zona de baja pendiente, según vamos alcanzando el mínimo, cada vez avanzará mucho más despacio, lo que hará el entrenamiento muy lento. Si el learning rate, por el contrario, es demasiado alto, en la zona de pendiente muy elevada, hará pegar un salto al factor de corrección de los pesos, que nos iremos a la otra cara de la función de coste. El proceso de entrenamiento será muy inestable e incluso podría no converger. El rango de un learning rate de partida **está entre 0,0001 y 0,001**. A veces hay que tomar uno más alto o más bajo de los valores de ese rango, pero puedes empezar por ahí e ir probando.



Carmen Bartolomé (CC BY-SA)

En keras, al llamar a la función de optimización, podemos pasarle un valor de learning rate como argumento por clave:

```
opt = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

Para saber más…

Para conocer mejor el uso de los optimizadores con keras, puedes consultar la documentación que hay en su API reference. También puedes leer este artículo en el que el autor hace un experimento probando diferentes optimizadores para varios casos.

2.5. Entrenamiento de una red neuronal

Tras toda la configuración del modelo y de los parámetros que permitirán el entrenamiento del mismo, ya solo queda proceder a éste. Keras proporciona el método fit para el entrenamiento de un modelo basado en redes neuronales profundas. Un ejemplo de la aplicación de esta función sería:

```
model.fit(X_train, y_train, epochs = 20, batch_size = 40)
```

El método fit necesita tres parámetros ineludibles: los datos de entrada X, las etiquetas o datos de salida y finalmente, el número de epochs.

El parámetro **epochs** representa el número de iteraciones del entrenamiento. En cada iteración, tiene lugar el proceso por el cual el optimizador busca un mínimo de la función de coste. A base de hacer varias iteraciones con distintas muestras de datos, se va buscando un mínimo cada vez más "mínimo". ¿Recuerdas que queremos quedarnos con el punto de la función de coste que constituye el mínimo global?

Las muestras que se utilizan en cada iteración o epoch, se gestionan con el parámetro **batch_size**. Si no se indica nada, este parámetro toma el valor, por defecto, de 32 muestras. En el ejemplo, estamos indicando que queremos 40 muestras o lotes, lo cual implica que el conjunto total de datos se dividirá entre ese número de lotes. Por ejemplo, si tenemos 60.000 registros, entre 40 lotes, nos da 1500 registros por iteración. Esto quiere decir que, en cada epoch, la función de optimización irá fijando el valor de los pesos para minimizar el valor de la función de coste según el comportamiento de la red con esos 1500 casos de X y si se aciertan o no sus correspondientes etiquetas y.

A la hora de entrenar un modelo de red neuronal podemos indicar un conjunto de validación con el objetivo de reducir el sobreajuste. Para ello podemos utilizar el parámetro **validation_data** que es una tupla donde el primer elemento es el conjunto de características para realizar la predicción y el segundo elemento es la predicción que se debería hacer para el anterior conjunto.

```
model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val,y_val))
```

Aquí hay otra opción: el argumento **validation_split** le permite automáticamente parte de reserva de sus datos de entrenamiento para su validación. El valor del argumento representa la fracción de los datos a ser reservados para la validación, por lo que se debe establecer en un número mayor que 0 y menor que 1. Por ejemplo, **validation_split=0.2** medios de "uso 20% de los datos para la validación".

```
model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_split=0.2)
```

2.6. Proyectos basados en redes neuronales

En esta sección veremos 4 ejemplos de proyectos donde las redes neuronales son el clasificador utilizado. Es importante que veas las diferentes formas de definir la arquitectura, compilación y entrenamiento de las redes.

2.6.1. Proyecto 1. Conversión Celsius a Fahrenheit

En el siguiente enlace se muestra cómo desarrollar una primera red neuronal para la conversión de grados Celsius a Fahrenheit https://www.youtube.com/watch?v=iX_on3VxZzk. El código fuente lo puedes encontrar en https://colab.research.google.com/drive/1JVGNOFkONX_I6qWWvCkSWcGoJX-GVmaw

2.6.2. Proyecto 2. Clasificador de imágenes

A continuación, tienes un ejemplo de un modelo para clasificación de imágenes, en el que puedes ver todos los elementos e hiperparámetros que hemos visto. Fíjate bien en qué función de coste se ha utilizado, y qué optimizador.

https://colab.research.google.com/drive/1uQWKXU4SUJXX6_DJQFWpG5gGYwI1gy_z

2.6.3. Proyecto 3. Clasificador comentarios películas

También te presento un ejemplo donde se utiliza una red neuronal con el objetivo de clasificar comentarios sobre películas.

https://colab.research.google.com/drive/1LLAJmEszLWB_ui6XLcNqb33WyCGIxjDs

2.6.4. Proyecto 4. Reconocimiento de dígitos

Otro ejemplo que presento es la identificación de dígitos utilizando redes neuronales. Puedes ver el ejemplo en el siguiente enlace:

https://drive.google.com/file/d/1ZGgXMrSe7egUbkVkgxmHP_Fd64PNpy7G/view

3. *Bibliografía*

- Aurélien Géron. Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow. O'Reilly.
- Aurélien Géron. Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow. O'Reilly.
- Peter Bruce, Andrew Bruce y Peter Gedeck. Estadística práctica para ciencia de datos con R y Python. Marcombo.
- Materiales formativos FP Online del Ministerio de Educación y Formación Profesional. Módulo de Programación de Inteligencia Artificial.
- https://keras.io
- https://www.tensorflow.org