PER (E.T.S. de Ingeniería Informática)
Curso 2021-2022

Práctica 0. Introducción a la tarea de clasificación MNIST

Jorge Civera Saiz, Carlos D. Martínez Hinarejos,
Javier Iranzo Sánchez y Albert Sanchis Navarro
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Trabajo previo a la sesión de prácticas

Para la realización de esta práctica se supone que se ha adquirido previamente experiencia en el lenguaje de programación *Python*, tanto en Sistemas Inteligentes (SIN) como en otras asignaturas cursadas. Se deberá haber leído de forma detallada la totalidad del boletín práctico para poder centrarse en profundidad en la parte que hay que desarrollar en el laboratorio, la cual durará dos sesiones.

2. Introducción

Varias de las técnicas empleadas dentro del área de Reconocimiento de Formas (RF) y Aprendizaje Automático (AA) emplean cálculos matriciales. Es el caso de técnicas básicas de reducción de dimensionalidad, como *Principal Component Analysis* (PCA) y *Linear Discriminant Analysis* (LDA), así como clasificadores basados en funciones discriminantes lineales (Perceptron), o en distribuciones de probabilidad como son la Bernoulli, la multinomial y la Gaussiana entre otras. Por tanto, la aplicación de una herramienta que implemente de forma sencilla estos cálculos matriciales puede ayudar a obtener más rápidamente los sistemas de RF y AA que hacen uso de estas funcionalidades.

El lenguaje de programación Python es utilizado ampliamente para el desarrollo de sistemas de RF y AA gracias a la disponibilidad de potentes librerías que lo facilitan. En las sesiones de prácticas de la asignatura utilizaremos principalmente la librería NumPy de Python por sus capacidades de cálculos matriciales.

3. Tarea de clasificación: MNIST

En esta sección vamos a introducir la tarea de clasificación MNIST que se utilizará como tarea de referencia para evaluar los distintos clasificadores que se estudian tanto en la asignatura de Percepción, como en la asignatura de Aprendizaje Automático del primer semestre del cuarto año.

3.1. Descripción

La base de datos MNIST[1] consiste en una colección de imágenes de dígitos manuscritos (10 clases) con unas dimensiones de 28 x 28 píxeles en escala de 256 niveles de grises. Las dígitos que aparecen en las imágenes han sido normalizados en tamaño (20 x 20 píxeles) y centrados. Esta base de datos es un subconjunto de una base de datos más grande disponible desde el *National Institute of Standards and Technology* (NIST). Ha sido particionada en 60.000 imágenes de entrenamiento y 10.000 de test, que corresponde con los siguientes cuatro ficheros:

- Imágenes de entrenamiento: 60000 x 784 (`train-images-idx3-ubyte.txt.gz`)
- Etiquetas de clase de entrenamiento: 60000 x 1 (`train-labels-idx1-ubyte.txt.gz`)
- Imágenes de test: 10000 x 784 (`t10k-images-idx3-ubyte.txt.gz`)
- Etiquetas de clase de test: 10000 x 1 (`t10k-labels-idx1-ubyte.txt.gz`)

Estos ficheros se pueden descargar ejecutando el script `00-preprocess.sh` disponible en PoliformaT.

En la Figura 1 se muestra una representación directa de un dígito 0 que se corresponde con la segunda fila de datos del fichero `train-images-idx3-ubyte.txt.gz` que ha sido

