Práctica 2: Visión artificial y aprendizaje

Objetivos

- Comprender el funcionamiento general de las técnicas de aprendizaje supervisado y en concreto los métodos de *boosting* y las redes neuronales artificiales.
- Entender el concepto de imagen y píxel, y cómo podemos utilizar las técnicas de aprendizaje supervisado para distinguir lo que representa una imagen.
- Conocer en detalle el algoritmo adaboost basado en clasificadores débiles de umbral.
- Entender el papel de un clasificador débil en métodos de *boosting*.
- Conocer otros tipos de clasificadores débiles.
- Conocer las librerías Keras y scikit-learn para implementar clasificadores de forma rápida y efectiva utilizando diversidad de técnicas de aprendizaje automático.
- Implementar clasificadores para dígitos manuscritos con buenas tasas de acierto mediante técnicas de adaboost y redes neuronales.
- Utilizar gráficas para realizar configurar correctamente los distintos clasificadores y comparativas de rendimiento entre diferentes métodos.

Sesión 1: Introducción al problema a resolver y el entorno de trabajo. Normas de entrega y evaluación.

Introducción

En esta práctica trabajaremos con técnicas de aprendizaje supervisado aplicadas al reconocimiento automático de números manuscritos, en nuestro caso las cifras del 0 al 9 de la base de datos MNIST (http://yann.lecun.com/exdb/mnist/). Esto nos permitirá también trabajar con imágenes y píxeles, en problemas reales de dificultad moderada. Utilizaremos también la librería de *machine learning* scikit-learn, que nos permite configurar modelos de clasificación y regresión utilizando una gran variedad de técnicas de aprendizaje automático.

La práctica se divide en dos partes. En la **primera parte** desarrollaremos en Python, programado desde cero, un clasificador multiclase adaboost para reconocer las cifras del 0 al 9. En la **segunda parte** utilizaremos la librería scikit-learn (https://scikit-learn.org/) de Python para implementar el

mismo clasificador multiclase adaboost, y un multi-layer perceptron (MLP) para resolver el mismo problema utilizando la librería Keras (https://keras.io/) de TensorFlow.

Entorno de trabajo

En los laboratorios trabajaremos con Python en Linux. Para quienes no tengan experiencia con Python se recomienda utilizar el IDE Thonny, ya instalado en los laboratorios, puesto que da mejores ayudas a programadores de Python nóveles que otros IDE como Visual Studio. Dado que Python es un lenguaje interpretado multiplataforma muy extendido y se puede usar en Windows y otros sistemas operativos, cada estudiante puede elegir el entorno de trabajo que prefiera (tanto sistema operativo como IDE), pero deberán asegurarse de que su código fuente funciona correctamente en Linux en los PC de los laboratorios, porque ese es el entorno en que se evaluará la entrega. Además de las librerías de scikit-learn y Keras mencionadas antes, en general siempre trabajaremos con Numpy (https://numpy.org/) para la manipulación eficiente de arrays, y alguna de las facilidades que nos provee Pandas (https://pandas.pydata.org/), y utilizaremos Matplotlib (https://matplotlib.org/) para el dibujado de imágenes y gráficas en pantalla. Puedes utilizar cualquier librería de Python que se pueda instalar usando `pip --user install <librería>` en los laboratorios. Como alternativa, tendréis la opción de utilizar Google Colab para realizar la práctica. **Si tu código no puede ejecutarse en los laboratorios o en Google Colab, no será evaluado, salvo acuerdo previo con tu profesor de prácticas.**

Evaluación

Cada una de las dos partes aporta hasta 5 puntos de la nota final (máximo 10). Dentro de cada parte hay apartados obligatorios y optativos. Los obligatorios suman hasta 5 puntos en cada parte, y los optativos permiten mejorar la nota de los obligatorios hasta su nota máxima. La evaluación de las tareas se hace conjuntamente en cada parte, su implementación supone el 40% de la nota y su documentación el 60%.

