Fundamentos de Aprendizaje Automático 2016/2017

PRÁCTICA Nº 2

1. Objetivo

En esta práctica continuaremos el estudio de los clasificadores implementando dos nuevos algoritmos: los *vecinos próximos* y la *regresión logística*. Paralelamente a la implementación propia, se deben comparar sus resultados con los que proporciona la librería de aprendizaje automático *scikit-learn* (http://scikit-learn.org/stable/).

2. Tareas

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

- *1º semana*: Utilizando la librería de aprendizaje automático scikit-learn, comparar los resultados de la clasificación Naive-Bayes obtenidos por la implementación propia, con los ofrecidos por la herramienta. Esta tarea formaba ya parte de la práctica 1 pero será evaluada en esta práctica.

- *1ª y 2ª semana:* Implementar el algoritmo de *vecinos próximos* (clase **ClasificadorVecinosProximos**) para realizar una tarea de clasificación de los siguientes conjuntos de datos, probando con diferentes valores de vecindad (K=1, 3, 5, 11, 21 y 51). Ejecutar también estos conjuntos de datos con la librería de scikit-learn para vecinos próximos (ver apartado 3).

  - ✓ Conjunto de datos ***example3.data*** para clasificar datos bidimensionales. Se proporciona en Moodle este conjunto de datos. Estos datos y las fronteras de decisión obtenidas con el clasificador pueden representarse utilizando la función descrita en el apartado 4.
  - ✓ Conjunto de datos ***example4.data*** para clasificar datos bidimensionales. Se proporciona en Moodle este conjunto de datos. Estos datos y las fronteras de decisión obtenidas con el clasificador pueden representarse utilizando la función descrita en el apartado 4.
  - ✓ Conjunto de datos ***wine*** (http://archive.ics.uci.edu/ml/datasets/Wine) para determinar el tipo de vino en base a propiedades químicas. Este conjunto de datos ya se proporcionó en la práctica 1.
  - ✓ Conjunto de datos ***wdbc*** (http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/). Se proporciona en Moodle una versión de este conjunto.
  - ✓ Conjunto de datos de ***dígitos*** obtenido en las clases de teoría. Una versión procesada de los datos (`digits.data`) ya se proporcionó en la práctica 1.

  En el uso del algoritmo de vecinos próximos es recomendable normalizar los atributos de forma que cada uno de ellos tenga media 0 y desviación típica 1.

En el caso de los vecinos próximos, la normalización es especialmente importante puesto que evita problemas de escala. Por ejemplo, si un atributo está inicialmente representado en centímetros y posteriormente se representa en milímetros, la contribución de ese atributo al calcular la distancia euclídea entre puntos será mayor simplemente por el cambio de unidades. Normalizar los atributos evita este tipo de efectos. Para llevar a cabo la normalización, se añadirán nuevos métodos a la clase Datos tal y como se describe en el apartado 5.

- *3ª semana*: Implementar el algoritmo *regresión logística* (clase **ClasificadorRegresionLogistica**), aplicando el algoritmo de maximización de la verosimilitud visto en las clases de teoría. Comparar los resultados obtenidos para los conjuntos de datos ***example3***, ***example4***, y ***wdbc*** con el algoritmo de *regresión logística* y con el de *vecinos próximos*. Ejecutar este conjunto de datos con la regresión logística de la librería de scikit-learn (ver apartado 3).

  El algoritmo de regresión logística visto en las clases de teoría resuelve problemas de clasificación binarios (dos clases). Sin embargo, los conjuntos de datos **wine** y **dígitos** son problemas de clasificación multiclase, con 3 y 10 clases, respectivamente. Para poder aplicar la regresión logística a estos conjuntos de datos, se aplicará la estrategia de *uno contra el resto*, comúnmente conocida como ***one-versus-all*** o ***one-versus-rest***, que permite resolver problemas multiclase a partir de clasificadores binarios. Esta estrategia es la misma que utiliza por defecto la implementación de la regresión logística de scikit-learn para problemas multiclase.