[1] http://yann.lecun.com/exdb/mnist

```
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  51 159 253 159  50   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  48 238 252 252 252 237   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0  54 227 253 252 239 233 252  57   6   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  10  60 224 252 253 252 202  84 252 253 122   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0 163 252 252 252 253 252 252  96 189 253 167   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0  51 238 253 253 190 114 253 228  47  79 255 168   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0  48 238 252 252 179  12  75 121  21   0   0 253 243  50   0   0   0   0
  0   0   0   0   0   0   0   0  38 165 253 233 208  84   0   0   0   0   0   0 253 252 165   0   0   0   0   0
  0   0   0   0   0   0   0   7 178 252 240  71  19  28   0   0   0   0   0   0 253 252 195   0   0   0   0   0
  0   0   0   0   0   0   0  57 252 252  63   0   0   0   0   0   0   0   0   0 253 252 195   0   0   0   0   0
  0   0   0   0   0   0   0 198 253 190   0   0   0   0   0   0   0   0   0   0 255 253 196   0   0   0   0   0
  0   0   0   0   0   0  76 246 252 112   0   0   0   0   0   0   0   0   0   0 253 252 148   0   0   0   0   0
  0   0   0   0   0   0  85 252 230  25   0   0   0   0   0   0   0   0   7 135 253 186  12   0   0   0   0   0
  0   0   0   0   0   0  85 252 223   0   0   0   0   0   0   0   0   7 131 252 225  71   0   0   0   0   0   0
  0   0   0   0   0   0  85 252 145   0   0   0   0   0   0   0   0  48 165 252 173   0   0   0   0   0   0   0
  0   0   0   0   0   0  86 253 225   0   0   0   0   0   0 114 238 253 162   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0  85 252 249 146  48  29  85 178 225 253 223 167  56   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0  85 252 252 252 229 215 252 252 252 196 130   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0  28 199 252 252 253 252 252 233 145   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0  25 128 252 253 252 141  37   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figura 1: Representación *ascii* de un dígito 0 manuscrito de MNIST.

formateada a 28 filas y 28 columnas. Se puede apreciar como el tamaño del dígito está normalizado a 20 x 20 píxeles centrado sobre un fondo blanco.

La tarea MNIST ha sido cuidadosamente elaborada para que el conjunto de escritores de entrenamiento y test sea disjunto. De esta forma, no hay dígitos del mismo escritor en el entrenamiento y test. Asimismo, existen dos tipos de escritores que conviven en el conjunto de entrenamiento y en el test, que se corresponde con estudiantes de instituto y trabajadores de la oficina del censo, respectivamente. Estos últimos poseían una escritura más clara y fácil de reconocer.

En la web de la base de datos MNIST se proporcionan muchos más detalles sobre su elaboración. En esta misma página web se muestra una tabla de clasificadores (y preproceso aplicado) con la tasa de error conseguida sobre esta tarea y la referencia en forma de enlace a una descripción más detallada sobre el resultado conseguido por el investigador correspondiente.

MNIST es una base de datos adecuada para probar técnicas de aprendizaje automático y métodos de procesamiento de patrones en datos reales dedicando un esfuerzo mínimo al procesamiento de las imágenes y el formato.

3.2. Carga de datos

Los ficheros mencionados anteriormente son ficheros de texto *ascii* comprimidos. Sin embargo, para acelerar la carga de estos ficheros los convertiremos a ficheros binarios comprimidos utilizando la librería NumPy. Esta conversión está implementada en el script Python `01-nptxt2bin.py`, que está disponible PoliformaT, y genera ficheros con la extensión `npz`. Los ficheros resultantes pueden ser cargados fácilmente desde Python:

```
>>> import numpy as np
```

```
>>> X=np.load('train-images-idx3-ubyte.npz')['X'];
```

Comprobaremos que la variable X se ha cargado, así como sus dimensiones:

```
>>> np.shape(X)
(60000, 784)
```

El fichero de imágenes de test `t10k-images-idx3-ubyte.npz` se carga de la misma manera en la variable `Y`:

```
>>> Y=np.load('t10k-images-idx3-ubyte.npz')['Y'];
>>> np.shape(Y)
(10000, 784)
```

Dado que el objetivo es evaluar la tasa de error de clasificación disponemos de las etiquetas de clase de las imágenes, tanto de entrenamiento como de test. Estas etiquetas están en los ficheros `train-labels-idx1-ubyte.npz` y `t10k-labels-idx1-ubyte.npz`, respectivamente. Cargad estos ficheros para comprobar las dimensiones de las variables.

3.3. Visualización de dígitos

Podemos visualizar la imagen de la Fig. 1 que se corresponde con la segunda fila de la variable `X` teniendo en cuenta que es una imagen de 28 x 28 píxeles:

```
>>> import matplotlib.pyplot as plt
>>> x=np.reshape(X[1,:],(28,28));
>>> plt.imshow(x,cmap='gray_r');
>>> plt.axis('off'); plt.show();
```

También podemos visualizar las 20 primeras imágenes de la base de datos MNIST para hacernos una idea de la dificultad de esta tarea real (cierra la ventana de la imagen para ver la siguiente):

