Práctica 2: Visión Artificial y Restricciones

Sistemas Inteligentes

—

Alejandro Guillén Merino - 48790456G
Universidad de Alicante
Escuela Politécnica Superior

Índice

Objetivos

A lo largo de esta documentación se va a explicar el proceso de implementación de clasificadores adaboost para imágenes de números manuscritos, tanto escritos en python como usando scikit-learn, y un clasificador MLP con Keras.

Además, se mostrarán comparativas entre los diferentes modelos y se comentará como lograr resultados óptimos con cada uno de ellos y que ocurre con cada uno de ellos, así como resolver la pregunta de qué modelo es mejor para el problema que se está resolviendo.

Sesión 1: Introducción al problema y entorno de trabajo

I. Trabajando con las imágenes de MNIST

El problema al que se le va a tratar de dar solución es la clasificación de imágenes con dígitos manuscritos mediante el uso de clasificadores. Para ello, se han utilizado las imágenes de MNIST.

Éstas son un array de 60.000 elementos, que se dividen en arrays de 28x28, que son las propias imágenes. Éstas se encuentran en X_train, mientras que en Y_train se encuentra otro array de 60.000 elementos.

Los arrays X_test y Y_test son similares a los anteriores, pero con tan solo 1.000 observaciones, ya que la función de los anteriores es entrenar (train), mientras que la de estos es probar el funcionamiento (test).

Sesión 2: Adaboost binario con clasificadores de umbral

I. Tarea 1A: Implementación de Adaboost y DecisionStump

- DecisionStump:

Se trata del clasificador débil. Su implementación es sencillas, aunque hay algunos aspectos a tener en cuenta:

```python
class DecisionStump:
    def __init__(self, n_features):
        self.caracteristica = random.randint(0, n_features - 1)
        self.umbral = np.random.rand()
        self.polaridad = np.random.choice([-1, 1])

    def predict(self, X):
        caracteristicas = X[:, self.caracteristica]
        predicciones = np.where(self.polaridad * caracteristicas > self.polaridad * self.umbral, 1, -1)
        return predicciones
```

Imágen 1: Implementación de DecisionStump

Para la implementación se deben seleccionar en __init__ elementos de manera aleatoria. Éstos están definidos, en el caso de la característica, entre 0 y el número de características del clasificador (se explica más adelante) y entre -1 y 1 para la polaridad.

Por otro lado, el método predict se ha implementado haciendo uso de la librería numpy en lugar de hacer un bucle normal ya que de esta manera se reduce significativamente el tiempo de ejecución.

- Adaboost Binario:

Por otro lado se ha implementado el clasificador Adaboost Binario, que será el encargado de clasificar las imágenes. Para ello, cuenta con un método para entrenarlo (fit()) y otro para recibir las predicciones (predict()).

```python
for a in range(self.A):
    # Crear un nuevo clasificador débil aleatorio
    clasificadorDebil = DecisionStump(n_caracteristicas)
    clasificadoresT.append(clasificadorDebil)

    # Calcular predicciones de ese clasificador para todas las observaciones
    prediccion = clasificadorDebil.predict(X)

    # Calcular el error: comparar predicciones con los valores deseados y acumular los pesos de las observaciones mal clasificadas
    error = np.sum(pesos * (prediccion != Y))
    erroresT.append(error)

    # Actualizar mejor clasificador hasta el momento: el que tenga menor error
    if error < mejorError:
        mejorClasificador = clasificadorDebil
        mejorPrediccion = prediccion
        mejorError = error
```

Imagen 2: Fragmento método fit() adaboost

En este fragmento se puede observar cómo se recorre todo A y en cada iteración se crea un nuevo clasificador débil (decisionStump) anteriormente visto, y se obtiene la predicción del mismo, obteniendo el error y así sabiendo si nos encontramos antes una mejor opción que en algún caso anterior.

Con estas dos clases creadas ya se puede comenzar a clasificar imágenes, aunque tan solo de manera binaria, es decir, por ahora se puede decir si la imagen aportada contiene o no el número proporcionado.