Esta práctica tiene **dos hitos de entrega, ambos obligatorios**. En el primer hito has de entregar la implementación de la primera parte de la práctica, que será revisada pero no contará para la calificación final: la usaremos para comprobar el desarrollo de la práctica hasta el momento y ayudar a corregir errores y solucionar problemas antes de la entrega final. El segundo hito será la entrega final, en la que se basará la calificación. Sin embargo, **si no entregas el hito 1** o si no alcanzas un mínimo de implementación de los apartados de la primera parte, podrá suponer una **penalización de hasta un 20% de la nota final**.

Documentación

La documentación es la parte más **importante** de la práctica, cuenta como el **60% de la nota máxima**. Tu documentación incluirá una sección para cada apartado de la práctica, donde pondrás las respuestas a las preguntas que se te plantean y los resultados que se piden en cada apartado,

además de cualquier consideración que consideres relevante sobre el desarrollo de la práctica, pruebas extra que hayas realizado, mejoras que hayas implementado, o conclusiones extraídas de esos experimentos. Es importante que concretes y sintetices (que vayas al grano y que resumas). Si no tienes clara la respuesta a una pregunta o de qué manera debes informar sobre tus experimentos, es mejor que le pidas ayuda al profesor durante las horas de prácticas y no que copies parrafadas de ChatGPT que no terminas de entender o que pongas páginas y páginas de gráficas o tablas repetitivas.

La documentación **debe incluir una sección de bibliografía** con todas las referencias que utilices, o estas referencias deben enlazarse en el texto donde se usen (entre paréntesis o como nota al pie). Las consultas a ChatGPT, Bing y otros chatbots basados en grandes modelos de lenguaje son también parte de la bibliografía. Si copias sus respuestas textualmente, debes marcarlas como una cita, entre comillas, dado que los resultados de un *prompt* no se pueden enlazar. Aunque solo los utilices como fuente de documentación (sin citar textualmente), también debes referenciar el chatbot en la bibliografía, con preferencia por indicar el prompt concreto con el que se le hizo la consulta al chatbot. Cada referencia de la sección de bibliografía debe aparecer citada en el texto, en los apartados en que se ha usado. Una bibliografía mínima puede consistir solo en los apuntes de clase (aunque recomendamos que investigues más por tu cuenta). **La ausencia de bibliografía, o si no se referencia en el texto donde se use, podrá penalizar hasta 1 punto**.

Formato de entrega

Las prácticas son individuales, no se pueden hacer en grupos y no se puede reusar código o texto de otros compañeros. La entrega se realizará **a través de Moodle**, y debe consistir en dos ficheros, **un fichero .pdf con la documentación y otro fichero .py con todo el código Python utilizado por el estudiante**, tanto las implementaciones de los algoritmos requeridos como cualquier código utilizado para producir cualquier resultado que se muestre en la documentación. Ambos archivos se deben llamar con el nombre completo del estudiante, usando '_' como separador entre palabras (p. ej. Juan_Carlos_Sánchez-Arjona_de_los_Rios.py y .pdf).