```
>>> for n in range(20):
...   x=np.reshape(X[n,:],(28,28));
...   plt.imshow(x,cmap='gray_r');
...   plt.axis('off'); plt.show();
```

4. Revisitando Python y NumPy

4.1. Funciones básicas en vectores y matrices

Como ya habéis estudiado en las prácticas de la asignatura de SIN, la librería NumPy[2] de Python aporta múltiples funciones para operar con vectores y matrices. En esta sección se revisan algunas de ellas que serán útiles para el proyecto de esta asignatura.

[2] https://numpy.org/doc/stable/reference/

La función más habitual para obtener las dimensiones de un vector o una matriz es la función `shape`. Se puede obtener el número de filas y columnas de una matriz en una única instrucción:

```
>>> rows,cols = np.shape(X)
```

Otra función de utilidad para estas prácticas es la función `unique` que nos permite obtener el conjunto de elementos diferentes de un vector eliminando duplicados. Por ejemplo, podemos aplicar la función `unique` al vector de etiquetas de clase de entrenamiento `xl`:

```
>>> labs = np.unique(xl)
>>> labs
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Por defecto, el vector resultante es un vector de número reales que conviene convertir a enteros al tratarse de etiquetas de clase:

```
>>> labs=labs.astype(int)
>>> labs
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Otra función muy útil para seleccionar un conjunto de filas o columnas en base a una condición es la función `where`. La función `where` devuelve los índices (posiciones) de aquellos elementos que cumplen una condición. Por ejemplo, se puede obtener los índices de las muestras que pertenecen a la clase del dígito 9 en el vector de etiquetas de clase de entrenamiento `xl`:

```
>>> ind9 = np.where(xl==9)
>>> ind9
(array([    4,    19,    22, ..., 59973, 59990, 59992]),)
```

Estos índices nos permiten extraer del conjunto de entrenamiento aquellas muestras (filas) que pertenecen a la clase del dígito 9:

```
>>> X[ind9]
```

Recuerda como en NumPy se puede indexar un subconjunto de filas (o columnas) de un vector o matriz mediante un vector.

En general, las funciones NumPy se aplican a la totalidad del vector o matriz, este es el caso de funciones como `sum` y `max`. Estas funciones aplicadas sobre un vector obtienen la suma y valor máximo del vector, pero cuando se aplican a matrices, se debe especificar como segundo parámetro el eje en que se realiza la operación, ya sea por columnas (`axis=0`) o por filas (`axis=1`). Por ejemplo:

```
>>> suma = np.sum(X[:,9:15],axis=0)
>>> suma
array([  0.,   0.,   0., 126., 470., 216.])
>>> np.max(X[:,9:15],axis=0)
array([  0.,   0.,   0., 116., 254., 216.])
```

En este caso aprovechamos este ejemplo para recordarte que NumPy permite seleccionar rangos de columnas (o filas) mediante el operador `:`.

Otra función de gran utilidad es la función `argmax` que, como su nombre indica, devuelve el índice o índices del valor máximo:

```
>>> np.argmax(X[:,9:15],axis=0)
array([    0,     0,     0, 12905, 12905, 12352])
```

Por otra parte, la función `sort` devuelve el vector ordenado de menor a mayor:

```
>>> np.sort(suma)
array([  0.,   0.,   0., 126., 216., 470.])
```

mientras que la función `argsort` devuelve el vector de índices de ordenación:

```
>>> i = np.argsort(suma)
>>> i
array([0, 1, 2, 3, 5, 4])
```

Si aplicamos el vector de ordenación `i` al vector `suma` obtenemos el vector ordenado:

```
>>> suma[i]
array([  0.,   0.,   0., 126., 216., 470.])
```

La librería NumPy contiene otras funciones que pueden resultar de utilidad como son la función `diag` que extrae la diagonal de una matriz o genera a partir de un vector una matriz diagonal, y las funciones `eye`, `ones` y `zeros` que devuelven la matriz identidad, todo unos y todo ceros, respectivamente.

Adicionalmente, la librería NumPy proporciona un conjunto de funciones de algebra lineal `numpy.linalg`[3] para operar con matrices, como son `det` para calcular el determinante de una matriz, `inv` para calcular la inversa de una matriz, y `eig` que calcula valores y vectores propios. Estas funciones también se pueden encontrar en la librería `scipy.linalg`[4] de Python que ofrece una funcionalidad ligeramente diferente o adicional. Esta funcionalidad adicional se puede observar en la función `eig` que permite la resolución de valores propios generalizados, utilizados en técnicas de reducción de dimensionalidad como LDA.

[3] https://numpy.org/doc/stable/reference/routines.linalg.html

[4] https://docs.scipy.org/doc/scipy/reference/reference/linalg.html#module-scipy.linalg

4.2. Operadores de iteración y condicionales

Como es de esperar de cualquier lenguaje de programación, Python dispone de operadores de iteración como `for` y `while`, así como condicionales `if` y `match-case`. Sin embargo, el operador `for` permite iterar sobre los valores de un vector (o matriz):

