Fundamentos de Aprendizaje Automático 2024/2025

PRÁCTICA DE TRATAMIENTO DE DATOS, PARTICIONAMIENTO Y DISEÑO PRELIMINAR DE LA APLICACIÓN

1. Objetivos

El objetivo de esta práctica es desarrollar la estructura de clases principales de la aplicación que servirán como elemento de control para las prácticas posteriores. Dentro de esta estructura se encuentra, por un lado, la implementación de la clase *Datos* que se empleará durante el curso para gestionar los diferentes datasets con los que probar los algoritmos de clasificación a estudiar y, por otro, la implementación de las clases encargadas de las estrategias de *Particionamiento* y los detalles comunes a todos los *Clasificadores*.

2. Preliminares

Las prácticas se realizan en Python y se puede emplear la distribución que se considere oportuna, teniendo en cuenta que necesitaremos al menos los paquetes Numpy y Scikit-learn. Pandas es también muy recomendable. Por otro lado, **todas las entregas deben incluir un Jupyter Notebook** que presente de forma conjunta las implementaciones de código y la discusión de los resultados obtenidos. **No es necesaria una memoria en otro formato**.

3. Clase Datos

Los algoritmos de aprendizaje automático permiten construir modelos a partir de un conjunto de datos (datasets). Estos datasets se proporcionarán ya adaptados en Moodle para su uso en las prácticas. A grandes rasgos son ficheros en formato csv, donde la primera fila se destina al nombre de los atributos y la clase.

Como primer ejemplo, en esta práctica se proporcionan en *Moodle* los ficheros correspondientes a los conjuntos de datos ***balloons*** y ***heart disease.***

En primer lugar nos centraremos en implementar el tratamiento preliminar de estos dataset mediante la clase *Datos*, que leerá los datos del fichero de entrada y almacenará la información necesaria para su posterior uso por los algoritmos de aprendizaje automático y métodos de particionado. Una posible estructura de implementación se comenta en el siguiente apartado. Una vez que el procesamiento de datos esté completado, pasaremos a la implementación de las mencionadas estrategias de particionamiento y de los métodos de clasificación y entrenamiento comunes a todos los modelos de clasificación.

Con el objetivo de desarrollar una aplicación lo más flexible y general posible se plantea la siguiente estructura para la clase *Datos*:

```
import pandas as pd
import numpy as np
```

```
class Datos:

  # Constructor: procesar el fichero para asignar correctamente las
variables nominalAtributos, datos y diccionario
  def __init__(self, nombreFichero):

  # Devuelve el subconjunto de los datos cuyos índices se pasan como
argumento
  def extraeDatos(self, idx):
```

La mayor parte del procesamiento de los datos se realiza en el constructor, donde se abrirá el fichero original y se tratará para construir las estructuras en memoria más apropiadas para la clasificación. El método *extraeDatos* recibirá una lista de índices correspondientes a un subconjunto de patrones a seleccionar sobre el conjunto total de datos. El método devolverá la submatriz de datos que corresponde a los índices pedidos.

Se pueden definir en la clase *Datos* los atributos que se consideren necesarios, pero no debe modificarse la signatura de los métodos. Se recomiendan al menos los siguientes atributos:

- **`nominalAtributos`**: Lista de valores booleanos con la misma longitud que el número de atributos del problema (incluyendo la clase) que contendrá *True* en caso de que el atributo sea nominal (discreto) y *False* en caso contrario (numérico). Esta estructura será útil posteriormente en otros métodos. En los datasets proporcionados trabajaremos generalmente con tres tipos de datos: nominales y números enteros o reales. En función de este tipo asignaremos True si es nominal y False si es entero o real. La clase la vamos a considerar siempre como un valor nominal (object en Python). Si algún dato no corresponde a uno de estos tres tipos, se deberá informar del error. Por ejemplo, se puede lanzar una excepción del tipo `ValueError`
- **`datos`**: Array bidimensional que se utilizará para almacenar los datos. Se recomienda que esta estructura sea un array Numpy, bien directamente, bien a través del uso de Data Frames de la librería Pandas (Intro to data structures — pandas 1.4.4 documentation (pydata.org)).
- **`diccionarios`**: Actúa como un conversor de datos nominales a datos numéricos, ya que la mayoría de los algoritmos de clasificación trabajan mejor con este tipo de dato. El diccionario contendrá para cada uno de los atributos del dataset, un conjunto de pares *clave-valor*, donde *clave* es el dato nominal original del dataset y *valor* es el valor numérico que asociamos a ese dato nominal. Si el atributo en el dataset original ya es un valor continuo, su conjunto de pares clave-valor estará vacío. En el caso de las variables nominales, el diccionario establecerá la relación entre los valores nominales o categóricos (claves) y un entero (valor). Para garantizar que este mapeo es consistente para diferentes ficheros y permutaciones de los datos, los enteros deben asignarse en orden lexicográfico de las claves. Así, por ejemplo, en el caso del conjunto de datos balloons, donde el primer atributo puede tomar los valores 'YELLOW', o 'PURPLE', el diccionario correspondiente deberá ser: `{'PURPLE': 0, 'YELLOW': 1}`. Una vez obtenidos los diccionarios de cada atributo, las variables categóricas (nominales) deberán ser codificadas a los valores enteros asignados en el diccionario para su posterior almacenamiento en el array `datos`. Como veremos a lo largo de las prácticas, el uso de estas matrices, con valores numéricos, facilita mucho la programación y acelera la ejecución de los algoritmos. En este sentido, la combinación entre la matriz