Para permitir la evaluación de tu entrega, deberás crear un módulo (una función) en tu archivo Python para cada tarea de la práctica, que llamará a todas las funciones implicadas en completar esa tarea. Puede que algunas tareas requieran dos módulos, o que reusen funciones de otras tareas con distintos parámetros. Si los nombres de tus funciones son ambiguos, utiliza comentarios para dejar claro qué hace cada módulo. Estos módulos se podrán llamar individualmente en la última sección de tu archivo Python, utilizando el condicional `__name__ == "__main__"`. La idea es que los profesores podamos ejecutar solo la parte que queramos de tu código, sin tener que esperar a que se entrenen todos los clasificadores de todas las tareas.

```
if __name__ == "__main__":

    ## Las llamadas a funciones auxiliares que sean relevantes para algo
    ## en la evaluación pueden dejarse comentadas en esta sección
    #test_DecisionStump(9, 59, 0.4354, 1)

    rend_1A = tareas_1A_y_1B_adaboost_binario(clase=9, T=10, A=10, verbose=True)

    #tarea_1C_graficas_rendimiento(rend_1A)
    ## Una parte de la tarea 1C es fijar los parámetros más adecuados.
    ## Se puede implementar reusando el código de las tareas 1A y 1B
    tareas_1A_y_1B_adaboost_binario(clase=9, T=incognita, A=incognita)

    rend_1D = tarea_1D_adaboost_multiclase(T=incognita, A=incognita)
    #rend_1E = tarea_1E_adaboost_multiclase_mejorado(T=incognita, A=incognita)
    #tarea_1E_graficas_rendimiento(rend_1D, rend_1E)

    rend_2A = tarea_2A_AdaBoostClassifier_default(n_estimators=incognita)
    #tarea_2B_graficas_rendimiento(rend_1E, rend_2A)
    rend_2C = tarea_2C_AdaBoostClassifier_faster(n_estimators=incognita)
    #tarea_2C_graficas_rendimiento(rend_2A, rend_2C)

    rend_2D = tarea_2D_AdaBoostClassifier_DecisionTree(incognitas)
    rend_2E = tarea_2E_MLP_Keras(n_hid_lyrs=incognita, n_nrns_lyr=incognitas)
    rend_2F = tarea_2F_CNN_Keras(incognitas)
    #tarea_2G_graficas_rendimiento(rend_1F, rend_2C, rend_2D, rend_2E, rend_2F)
```

Fechas de entrega

Hito 1 (primera parte de la práctica): hasta el **3 de diciembre de 2023 a las 23:55h**.

Hito 2 (entrega final): hasta el **24 de diciembre de 2023 a las 23:55h**.

Trabajando con las imágenes de números manuscritos de MNIST

Ahora empezamos con lo que es la sesión de prácticas propiamente dicha. Vamos a cargar la base de datos de MNIST y familiarizarnos con la manipulación de esas imágenes. Hay muchas maneras de cargar las imágenes de MNIST en un programa Pyhthon. Nosotros utilizaremos la función que nos facilita Keras. Para ello primero hay que importar esa librería. Al ejecutar ese import veremos una serie de mensajes de advertencia de TensorFlow, quejándose de que no encuentra soporte para usar CUDA en la tarjeta gráfica. Quienes tengan una tarjeta NVIDIA pueden instalar el soporte para CUDA y conseguir entrenamientos de modelos de aprendizaje mucho más rápidos gracias a la paralelización, pero no es necesario en esta práctica. Si queremos eliminar esos mensajes de error por comodidad, podemos utilizar el siguiente código:

```
import logging, os
logging.disable(logging.WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from tensorflow import keras
```

Ahora podemos cargar las imágenes de MNIST y ver en qué consisten:

```
(X_train, Y_train), (X_test, Y_test) = keras.datasets.mnist.load_data()

print(X_train.shape, X_train.dtype)
print(Y_train.shape, Y_train.dtype)
print(X_test.shape, X_test.dtype)
print(Y_test.shape, Y_test.dtype)
```

Vemos que `X_train` es un array de 60000 elementos (en este caso observaciones, o *samples* en inglés) que a su vez son arrays de 28x28 `uint8` (unsigned int de 8 bits, números de 0 a 255), que son las imágenes de números manuscritos de 28x28 píxeles en escala de 256 grises. `Y_train` es otro array de 60000 elementos también `uint8`. Y los arrays `X_test` e `Y_test` son iguales pero contienen 10000 observaciones cada uno. Podemos mostrar en pantalla las imágenes que hay en `X_train` e `X_test` utilizando Matplotlib:

```
import matplotlib.pyplot as plt

def show_image(imagen, title):
    plt.figure()
    plt.suptitle(title)
    plt.imshow(imagen, cmap = “Greys”)
    plt.show()

for i in range(3):
    title = "Mostrando imagen X_train[" + str(i) + "]"
    title = title + " -- Y_train[" + str(i) + "] = " + str(Y_train[i])
    show_image(X_train[i], title)
```