```
>>> for c in labs:
...   print("c = %d" % c);
...
c = 0
c = 1
c = 2
c = 3
c = 4
c = 5
c = 6
c = 7
c = 8
c = 9
```

Ejercicio 4.2.1 Utilizando las funciones que se han descrito, calcula el vector media de cada dígito y almacénalo en las columnas de la matriz `medias`.

Ejercicio 4.2.2 Realiza el mismo cálculo que en el ejercicio anterior pero haciendo uso de la función `mean` de NumpPy.

4.3. Funciones

Al igual que en otros lenguajes de programación, en Python se pueden definir funciones de usuario. Por ejemplo, una implementación de la función `mean` en el fichero `media.py`:

```
import numpy as np

def mean(X):
  m = np.sum(X,axis=0)/np.shape(X)[0];
  return m;
```

Para poder utilizar esta función en otro fichero debes importarla previamente:

```
from media import mean
```

4.4. Scripts

Python se puede usar de forma **no interactiva** escribiendo *scripts* (también llamados programas) que son interpretados y donde se pueden emplear las mismas instrucciones que en el modo interactivo. Por ejemplo, suponiendo que tenemos en el directorio actual el fichero `media.py`, podemos definir el siguiente script `script.py` que carga los datos de entrenamiento e invoca a la función `mean` por cada clase, para finalmente representar gráficamente la media de cada clase:

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from media import mean

if len(sys.argv)!=3:
  print('Usage: %s <trdata> <trlabels>' % sys.argv[0]);
  sys.exit(1);

X= np.load(sys.argv[1])['X'];
xl=np.load(sys.argv[2])['xl'];

labs = np.unique(xl).astype(int);
m = np.zeros((labs.shape[0],X.shape[1]));

for c in labs:
  Xc = X[np.where(xl==c)];
  m[c] = mean(Xc);
  # m[c]=np.mean(Xc,axis=0);

for c in labs:
  x=np.reshape(m[c],(28,28));
  plt.imshow(x,cmap='gray_r');
  plt.axis('off'); plt.show();
```

Como se puede observar, los argumentos del script están disponibles en `sys.argv`. Estos argumentos están en formato de cadena, y pueden convertirse a formato numérico, si fuera necesario, empleando la función `int` para valores escalares o `fromstring` de NumPy para vectores..

Estos scripts se pueden ejecutar desde el intérprete de comandos de `bash`:

```
python script.py train-images-idx3-ubyte.npz train-labels-idx1-ubyte.npz
```

5. Ejercicio: optimización de un clasificador lineal

Los siguiente ejercicios están pensados para poner en práctica técnicas que se utilizarán en el proyecto de la asignatura. Para estos ejercicios, se retoma la práctica del bloque 2 de SIN para plantear una optimización del código proporcionado en su momento.

5.1. Ejercicio: optimizando un clasificador lineal

En PoliformaT se encuentra el fichero `SINLab2.tgz` para este ejercicio. En la práctica del bloque 2 de SIN se estudia una implementación en Python del algoritmo Perceptron disponible en PoliformaT en el fichero `perceptron.py`.

Este algoritmo estima un conjunto de pesos que minimiza el error de clasificación en el conjunto de entrenamiento. Asimismo, hay dos parámetros que controlan el comportamiento del algoritmo: el factor de aprendizaje α y el margen b.

También se disponía de una implementación del clasificador para funciones discriminantes lineales en la función Python `linmach.py`:

```
import math
import numpy as np

def linmach(w,x):
  C = w.shape[1]; cstar=1; max=-math.inf;
  for c in range(C):
    g=np.dot(w[:,c],x);
    if g>max:
      max=g; cstar=c;
  return cstar;
```

Esta función dada una matriz de pesos `w`, donde los pesos de cada clase están dispuestos por columnas, y una muestra de test `x`, devuelve la etiqueta de clase `cstar` en la que se clasifica la muestra `x`.