`datos` y el `diccionario` nos permite saber cuáles son los valores categóricos originales asociados a cada valor numérico y usar ese valor numérico para los cálculos matemáticos.

4. Particionamiento y Diseño Básico de la Aplicación

Introducción al particionamiento

La tarea de clasificación permite construir modelos a partir de un conjunto de datos. El modelo puede ayudar a resolver un problema pero es necesario evaluar su calidad antes de utilizarlo en un entorno real. Para ello es necesario un proceso de evaluación que nos ofrezca alguna medida objetiva de la precisión del modelo creado. Para la evaluación se podría utilizar el propio conjunto de entrenamiento como referencia, pero esta aproximación conduce normalmente a lo que se conoce como *sobreajuste*, es decir, la tendencia del modelo a trabajar bien con conjuntos de datos similares al de entrenamiento pero poco generalizable a otros datos. Lo más habitual por tanto es utilizar conjuntos diferentes para crear el modelo (*datos de entrenamiento*) y para evaluarlo (*datos de prueba o validación*).

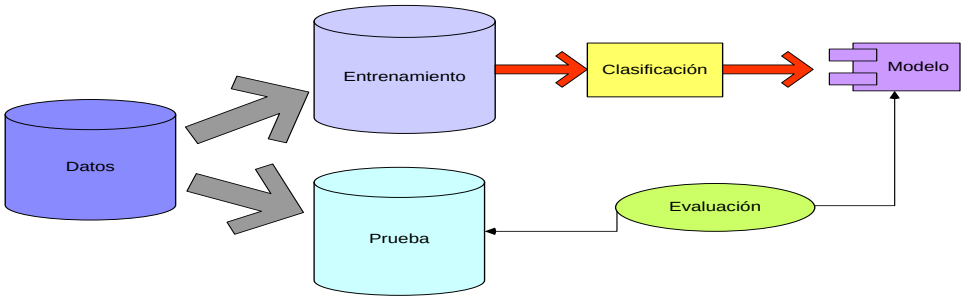

Figura 1: Evaluación de un modelo mediante conjuntos de entrenamiento y prueba [1]

Esta tarea se conoce como particionado de los datos y es previa al propio análisis. La utilización de dos conjuntos de datos distintos puede ayudar a resolver problemas como el sobreajuste, pero no está exenta de otro tipo de problemas, como la disponibilidad de pocos datos o la dependencia del modo en que se realice la partición de los datos.

Una manera de evitar los problemas de dependencia consiste en utilizar la técnica de *validación cruzada [1] (cross-validation)*. Con este método los datos se dividen en **k** grupos (comúnmente conocidos como *folds*), uno de los cuales se reserva para el conjunto de prueba y los **k-1** restantes se usan como conjunto de entrenamiento para construir el modelo. El modelo se evalúa entonces para predecir el resultado sobre los datos de prueba reservados. Este proceso se repite **k** veces, dejando cada vez un grupo diferente como conjunto de pruebas y el resto como conjunto de entrenamiento. De esta manera se obtienen **k** tasas de error, a partir de las que puede obtenerse la tasa de error promedio y la desviación típica de los errores, que nos darán la estimación de la precisión del modelo (ver Figura 2).