Gracias a Numpy, podemos acceder a un pixel (fila, columna) de todas las imágenes simultáneamente, contar cuántos valores distintos tiene, cambiar sus valores o pintar una gráfica con esos valores:

```
def plot_X(X, title, fila, columna):
    plt.title(title)
    plt.plot(X)
    plt.xscale(xscale)
    plt.yscale(yscale)
    plt.show()

features_fila_col = X_train[:, fila, columna]
print(len(np.unique(features_fila_col)))

title = "Valores en (" + str(fila) + ", " + str(columna) + ")"
plot_X(features, title, fila, columna)
```

Si sobra tiempo, podéis probar más cosas para adquirir más destreza en Python o sacar conclusiones acerca de los datos sobre los que trabajamos. Por ejemplo, contar la cantidad de seises en el conjunto de entrenamiento, o la cantidad de píxeles que no valen más de un cierto valor Z.

Sesión 2: Adaboost binario con clasificadores de umbral

Tarea 1A (OBLIGATORIA): implementa las clases Adaboost y DecisionStump

Vuestra primera tarea será crear un fichero .py con vuestro nombre y apellidos, que será el que usaréis en la entrega, y copiar el siguiente código Python incompleto con la clase `Adaboost` y su constructor, su método de entrenamiento (enseñar al clasificador a clasificar correctamente) y su método de predicción (obtener el resultado de esa clasificación):

```
class Adaboost:
  ## Constructor de clase, con número de clasificadores e intentos por clasificador
  def __init__(self, T=5, A=20):
    # Dar valores a los parámetros del clasificador e iniciar la lista de clasificadores débiles vacía
    return self
  ## Método para entrenar un clasificador fuerte a partir de clasificadores débiles mediante Adaboost
  def fit(self, X, Y, verbose = False):
    # Obtener el número de observaciones y de características por observación de X
    # Iniciar pesos de las observaciones a 1/n_observaciones
    # Bucle de entrenamiento Adaboost: desde 1 hasta T repetir
      # Bucle de búsqueda de un buen clasificador débil: desde 1 hasta A repetir
        # Crear un nuevo clasificador débil aleatorio
        # Calcular predicciones de ese clasificador para todas las observaciones
        # Calcular el error: comparar predicciones con los valores deseados
        # y acumular los pesos de las observaciones mal clasificadas
        # Actualizar mejor clasificador hasta el momento: el que tenga menor error
      # Calcular el valor de alfa y las predicciones del mejor clasificador débil
      # Actualizar pesos de las observaciones en función de las predicciones, los valores deseados y alfa
      # Normalizar a 1 los pesos
      # Guardar el clasificador en la lista de clasificadores de Adaboost
  ## Método para obtener una predicción con el clasificador fuerte Adaboost
  def predict(self, X):
    # Calcular las predicciones de cada clasificador débil para cada input multiplicadas por su alfa
    # Sumar para cada input todas las predicciones ponderadas y decidir la clase en función del signo
```

A lo largo de esta sesión completaremos este código. Pero antes tendremos que programar una clase para nuestros clasificadores débiles, que serán los más sencillos: clasificadores de umbral, también llamados tocones de decisión (del inglés *decision stump*), que son árboles de decisión de profundidad 1 y (a priori) sin método de entrenamiento propio (solo generación aleatoria). En el siguiente recuadro tienes el pseudocódigo de un clasificador de umbral sencillo de este tipo. Cópialo en tu archivo .py antes del código de la clase `Adaboost` y complétalo.

```
class DecisionStump:
  ## Constructor de clase, con número de características
  def __init__(self, n_features):
    # Seleccionar al azar una característica, un umbral y una polaridad.
    return self
  ## Método para obtener una predicción con el clasificador débil
  def predict(self, X):
    # Si la característica que comprueba este clasificador es mayor que el umbral y la polaridad es 1
    # o si es menor que el umbral y la polaridad es -1, devolver 1 (pertenece a la clase)
    # Si no, devolver -1 (no pertenece a la clase)
```

Ten en cuenta que X e Y serán arrays de Numpy, y que las funciones de predicción también devolverán arrays de la misma longitud. Tras completar el código de ambas clases, comprueba que todos los módulos devuelven los resultados esperados. Se valora que el código esté correctamente comentado. La documentación que requieren esta tarea y la siguiente es breve, solo hace falta comentar aspectos relevantes del desarrollo del código y de cómo usarlo, y mostrar los resultados. No incluyas el código entero en la documentación, solo fragmentos sobre los que quieras explicar algo y que creas que no sobra con comentarlo en el código.