II. Tarea 1B: Mostrar resultados del clasificador adaboost

Para mostrar los resultados y comprobar el funcionamiento, se cargan los datos de entrenamiento de MNIST y se filtran por la clase (número) deseado. Se indican una T y una A (intentos y clasificadores) y se crea una instancia de la clase Adaboost, que se crea con los valores antes indicados. A continuación, se invoca al método fit con los valores X_train e Y_train para entrenar el clasificador. Por último, una vez entrenado el clasificador, se invoca al método predict para que devuelva una predicción de los valores de X_test e Y_test.

A continuación muestro el resultado de una ejecución:

```
alejandro@agm-HP-Laptop:~/Documentos/3º Informática/SI/Practica 2/Código$ python3 Alejandro_Guillén_Merino.py
Entrenando clasificador Adaboost para el digito = 9 con T = 15 y A = 15...
Tiempo de entrenamiento: 0.10 segundos
Tasa de acierto (train, test) y tiempo: 88.57%, 88.88%, 0.104 s.
```

Imagen 3: Tasa de acierto del clasificador binario

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

I. Tarea 1C: Ajuste óptimo de T y A

Para establecer cuales son los valores óptimos para T y A se ha pedido realizar una gráfica, como la que se muestra a continuación:

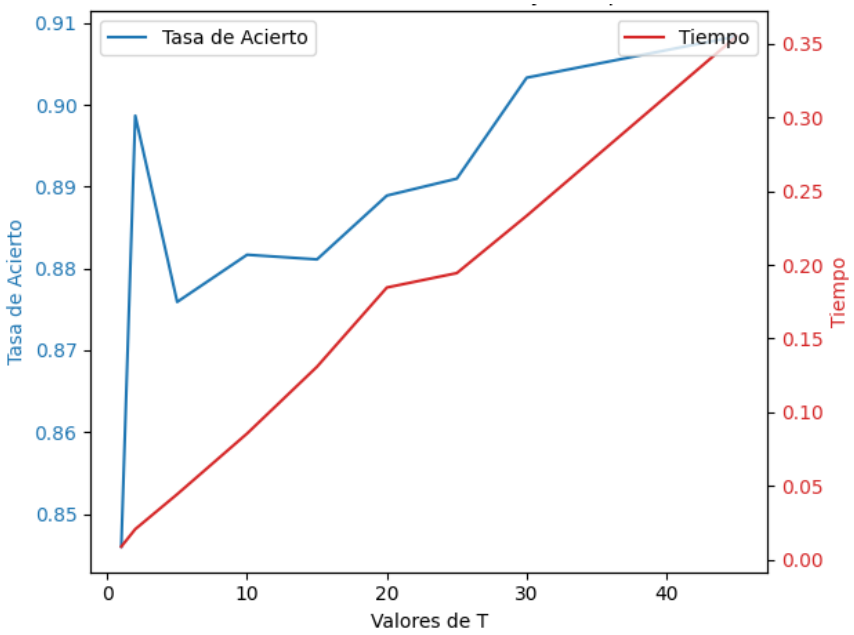


Imagen 4: Gráfica obtenida al ejecutar el código

En la gráfica se obtiene el tiempo y la tasa de acierto de la media de los valores de A para cada valor de T.

De ésta gráfica se pueden deducir varias cosas:

- Cuánto mayor es T, más tiempo es necesario para completar el problema. Esto es debido a que se deben realizar más intentos para cada clasificador.
- La tasa de aumento es mayor cuanto más intentos se realizan para cada clasificador débil, sin llegar a conllevar una cantidad excesiva de tiempo.
- El valor de T óptimo será uno grande, también dependiendo de cómo influyen los valores de A.

Los valores de A representan los clasificadores débiles, que no es necesario que sean muchos. Por tanto, se estima que el resultado óptimo se puede obtener con los valores T=4250 y A=2, siendo T * A <= 900.