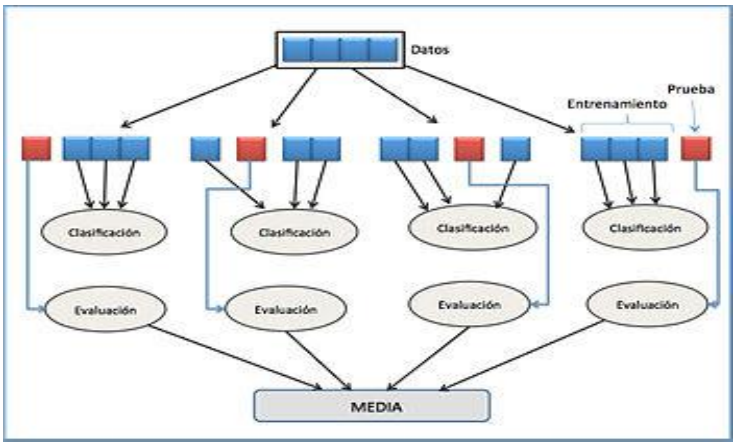

Figura 2: Estrategia de validación cruzada con cuatro grupos (folds). K=4. Fuente: https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada

Otra posibilidad es utilizar la estrategia de retención, *validación simple*, o hold-out donde se genera un solo conjunto de entrenamiento y un solo conjunto de prueba. Típicamente se especifica el tamaño deseado para una de estas dos particiones; por ejemplo, el tamaño del conjunto de entrenamiento (ver Figura 3). Cuando se usa esta técnica para evaluar métodos de aprendizaje automático, es común realizar varias particiones hold-out para diferentes permutaciones de los datos y, a partir de los errores obtenidos en cada permutación, calcular la media (y desviación típica).

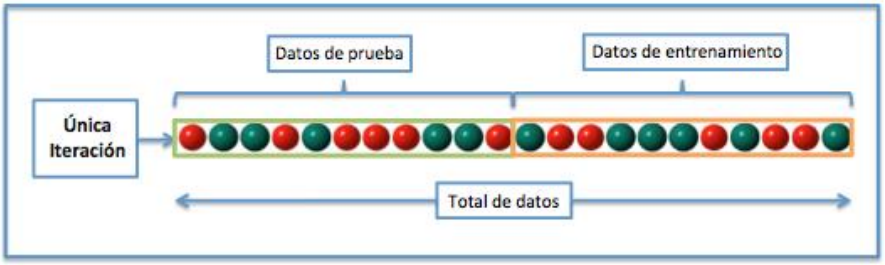

Figura 3: Ejemplo de estrategia hold-out con una sola partición de datos. Fuente: https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada

NOTA: Cuando se realiza el particionado es importante asegurarse de que las particiones son generadas sin ningún tipo de sesgo. Por ejemplo, si se tienen los patrones ordenados por clase y las particiones se generan sin permutar los datos, no se estaría obteniendo una muestra "realista" de la distribución subyacente en los datos.

Diseño de la Aplicación

Con el objetivo de desarrollar una aplicación lo más flexible y general posible se sugiere un diseño basado en patrones cuyos detalles más importantes se muestran en la Figura 4 y se especifican a continuación. Este diseño es una sugerencia y puede ser modificado si se considera necesario. Las modificaciones realizadas (diseño de nuevas clases y/o atributos) deberán ser descritas en la entrega de la práctica. No obstante, se penalizarán diseños que incurran en problemas ya estudiados en asignaturas previas: redundancia, falta de adaptación a cambios, etc.