  La estrategia *one-versus-all* se basa en la descomposición de un problema multiclase en varios problemas de clasificación binarios, de tal forma que se generan tantos problemas de clasificación binarios como número de clases tenga el problema de clasificación original. Cada problema de clasificación binario se define como el problema de clasificar una de las clases (clase positiva) frente al resto (clase negativa). Una vez entrenados cada uno de los clasificadores binarios, la clasificación se realiza evaluando el patrón de test en cada uno de los clasificadores binarios, que deben dar como resultado además de la predicción (clase positiva/negativa) un *score* de la confianza de su predicción, a mayor *score* más "seguro" está el clasificador en su predicción. Por tanto, para cada patrón de test se dispondrán de tantos scores como número de clases tenga el problema original y, la predicción final será aquella clase con un valor de confianza más alto. La Figura 1 muestra un ejemplo de aplicación de la estrategia ***one-versus-all***.

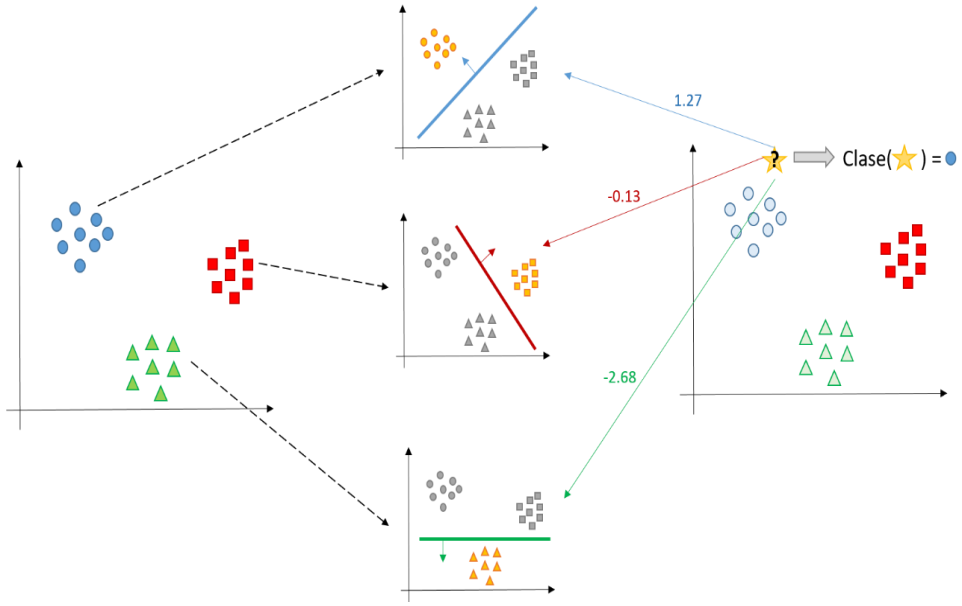


Figura 1 Esquema de funcionamiento de la estrategia uno contra el resto (one-versus-all) para problemas de clasificación multiclase

En esta práctica se proporciona el código de la clase **ClasificadorMulticlase** que, a partir de un algoritmo de clasificación binaria, realiza el entrenamiento y clasificación siguiendo la estrategia one-versus-all. Para poder aplicar este clasificador, es necesario, por tanto, implementar una función en la clase **ClasificadorRegresionLogistica** que para un conjunto de datos de validación devuelva su valor de confianza para cada uno de ellos. El valor de la confianza del clasificador de regresión logística se medirá como la distancia con signo del punto a evaluar al hiperplano separador. Es decir, dado el vector $\boldsymbol{w}$ calculado por el algoritmo de regresión logística, y dado el punto $\boldsymbol{x_0}$ del cual se quiere evaluar la confianza del clasificador, ésta se calculará como sigue

$$\frac{\boldsymbol{w} \cdot \boldsymbol{x_0}}{\|\boldsymbol{w}\|}$$

La función a implementar para la obtención de los scores se describe en el apartado 5. Con la nueva versión multiclase de la Regresión Logística se deben obtener resultados para los conjuntos ***wine*** y ***dígitos***, así como compararlos con los obtenidos por los vecinos próximos y la librería correspondiente a la regresión logística de scikit-learn.