II. Tarea 1D: Clasificador Multiclase

En esta tarea se va a implementar un clasificador multiclase, ya que para clasificar los números del 1 al 9 no es suficiente con uno binario. El funcionamiento del clasificador multiclase es el siguiente, teniendo en cuenta que solo requiere de algunas modificaciones con respecto al clasificador binario:

- Se deben definir todas las clases en un array, para poder crear un clasificador adaboost binario independiente para cada una.
- Se debe modificar la forma de cargar los datos de MNIST, ya que de la manera que estaba anteriormente implementado tan solo devolvía los datos filtrados de una clase, mientras que ahora obtiene los de todas las clases, cada uno en una posición de un array.
- Se entrena el clasificador adaboost para cada dígito y se obtiene, posteriormente, una predicción, teniendo en cuenta que cada imagen tan solo puede tener asignada una clase, aunque de positiva en más de una. Por ello se modifica la predicción, haciendo que en vez de redondear a +1 y -1, se adopten valores entre ambos números, permitiendo valorar cual de las imágenes es más probable que sea de la clase según si es más cercana a un valor u otro. Por ejemplo, si para una imagen hay dos clases, la primera indica 0.54 y la segunda 0.001, es más probable que la imagen pertenezca a la primera clase.

Con estas modificaciones, se obtiene el siguiente resultado:

```
****Inicio de tarea_1D_adaboost_multiclase****
Entrenando clasificador Adaboost para el digito = 0 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 1 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 2 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 3 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 4 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 5 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 6 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 7 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 8 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 9 con T = 450 y A = 2
Tasa de acierto (train, test) y tiempo: 80.78%, 81.93%, 6.198 s.
****Fin de tarea_1D_adaboost_multiclase****
```

Imagen 5: Tasa de acierto del clasificador multiclase

Sesión 4: Clasificador multiclase adaboost usando scikit-learn

I. Tarea 2A: Modelar clasificador adaboost con scikit-learn

Para modelar el clasificador adaboost con scikit-learn se ha consultado el manual de uso de scikit y otras fuentes. Y aunque es mayormente recomendado implementar el clasificador débil con DecisionTree, al pertenecer a un apartado posterior se ha omitido y se ha utilizado SVC (Support Vector Classification).

Además, se han definido el número de estimadores, fijado en 2, y se ha reducido el tamaño del entrenamiento (X_train, Y_train) a causa del largo tiempo que tardaba. Se ha optimizado, para obtener un resultado con un buen porcentaje de acierto y un tiempo razonable, un subconjunto de tamaño 2000.

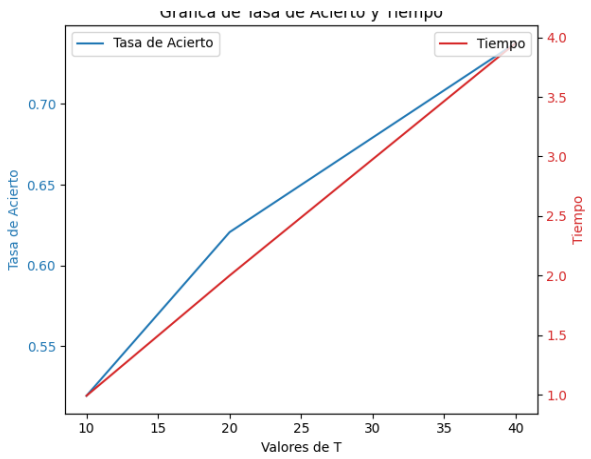
```
****Inicio de tarea_2A_AdaBoostClassifier_default****
Tiempo de entrenamiento: 6.65 segundos
Tasa de acierto (train, test) y tiempo: 95.10%, 88.30%, 6.655 s.
****Fin de tarea_2A_AdaBoostClassifier_default****
```

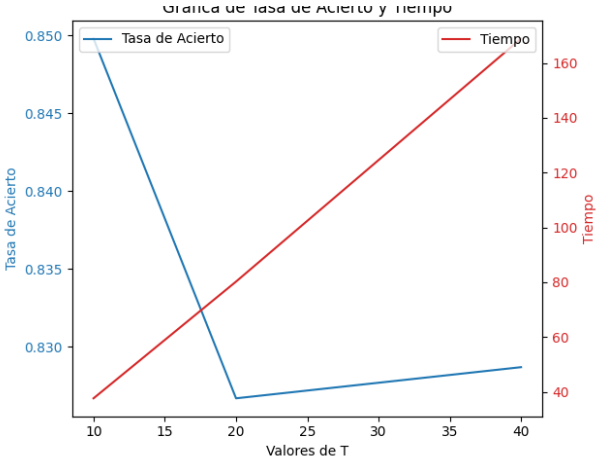
Imagen 6: Tasa de acierto clasificador con scikit-learn

Como se puede observar, con estos parámetros se gastan 6 segundos en entrenamiento y se consigue entre un 85% y un 95% de tasa de acierto.