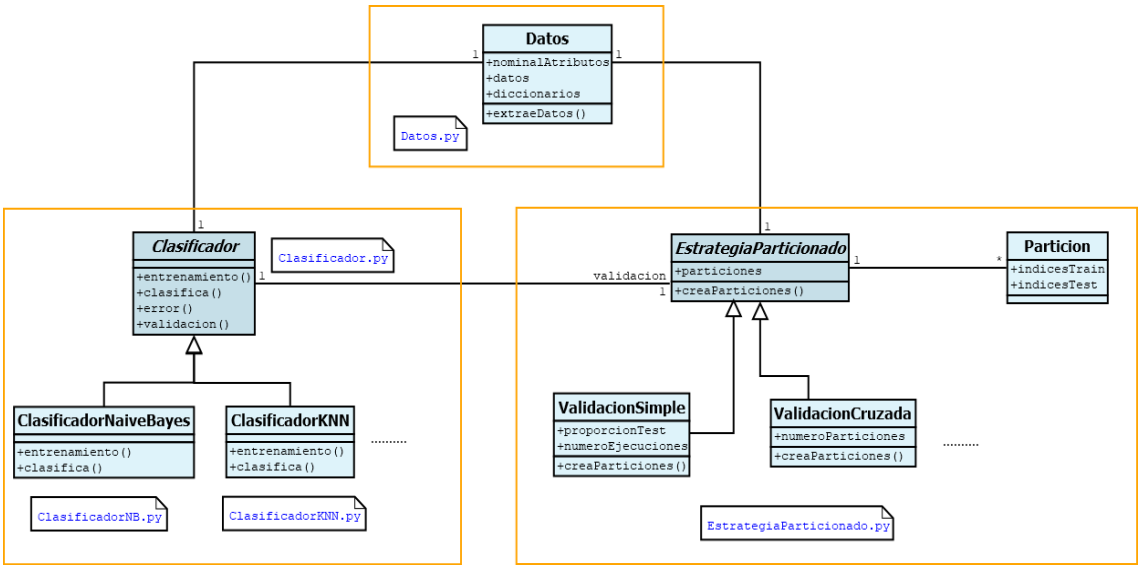

Figura 4: Esquema de clases y ficheros sugerido

PARTICIONADO

Para conseguir un particionado de los datos con la mayor flexibilidad posible se debe crear una clase abstracta *EstrategiaParticionado,* que separe la estrategia de particionado del particionado en sí mismo. Por su parte, la clase abstracta *Clasificador* utiliza una estrategia de particionado determinada para llevar a cabo su algoritmo de clasificación, empleando el conjunto de entrenamiento para construir el modelo y el conjunto de prueba para evaluarlo. Cada estrategia de particionado creará los conjuntos de entrenamiento y prueba de diferente forma según el algoritmo de particionado. Con el esquema propuesto es sencillo ir añadiendo diferentes estrategias de particionado en función de las necesidades de cada clasificador. La clase *Partición* contiene los conjuntos de índices para entrenamiento y prueba que se generan al utilizar cada tipo de particionado. **Es importante tener en cuenta que estos índices son simplemente listas de números que nos permiten acceder a las filas correspondientes de la matriz *datos* que se calculó en la primera parte de esta práctica**.

CLASIFICADORES

Como el objetivo de la asignatura es implementar y trabajar con diferentes clasificadores, la idea es crear una jerarquía de clasificadores, como se propone en la Figura 4. Los métodos **entrenamiento** y **clasifica** son auto-explicativos y cada uno utilizará el conjunto de índices de entrenamiento y prueba correspondiente, creados según el tipo de particionado. El método **error** calcula el número de errores comparando la clase predicha por el clasificador con el valor real de la clase, para cada instancia del conjunto de prueba. Finalmente, el método **validación** funciona como el método principal, calculando las particiones e iterando sobre el número de particiones calculadas para entrenar y clasificar sobre cada partición, obteniendo el error correspondiente a cada partición.

DATOS

La clase de datos se refiere a la comentada en la primera parte de la práctica, y como se indicó, su función principal es procesar los datos originales para construir las estructuras más adecuadas para la clasificación.

PLANTILLAS

En Moodle se encuentran los ficheros ***Datos.py***, ***Clasificador.py*** y ***EstrategiaParticionado.py*** con las plantillas para las distintas clases presentadas en la Figura 4. En esta práctica solo es necesario completar Datos.py y EstrategiaParticionado.py. Se incluye también un Notebook ***MainPracticaIntroFAA.ipynb*** para probar estas dos clases.

5. Actividades

La planificación temporal sugerida y las actividades a llevar a cabo son las siguientes:

- *1º semana*: Implementar la clase Datos para tratar el fichero original y construir las estructuras que contendrán los datos apropiados para su tratamiento posterior.

- *2ª semana*: Implementar el diseño de la aplicación necesaria para organizar los clasificadores y crear las particiones para los conjuntos de entrenamiento y prueba de un clasificador. Se deben implementar los dos métodos de particionado explicados, validación cruzada y validación simple, pero el diseño debería permitir añadir cualquier otro tipo de particionado de forma flexible.

6. Fecha de entrega y entregables

3 de Octubre de 2024. La entrega debe realizarse antes del comienzo de la clase de prácticas correspondiente. Se deberá entregar un fichero comprimido .zip con nombre **`FAAINTRO_<grupo>_<pareja>.zip`** (ejemplo `FAAINTRO_1461_1.zip`) y el siguiente contenido:

1. Código Python con la implementación de las clases *Datos* (**Datos.py**) y *EstrategiaParticionado* (**EstrategiaParticionado.py**)

Esta práctica se evalúa como APTA/NO APTA