3. Scikit-learn

Vecinos Próximos

Scikit-learn implementa dos clasificadores basados en vecindad. Para los propósitos de esta práctica interesa ***KNeighborsClassifier***. La versión básica de este algoritmo considera todos los vecinos por igual. Sin embargo, en algunas circunstancias es preferible ponderar el valor de los mismos y dar más importancia a los que estén más cercanos, dentro de la vecindad. El control de la opción básica o ponderada se establece con el parámetro weight. Si **weight='uniform'** se utiliza la opción básica (que corresponde también al valor por defecto) y si **weight='distance'** se asigna un peso proporcional a la inversa de la distancia. También se podría pasar una función propia para la distancia.

Regresión Logística

Scikit-learn proporciona una implementación de la *Regresión Logística* a través de la función ***LogisticRegression***.

4. Representación de fronteras de decisión para problemas de clasificación binarios con dos atributos

El fichero **plotModel.py** proporciona la implementación de la función `plotModel`:

```
plotModel(x,y,clase,clf,title)
```

`x`: valores del primer atributo para los ejemplos de entrenamiento
`y`: valores del segundo atributo para los ejemplos de entrenamiento
`clase`: etiquetas de los ejemplos de entrenamiento
`clf`: instanciación de un clasificador que implemente la clase abstracta Clasificador
`title`: string con el título de la gráfica

Por ejemplo, para el conjunto de datos example1.data de la práctica 1, se podría invocar a la función de la siguiente forma:

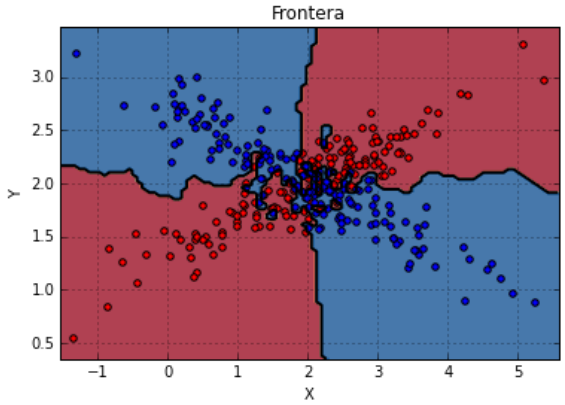
```
plotModel(dataset.datos[ii,0],dataset.datos[ii,1],dataset.datos
[ii,-1]!=0,clasificador, "Frontera")
```

Donde clasificador es una instanciación de la clase ClasificadorVecinosProximos:
`clasificador=Clasificador.ClasificadorVecinosProximos()`

`ii` son los índices de los patrones de entrenamiento de la última partición de la estrategia de particionado escogida:

```
ii=estrategia.particiones[-1].indicesTrain
```

Se obtendría la siguiente representación de los puntos y de entrenamiento y la frontera de decisión:



NOTA: la función plotModel invoca al método clasifica del clasificador sin proporcionar la etiqueta de los patrones en el parámetro datostest. Tener esto en cuenta a la hora de implementar la función clasifica de los métodos ClasificadorVecinosProximos y ClasificadorRegresionLogistica.

5. Lenguaje y Diseño

El lenguaje a utilizar para el desarrollo de la práctica será Python. Se utilizará la estructura de clases y métodos propuesta en las prácticas previas con alguna modificación, tal y como se describe en las siguientes secciones..

Normalización de datos

Para la normalización de los datos, se implementarán dos métodos nuevos en la clase Datos:

- `calcularMediasDesv(self,datostrain)`: esta función calculará las medias y desviaciones típicas de cada atributo continuo a partir de los datos de entrenamiento contenidos en la matriz `datostrain`.
- `normalizarDatos(self,datos)`: esta función normalizará cada uno de los atributos continuos en la matriz `datos` utilizando las medias y desviaciones típicas obtenidas en `calcularMediasDesv`.