II. Tarea 2B: Comparación de las implementaciones de adaboost

De las gráficas obtenidas en este apartado se pueden obtener conclusiones bastante interesantes, además de poder analizar las diferencias entre ambos modelos y deducir cual es mejor, al menos a mi parecer. Las gráficas son las siguientes:





Imágenes 7 y 8: Gráfica de la tarea 1D (Izquierda) y gráfica de la tarea 2A (derecha)

Bueno, a simple vista se observan varias diferencias.

Una, a la que no he conseguido encontrar explicación tras leer documentación, ver vídeos y consultar alguna IA es por que el clasificador con scikit-learn baja de repente su tasa de acierto.

Por otro lado, es evidente comparar las tasas de acierto, ya que en el adaboost multiclase las tasas de acierto no son muy altas, llegando al 75% en el rango de valores probados, mientras que con scikit-learn las tasas no bajan del 82% para estos valores.

Sin embargo, otro punto a tener en cuenta es el tiempo que conlleva cada operación, y es que el clasificador implementado con scikit-learn es muy lento, llegando a tardar 160 segundos para A=40 (o n_estimators en su caso), mientras que el tiempo máximo de adaboost multiclase es de apenas 4 segundos.

Por tanto, las diferencias son obvias. Con adaboost multiclase se obtienen peores resultados para este rango pero a un costo temporal muy inferior que con scikit-learn, aunque este último tarda mucho más.

III. Tarea 2C: Sustituir clasificador débil por árboles de decisión con scikit-learn

Para definir un árbol de decisiones se debe eliminar el clasificador débil SVC. Este nuevo clasificador contiene un parámetro para indicar la profundidad máxima, llamado max_depth. Esto indica la profundidad máxima que puede tener el árbol, es decir, limita la expansión del árbol.

Se debe tener en cuenta que a mañor profundidad del árbol, más preciso será el resultado ya que se podrán barajar más opciones y tomar una decisión más acotada.

Además, DecisionTreeClassifier cuenta con otros parámetros, como “splitter”, que determina la estrategia a seguir, o “criterion”, que especifica la función para medir la calidad de una división (en un nodo). Estos parámetros no han sido utilizados en mi implementación.

A continuación, se muestra una imagen con el resultado obtenido de poner la profundidad máxima del árbol a 2. Esto debido a que si se establece la profundidad más alta, el porcentaje aumenta, pero el tiempo crece exponencialmente, por lo que empieza a no valer tanto la pena, en mi opinión.

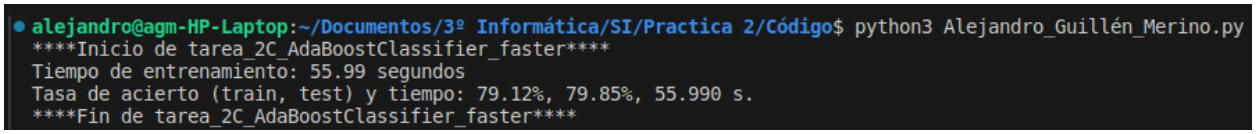
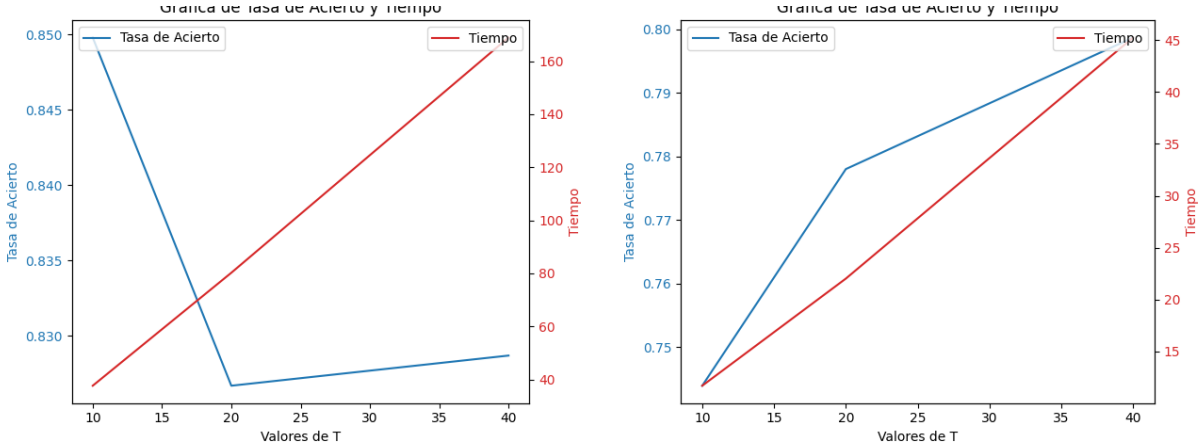