Tarea 1B (OBLIGATORIA): Mostrar resultados de tu clasificador adaboost

Una vez tengas las clases `DecisionStump` y `Adaboost` programadas, podrás probar sus capacidades de aprendizaje automático. Crea en tu archivo una función con los parámetros `clase` (uno de los 10 dígitos), `T` y `A`, que cargue los datos de MNIST, entrene tu clasificador adaboost con el conjunto de entrenamiento `X_train`, e imprima por pantalla las tasas de acierto (porcentaje de predicciones que coinciden con las correctas) con `X_train` y con `X_test`. Si tu función incluye más `print`s que esos, incluye un parámetro `verbose` que por defecto esté a `False` para que solo se muestren los `print`s extra cuando se ponga explícitamente a `True`.

```
user@user:~/GInf-SI/Prácticas/P2$ python3.8 lucas_martinez_bernabeu.py
Entrenando clasificador Adaboost para el dígito 9, T=20, A=10
Entrenando clasificadores de umbral (con dimensión, umbral, dirección y error):
Añadido clasificador  1:  59, 0.4354, +1, 0.099150
Añadido clasificador  2: 274, 0.2253, -1, 0.456836
Añadido clasificador  3: 524, 0.2999, -1, 0.417796
...
Añadido clasificador 18: 632, 0.0881, -1, 0.444582
Añadido clasificador 19: 595, 0.4053, -1, 0.445673
Añadido clasificador 20: 140, 0.7672, +1, 0.422934
Tasas acierto (train, test) y tiempo: 89.88%, 89.73%, 0.089 s.
```

Sesión 3: Ajuste de los parámetros de entrenamiento y clasificador multiclase a partir de clasificadores binarios

Como has visto, para entrenar tu adaboost tienes que darle valores a los parámetros T y A. En general, valores más altos permitirán resultados más acertados, siempre y cuando no se incurra en sobreentrenamiento. El sobreentrenamiento ocurre cuando el clasificador se ajusta tan bien a las características del conjunto de entrenamiento que pierde generalidad y obtiene peores resultados con el conjunto de prueba. Incluso si no se incurre en sobreentrenamiento, valores de T y A más altos siempre implican mayores tiempos de entrenamiento y costes computacionales. Por todo ello, no se puede dar valores muy altos sin más, se han de ajustar los parámetros para obtener buenos resultados y en un tiempo razonable. Para ello puedes apoyarte en gráficas que correlacionen estas variables y te permitan visualizar cómo interactúan.

Tarea 1C (OBLIGATORIA): Ajuste óptimo de T y A

Utilizando Matplotlib, genera gráficas de curvas que permitan relacionar el tiempo de entrenamiento con la tasa de acierto para distintos valores de T y A. La gráfica ideal (máxima nota) tiene los valores de uno de los parámetros en el eje horizontal, la tasa de acierto en el eje vertical izquierdo y el tiempo en el eje vertical derecho, y muestra para cada valor del otro parámetro dos curvas: una de la tasa de acierto referenciada al eje izquierdo y otra del tiempo referenciada al eje derecho. Puedes consultar en la web de Matplotlib o en tutoriales en otras webs ejemplos de uso con gráficas complejas.

En cuanto a los valores de T y A que pruebes, el objetivo es que sean suficientes y esparcidos de forma que te permitan deducir (y explicar en la documentación) qué efectos tiene en la tasa de acierto y el tiempo de entrenamiento los cambios en esos parámetros, y qué recomendaciones puedes dar sobre cómo ajustar los parámetros para conseguir los mejores resultados con la base de datos de MNIST. Si lo crees necesario, puedes hacer otras pruebas y mostrar otras gráficas que consideres útiles para entender los efectos de variar A y T (justifica razonadamente tus experimentos).