Función de scores para verosimilitud

Para incorporar el posible uso de *scores* de predicción en la estructura de clases utilizada hasta el momento, se sugiere declarar los siguientes métodos en la clase abstracta `Clasificador`:

```
  def clasifica(self,datosTest,atributosDiscretos,diccionario):
    scores = self.score(datosTest,atributosDiscretos,diccionario)
    return np.argmax(scores,axis=1)


  # devuelve una matriz numpy  con el score para cada clase y dato
  def score(self,datosTest,atributosDiscretos,diccionario):
    scores = np.zeros((len(datosTest),len(diccionario[-1])))
    preds = map(lambda x: int(x),
self.clasifica(datosTest,atributosDiscretos,diccionario))
    scores[range(datosTest.shape[0]),preds] = 1.0
    return scores
```

Es decir, el método clasifica dejará de ser un método abstracto. Cada método de clasificación deberá sobreescribir al menos una de estas funciones para asegurar su correcto funcionamiento.
En particular, ClasificadorVecinosProximos implementará el método clasifica, mientras que ClasificadorRegresionLogista implementará la función **`score`**`(self,datostest,atributosDiscretos,diccionario)`, que deberá devolver un numpy array de dos dimensiones (matriz) con tantas filas como datos de test, cada una de ellas con el valor de la confianza en la predicción de cada una de las clases.

ClasificadorMulticlase

En el fichero **ClasificadorMulticlase.py** puede encontrarse una implementación de la estrategia *one-versus-all*. Esta clase tiene ya implementados los métodos entrenamiento y clasifica. No es necesario, por tanto, realizar ningún cambio en la función validación de la clase abstracta Clasificador. Este código debe incluirse en el fichero **Clasificador.py**.

La clase ClasificadorMulticlase es un *wrapper* de clasificadores binarios, siguiendo el esquema de la Figura 1. Por tanto, en el constructor de esta clase, se debe pasar como único parámetro una instanciación del clasificador binario a utilizar. En esta práctica, únicamente el algoritmo de regresión logística necesita hacer uso de este *wrapper*. La instanciación de la clase ClasificadorMulticlase se realizaría de la siguiente forma:

```
clasificadorbase=Clasificador.ClasificadorRegresionLogistica()
clasificador=Clasificador.ClasificadorMulticlase(clasificadorba
se)
```

6. Fecha de entrega y entregables

Semana del 7 al 11 de Noviembre de 2016. La entrega debe realizarse antes del comienzo de la clase de prácticas correspondiente. Los grupos de los miércoles deberán entregar la práctica antes del día 10 de noviembre a las 18h. Se deberá entregar un fichero comprimido .zip con nombre **`FAAP2_<grupo>_<pareja>.zip`** (ejemplo `FAAP2_1461_1.zip`) y el siguiente contenido:

1. **Ipython Notebook (.ipynb)** con las instrucciones necesarias para realizar las pruebas descritas en el apartado 2 y el correspondiente análisis de resultados. El Notebook debe estructurarse para contener los siguientes apartados:

Apartado 1	Resultados de Naive-Bayes obtenidos por scikit-learn en los conjuntos de datos de la práctica 1 y análisis comparativo con los resultados obtenidos por la propia implementación en python.
Apartado 2	Resultados de la clasificación mediante vecinos próximos (implementación original) para diferentes valores de vecindad en los conjuntos de datos propuestos. Comparación con los resultados proporcionados por scikit-learn. Obtener los resultados tanto para datos normalizados como sin normalizar. Es importante realizar un análisis de los resultados del algoritmo de vecinos próximos en los conjuntos de datos example3.data y example4.data, tratando de justificar el rendimiento del algoritmo en base a las características de los datos (normalizados y sin normalizar), que deben ser representados utilizando la función descrita en el apartado 4.
Apartado 3	Resultados de la clasificación mediante regresión logística en el conjunto de datos propuesto. Comparación con los resultados proporcionados por scikit-learn. Es importante realizar un análisis de los resultados del algoritmo de regresión logística en los conjuntos de datos example3.data y example4.data, tratando de justificar el rendimiento del algoritmo en base a las características de los datos, que deben ser representados utilizando la función descrita en el apartado 4.

2. **Ipython Notebook exportado como html.**

3. Código Python (**ficheros .py**) necesario para la correcta ejecución del Notebook