Imagen 9: tasa de acierto del clasificador con DecisionTreeClassifier como clasificador débil

Por último, voy a comparar el primer clasificador implementado con scikit-learn, que tenía un clasificador débil SVC y el actual con un árbol de decisión.



Imágenes 10 y 11: Gráfica de la tarea 2A (Izquierda) y gráfica de la tarea 2C (derecha)

En esta ocasión se presenta una similar a la anterior. La tasa de acierto con DecisionTreeClassifier no es muy elevada. Sin embargo, tiene un tiempo significativamente más bajo.

Se nota, ya por la constante y por haber probado un valor superior en la prueba anterior, que la tasa de acierto del árbol de decisiones se empieza a estancar según llega al 80%. Esto nos indica que se pueden conseguir tasas de acierto muy buenos con tiempos no muy altos, al contrario de lo que pasa con SVC.

Sin duda, como ya se ha ido viendo, la mayor desventaja que tiene la implementación de scikit-learn sin el árbol de decisiones es su alto coste temporal.

Sesión 5: Clasificador multiclase con redes neuronales usando Keras

I. Tarea 2D: Clasificador MLP para MNIST con Keras

Tras implementar la red neuronal y experimentar con ella sus parámetros, estructura y resultados obtenidos, voy a hacer un análisis de las conclusiones a las que he llegado:

- La estructura se basa en una red de neuronas, de entrada, salida y una serie de capas ocultas por dónde pasa la información.
- ReLU (Rectified Linear Unit) es una función que devuelve 0 para las entradas negativas, siendo lineal para los valores positivos y no lineal para los negativos. Además, es fácil de calcular y permite un entrenamiento rápido.

- SoftMax: Es una función de activación utilizada en la capa de salida para problemas de clasificación multiclase, como la clasificación de imágenes de MNIST. Convierte un vector de números reales en una distribución de probabilidad, asignando probabilidades a cada clase.
- Epoch representa una pasada completa a través de todos los datos de entrenamiento durante el proceso de entrenamiento.
- El *batch_size* es la cantidad de ejemplos de entrenamiento que se utilizan en cada iteración durante el proceso de entrenamiento. Es la cantidad de datos que se procesan a la vez antes de actualizar los pesos del modelo.
- Validation_split especifica la fracción de los datos de entrenamiento que se reservará como conjunto de validación, por lo que no es recomendable que sea un valor muy elevado ya que se debe entrenar bien la red para dar un resultado óptimo.
- Las capas ocultas y las neuronas por capa. Esta es una parte importante, ya que se debe ajustar bien para lograr tener un resultado óptimo. Tener pocas capas puede resultar en mucha pérdida, y tener demasiadas puede conllevar un sobreajuste y un aumento de tiempo importante.