Apoyándote en las conclusiones que has sacado, encuentra una buena combinación de valores para T y A de forma que la tasa de acierto con el conjunto de test sea máxima, con la restricción de que T multiplicado por A sea igual o menor que 900 (por ejemplo, T=30 y A=30, o T=5 y A=180). Los valores que encuentres serán los que dejes listos para usar en el código que entregues. La calificación dependerá principalmente de la justificación que aportes y de los resultados que obtengas (la media entre las tasas de acierto con cada dígito de MNIST).

Tarea 1D (OBLIGATORIA): Clasificador multiclase

Como sabes de las clases de teoría y como hemos visto aquí, la salida de un clasificador binario solo permite indicar si la entrada pertenece o no a la clase que represente este clasificador. En el

caso de la clasificación de números manuscritos tenemos 10 clases distintas, por lo que un clasificador binario no es suficiente. En esta tarea te vas a apoyar en el clasificador binario que has implementado para componer un clasificador multiclase. Necesitaremos 10 clasificadores binarios independientes, cada uno encargado de predecir si una entrada pertenece o no a cada una de las 10 clases. Ten en cuenta que más de un clasificador podría dar +1 para una misma imagen, pero tu clasificador solo tiene que devolver una clase (del 0 al 9), así que necesitarás poder distinguir según la certeza de cada predicción. En el clasificador que has implementado en la tarea 1A la clasificación se devuelve como el signo del resultado, 1 o -1, a pesar de que los valores reales están entre +1 y -1 (un resultado de +0,001 se considera perteneciente a la clase igual que un resultado de +0,999, pero el primero es mucho más dudoso a ojos del clasificador). Tendrás que cambiar tu clase `Adaboost` para que devuelva los valores de las predicciones sin redondear, y que tu clasificador multiclase (que utiliza los clasificadores binarios) asigne una única clase a cada resultado a pesar de que más de un clasificador binario dé positivo o todos den negativo.

Tarea 1E (OPTATIVA): Mejoras creativas

La última tarea de esta parte de la práctica es un trabajo relativamente libre: mejora tu método adaboost para conseguir entrenamientos más eficaces (mejores tasas de acierto) y más rápidos.

Las únicas condiciones es que siga siendo un adaboost programado desde cero (puedes usar numpy y pandas para operar eficientemente con arrays de datos, pero no funciones y utilidades avanzadas como las que veremos en scikit en la segunda parte de esta práctica) y que utilices los conjuntos de entrenamiento y test sin modificar. Puedes aplicar preprocesamiento a las imágenes o implementar un clasificador débil más efectivo que los de umbral (p. ej. árboles de decisión o hiperplanos). Puedes aplicar mejoras en el propio algoritmo de adaboost siempre que siga siendo un algoritmo de adaptive boosting (p. ej. puedes mejorar la generación de los clasificadores débiles aleatorios, o hacer ajustes en la forma en que se añade cada nuevo clasificador, o introducir una forma de detectar el sobreentrenamiento). En la documentación has de incluir un resumen de la investigación que hagas sobre Adaboost y posibles mejoras (estudio del arte) y del desarrollo de tus modificaciones, y un análisis de los resultados que permita medir las ventajas y desventajas de tu Adaboost mejorado comparado con el Adaboost básico que has implementado en la primera tarea.

Recuerda: Entrega de la primera parte hasta el **3 de diciembre de 2023 a las 23:55h**.

La **no entrega** de este hito supone hasta un **20% de penalización** en la entrega final.

Sesión 4: Clasificador multiclase adaboost usando scikit-learn

En la primera parte de la práctica hemos trabajado con técnicas de aprendizaje automático, programándolas directamente en Python, y hemos podido sentir lo mismo que quienes ingeniaron estos métodos hace décadas. Hoy en día disponemos de librerías que ya traen implementadas muchos métodos de inteligencia artificial y ahorran mucho tiempo de implementación y depuración. En esta segunda parte vamos a experimentar con alguna de esas herramientas: scikit-learn, una librería general de técnicas de aprendizaje automático, y Keras, una librería de redes neuronales implementada sobre TensorFlow.