Después de repasar estos datos y de experimentar con estos parámetros, me he decidido por una capa de 256 neuronas, otra de 128 y una de salida de 10 neuronas, y estableciendo los valores de batch_size en 128 y validation_split en 0.2 para no perder muchos datos. Con esto, he obtenido el siguiente resultado:

```
alejandro@agm-HP-Laptop:~/Documentos/3º Informática/SI/Practica 2/Código$ python3 Alejandro_Guillén_Merino.py
****Inicio de tarea_2D_MLP_Keras****
Epoch 1/10
375/375 [==============================] - 1s 2ms/step - loss: 0.3053 - accuracy: 0.9122 - val_loss: 0.1395 - val_accuracy: 0.9604
Epoch 2/10
375/375 [==============================] - 1s 2ms/step - loss: 0.1135 - accuracy: 0.9662 - val_loss: 0.1065 - val_accuracy: 0.9691
Epoch 3/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0734 - accuracy: 0.9779 - val_loss: 0.0881 - val_accuracy: 0.9741
Epoch 4/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0524 - accuracy: 0.9836 - val_loss: 0.0878 - val_accuracy: 0.9728
Epoch 5/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0376 - accuracy: 0.9877 - val_loss: 0.0780 - val_accuracy: 0.9774
Epoch 6/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0294 - accuracy: 0.9906 - val_loss: 0.0902 - val_accuracy: 0.9732
Epoch 7/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0225 - accuracy: 0.9932 - val_loss: 0.0858 - val_accuracy: 0.9757
Epoch 8/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0175 - accuracy: 0.9945 - val_loss: 0.0894 - val_accuracy: 0.9753
Epoch 9/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0135 - accuracy: 0.9958 - val_loss: 0.1032 - val_accuracy: 0.9733
Epoch 10/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0137 - accuracy: 0.9954 - val_loss: 0.0831 - val_accuracy: 0.9799
313/313 [==============================] - 0s 645us/step - loss: 0.0757 - accuracy: 0.9801
Tasa de acierto y tiempo: 98.01%, 8.269 s.
***Fin de tarea_2D_MLP_Keras***
```

Imagen 12: Resultado del clasificador MLP

Sesión 6: Comparativa de técnicas

I. Tarea 2F: Comparativa de los modelos implementados

Este apartado me sirve también a modo de conclusión. Se han probado varios modelos a lo largo de esta práctica. A continuación, voy a citar lo mejor y peor de ellos y voy a decir, desde mi punto de vista, cual creo que es el mejor o me ha resultado mejor para lograr la mejor solución con el problema MNIST.

- Adaboost Binario (clase Adaboost): Es la primera implementación, con la que he comprendido el funcionamiento de todo al implementarlo yo mismo. Sin embargo, eso también es su mayor pega, ya que ha llevado bastante tiempo y es un poco complejo. A nivel tasa de acierto, consigue buenos resultados con un buen tiempo.
- Implementación con scikit-learn: Es muy sencillo de implementar y proporciona buenos resultados, aunque es tiempo que tarda es excesivo.
- Implementación con árbol de decisiones. La implementación es igual de sencilla que en el caso anterior y si bien no logra tener unas tasas de acierto tan buenas como el anterior, vale más la pena por su bajo tiempo de entrenamiento.
- Red neuronal con keras. Es muy sencillo de implementar. Quizá su pero característica son todos los parámetro y datos que se pueden ajustar, aunque esto también permite crear un caso específico para cada problema que se presente. Además, los resultados que se obtienen son excelentes y en un tiempo muy bajo.

Respecto a cómo lograr mejores resultados para MNIST en cada modelo, se ha ido resolviendo a lo largo del informe según se iba hablando de cada uno de ellos.

Después de analizar cada uno por separado, puedo afirmar que, desde mi propia experiencia, el mejor método para solucionar el problema MNIST es la red neuronal MLP, por su bajo coste temporal, su alta tasa de acierto y su facilidad de implementación así como todas las variables para ajustar.

Por último, muestro una imagen de todos los mensajes que deben aparecer en la terminal en una ejecución completa de todas las tareas del código:

```
alejandro@agm-HP-Laptop:~/Documentos/3º Informática/SI/Practica 2/Código$ python3 Alejandro_Guillén_Merino.py
****Inicio de tareas_1A_y_1B_adaboost_binario****
Entrenando clasificador Adaboost para el digito = 9 con T = 15 y A = 15...
Tiempo de entrenamiento: 0.11 segundos
Tasa de acierto (train, test) y tiempo: 88.05%, 88.18%, 0.112 s.
****Fin de tareas_1A_y_1B_adaboost_binario****

****Inicio de tarea_1C_graficas_rendimiento****

Se ha guardado la gráfica de entrenamiento en el archivo Grafica_Entrenamiento.png
Se han establecido los valores de T=425 y A=2 como los óptimos

****Fin de tarea_1C_graficas_rendimiento****

****Inicio de tarea_1D_adaboost_multiclase****
Entrenando clasificador Adaboost para el digito = 0 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 1 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 2 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 3 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 4 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 5 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 6 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 7 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 8 con T = 450 y A = 2
Entrenando clasificador Adaboost para el digito = 9 con T = 450 y A = 2
Tasa de acierto (train, test) y tiempo: 80.05%, 80.80%, 6.448 s.
****Fin de tarea_1D_adaboost_multiclase****

****Inicio de tarea_2A_AdaBoostClassifier_default****
Tiempo de entrenamiento: 6.57 segundos
Tasa de acierto (train, test) y tiempo: 95.10%, 88.30%, 6.568 s.
****Fin de tarea_2A_AdaBoostClassifier_default****

****Inicio de tarea_2B_graficas_rendimiento****
Se ha guardado la gráfica de test en el archivo Grafica_Test_1D.png
Se ha guardado la gráfica de test en el archivo Grafica_Test_2A.png
****Fin de tarea_2B_graficas_rendimiento****

****Inicio de tarea_2C_AdaBoostClassifier_faster****
Tiempo de entrenamiento: 53.49 segundos
Tasa de acierto (train, test) y tiempo: 79.12%, 79.85%, 53.493 s.
****Fin de tarea_2C_AdaBoostClassifier_faster****

****Inicio de tarea_2C_graficas_rendimiento****
Se ha calculado la gráfica de test de 2A anteriormente, no se vuelve a calcular
Se ha guardado la gráfica de test en el archivo Grafica_Test_2C.png
***Fin de tarea_2C_graficas_rendimiento***
```

```
****Inicio de tarea_2D_MLP_Keras****
Epoch 1/10
375/375 [==============================] - 1s 3ms/step - loss: 0.3033 - accuracy: 0.9127 - val_loss: 0.1536 - val_accuracy: 0.9559
Epoch 2/10
375/375 [==============================] - 1s 2ms/step - loss: 0.1185 - accuracy: 0.9643 - val_loss: 0.1040 - val_accuracy: 0.9703
Epoch 3/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0779 - accuracy: 0.9760 - val_loss: 0.0885 - val_accuracy: 0.9741
Epoch 4/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0540 - accuracy: 0.9839 - val_loss: 0.0993 - val_accuracy: 0.9698
Epoch 5/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0413 - accuracy: 0.9870 - val_loss: 0.0835 - val_accuracy: 0.9745
Epoch 6/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0292 - accuracy: 0.9913 - val_loss: 0.0816 - val_accuracy: 0.9761
Epoch 7/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0248 - accuracy: 0.9923 - val_loss: 0.0921 - val_accuracy: 0.9743
Epoch 8/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0169 - accuracy: 0.9951 - val_loss: 0.0914 - val_accuracy: 0.9766
Epoch 9/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0141 - accuracy: 0.9956 - val_loss: 0.1023 - val_accuracy: 0.9738
Epoch 10/10
375/375 [==============================] - 1s 2ms/step - loss: 0.0169 - accuracy: 0.9945 - val_loss: 0.0925 - val_accuracy: 0.9772
313/313 [==============================] - 0s 759us/step - loss: 0.0810 - accuracy: 0.9781
Tasa de acierto y tiempo: 97.81%, 9.183 s.
***Fin de tarea_2D_MLP_Keras***
```

Bibliografía

- OpenAI (ChatGPT-3.5) - https://chat.openai.com/
- Scikit-learn - https://scikit-learn.org/stable/user_guide.html
- DataCamp - https://www.datacamp.com/tutorial/decision-tree-classification-python
- GitHub Copilot - https://github.com/features/copilot