En esta sesión resolverás el mismo problema que en la primera parte, pero utilizando las utilidades de la librería scikit-learn, y practicarás con técnicas más avanzadas que los clasificadores de umbral.

Tarea 2A (OBLIGATORIA): Modela el clasificador adaboost con scikit-learn

En esta tarea has de documentarte en la web de scikit-learn sobre cómo utilizar la clase `AdaBoostClassifier`, e implementar una función en tu fichero de entrega para resolver el mismo problema que en la Tarea 1D, el clasificador multiclase, pero utilizando la clase de scikit-learn. Verás que vas a necesitar muchas menos líneas de código para hacer el mismo trabajo. Utiliza los valores por defecto del constructor de clase, y pásale como argumentos solo los conjuntos de entrenamiento (`X_train` e `Y_train`). Puedes documentar cuestiones relevantes sobre esta clase y cómo se utiliza.

Tarea 2B (OBLIGATORIA): Compara tu versión de adabost con la de scikit-learn y optimiza la configuración del clasificador débil por defecto

Dibuja una gráfica como la de la tarea 1C con las curvas de tasa de acierto y de tiempo de cada una de las versiones de Adaboost (1D, 1E si la has implementado, y 2A). Usa por ejemplo T={10, 20, 40} y A={10, 20, 40}. Para poder cambiar la configuración de `AdaBoostClassifier` tendrás que averiguar, si no lo has hecho ya, cómo se llama en scikit-learn a los parámetros que nosotros hemos llamado aquí T y A. Coméntalo en la documentación. Apoyándote en la gráfica, analiza y compara el rendimiento de ambos métodos, y analiza también las diferencias que aprecies en el tiempo de cómputo y la tasa de acierto alcanzada en función de los cambios en T y A, en una técnica y en la otra. El objetivo es que puedas responder a cuestiones como que si se consiguen resultados similares con parámetros similares en ambas técnicas o si tienen la misma importancia relativa T y A en ambas técnicas.

Con todo lo anterior, descubrirás que uno de los métodos es aparentemente mejor que el otro en términos de tasa de acierto alcanzada en la misma cantidad de tiempo. Ahora, apoyándote en la documentación de scikit-learn y los tutoriales que puedas consultar, encuentra una configuración de

`AdaBoostClassifier` y del clasificador débil que utiliza por defecto, y trata de conseguir mejores resultados (en términos de tasa de acierto y tiempo de entrenamiento) que los que has obtenido con tu implementación propia de Adaboost. Documenta los parámetros más relevantes de estas clases. En cualquier caso, razona por qué los cambios que hayas introducido en la configuración por defecto obtienen mejores resultados.

Tarea 2C (OPTATIVA): Sustituye el clasificador por árboles de decisión

En la tarea anterior habrás encontrado información sobre la clase `DecisionTreeClassifier` de scikit-learn. En esta tarea te permitimos explorar todo el potencial de ese tipo de clasificador (que no es necesariamente débil) utilizándolo dentro de tu clasificador `AdaBoostClassifier`. Básicamente tienes que repetir lo que has hecho en la tarea 2A sustituyendo el clasificador débil por defecto por un árbol de decisión completo. Tendrás que buscar una buena configuración de los parámetros de ese tipo de clasificador. En la documentación explica el funcionamiento general y los elementos principales de un árbol de decisión, y comenta los parámetros más relevantes de esa clase en scikit-learn, y cómo los has ajustado para conseguir el mejor resultado. Apóyate en gráficas para refrendar tus decisiones.

Sesión 5: Clasificador multiclase con redes neuronales usando Keras

Hasta el momento en esta práctica hemos estado usando la técnica de adaboost. En esta sesión trabajaremos con redes neuronales para resolver una vez más el problema de reconocer los dígitos manuscritos de MNIST.

Tarea 2D (OBLIGATORIA): Modela un clasificador MLP para MNIST con Keras

En las clases de teoría se han visto los perceptrones multicapa. Afortunadamente para ti, este año la parte principal de la práctica es sobre adaboost y no vas a tener que programar un perceptrón multicapa en Python con su algoritmo de *backpropagation*. En esta tarea vas a implementar un MLP usando la librería Keras. Puedes documentarte en la web de Keras sobre cómo programar en Python con muy pocas líneas de código un perceptrón multicapa. Al igual que en la tarea 2D, deberás explicar sus parámetros más relevantes y buscar una buena configuración de los mismos (en términos de tasa de acierto y tiempo de cómputo finales). Puedes documentar las distintas configuraciones que pruebes y los experimentos que realices para descubrir los efectos de variar los parámetros de un MLP de keras. En la documentación has de incluir una tabla donde se resuman, de forma que lo entienda un profano, para qué sirven y qué efectos tiene subir o bajar los parámetros que se mencionan en el párrafo siguiente (número de capas y de neuronas inclusive).

Lo primero que harás será decidir la estructura de tu MLP: las entradas que deberá tener (y en qué forma) y el número de neuronas en la capa de salida dependerá de los datos de entrada (`X_train` e `Y_train`). La cantidad de capas ocultas y de neuronas por capa oculta dependerá de la complejidad del problema a resolver. Puedes empezar utilizando los valores por defectos de los parámetros `batch_size` y `validation_split`, y una sola capa oculta con el número de neuronas que te parezca oportuno, utilizando ReLu o sigmoid como funciones de activación en la capa oculta y softmax en la capa de salida. Utiliza siempre el optimizador Adam. Una vez que funcione, podrás experimentar con el número de neuronas y de capas y con otros valores de los parámetros del MLP. También puedes experimentar con el parámetro `learning_rate` del optimizador.

Tarea 2E (OPTATIVA): Modela un clasificador mediante CNN para MNIST con Keras

En esta tarea, muy parecida a la anterior, volverás a clasificar los dígitos de MNIST con redes neuronales, pero esta vez utilizando redes convolucionales (CNN, de *convolutional neural networks*), que son modelos de redes neuronales más adaptados a problemas de visión artificial. Para completar esta tarea necesitarás documentarte sobre las CNN y buscar ejemplos de implementación utilizando Keras. Encontrarás varios en la propia web de Keras. La documentación ha de incluir los mismos aspectos que en la tarea anterior, pero en esta se valorará que además resumas las características de las CNN y sus ventajas sobre los MLP estándar.

Sesión 6: Comparativa de técnicas

Tarea 2F (OBLIGATORIA): Realiza una comparativa de los modelos implementados

En esta tarea has de hacer algo similar a lo que has hecho en la tarea 2B, pero en este caso has de realizar una breve comparativa de resultados entre todos los clasificadores multiclase que hayas implementado en las tareas anteriores: 1D, 1E (optativa), 2A, 2B, 2C (optativa), 2D y 2E (optativa). Debes realizar un análisis comparativo del rendimiento de cada clasificador, de forma que se pueda responder a la pregunta de qué clasificador es mejor para el problema que estamos resolviendo. También puedes discutir cuestiones sobre la facilidad de uso de cada método (particularmente de configuración de sus parámetros para alcanzar los mejores resultados), y dar recomendaciones sobre cómo lograr los mejores resultados con el problema de MNIST. Discute también si se ha producido sobreentrenamiento en alguno de los entrenamientos que has realizado en cualquiera de los apartados anteriores.

Sesión 7: Resolución de dudas y revisión de la documentación

En esta sesión resolveremos dudas que puedan quedar sobre la implementación de cualquiera de los apartados antes de la entrega final, y repasaremos las cuestiones que se han de tratar y las que no en la documentación.

AVISO IMPORTANTE

No cumplir cualquiera de las normas de entrega descritas al principio de este documento puede suponer un suspenso de la práctica.

Las prácticas son individuales. No se pueden hacer en pareja o grupos.

Cualquier código copiado supondrá un suspenso de la práctica para todos los estudiantes implicados y **se informará a la dirección de la EPS para la toma de medidas oportunas** (Reglamento para la Evaluación de Aprendizajes de la Universidad de Alicante, BOUA 9/12/2015, y documento de Actuación ante copia en pruebas de evaluación de la EPS).