Ejercicio 5.1 Implementa una versión *matricial* de la función `linmach.py` que, en lugar de recibir una única muestra de test, reciba un conjunto de muestras de test dispuestas por filas en una matriz `X`, y devuelva un vector de etiquetas de clase `cstar`, donde cada elemento (fila) es la clasificación de una muestra de test. Para comprobar tu versión matricial de la función `linmach.py` deberás modificar adecuadamente el script Python `experiment.py`. Primeramente, entrena los vectores de pesos mediante Perceptron en un pequeño subconjunto (1 %) del conjunto de entrenamiento de MNIST, y después compara el error de clasificación entre la versión de SIN y la versión matricial que has implementado.

Nota: No debes utilizar operadores iterativos (`for`, o `while`) para la implementación de la versión matricial de `linmach.py`, sino el producto matricial y la función `argmax` de NumPy.

Ejercicio 5.2 La función `confus` proporciona una estimación del error empírico y de la matriz de confusión al comparar la etiqueta de clase real `te(:,L)` con la etiqueta de clase estimada `rl` por el clasificador. Reemplaza la llamada a `confus` por tu propia estimación del error empírico, por simplicidad no calcules la matriz de confusión. Comprueba su correcto funcionamiento como has hecho en el anterior ejercicio.

Nota: No se deben utilizar operadores iterativos (`for` o `while`) para su implementación, sino las funciones NumPy: `not_equal`, `sum`, `mean`, etc.

Ejercicio 5.3 Como recordarás, el algoritmo Perceptron dispone principalmente de dos parámetros que pueden ser ajustados para minimizar el número de errores de clasificación. Estos parámetros son α, el factor de aprendizaje que controla la magnitud en que los pesos se modifican tras cada error de clasificación en el conjunto de entrenamiento, y b, el margen por el cual la función discriminante de la clase correcta debe superar al resto de clases.

En la asignatura de SIN, el ajuste de estos parámetros se realizaba a la vista del error de clasificación en el conjunto de test. Sin embargo, lo habitual es dedicar un pequeño subconjunto extraído del conjunto de entrenamiento para el ajuste de parámetros. Dicho subconjunto se conoce como conjunto de validación o *development*. Una vez realizada la exploración de los valores de los parámetros en el conjunto de validación, los parámetros que minimizan el error de clasificación en el conjunto de validación se utilizan para estimar un clasificador con todos los datos de entrenamiento y calcular el error de clasificación en el conjunto de test, que es la cifra de error que se proporciona como tasa de error en la tarea MNIST.

Haz una copia de tu script `experiment.py` como `evaluation.py`, y modifica tu script `experiment.py` para que no necesite el conjunto de test, pero en su lugar utilice un porcentaje del conjunto de entrenamiento como conjunto de validación:

```
import sys
import math
import numpy as np
from perceptron import perceptron
from linmach import linmach

if len(sys.argv)!=8:
  print('Usage: %s <trdata> <trlabels> <alphas> <bs> <maxK> <%%tr> <%%dv>' % sys.argv[0]
  sys.exit(1);

X= np.load(sys.argv[1])['X'];
xl=np.load(sys.argv[2])['xl'];
alphas=np.fromstring(sys.argv[3],sep=' ');
bs=np.fromstring(sys.argv[4],sep=' ');
K=int(sys.argv[5]);
trper=int(sys.argv[6]);
dvper=int(sys.argv[7]);
```

```
[...]
```

Por simplicidad, selecciona la parte final del conjunto de entrenamiento como conjunto de validación.

Concretamente, realiza un experimento que utilice del conjunto de entrenamiento un 5 % tanto para entrenamiento como para validación y explore diversos valores de $\alpha = [0.1\ 0.01\ 0.001]$ y $b = [1.0\ 10.0\ 100.0]$. A la vista de los resultados entrena un clasificador con los parámetros que proporcionan los mejores resultados y evalúalo sobre el conjunto de test. Compara los resultados obtenidos con los resultados reportados en la tarea MNIST para el clasificador *linear classifier (1-layer NN)*.

Ejercicio 5.4 La función `perceptron.py` devuelve el vector de pesos estimado para cada clase, la iteración `k` en la que se detuvo el proceso de entrenamiento y el número de errores de clasificación en el conjunto de entrenamiento en esa iteración k.

Un estudio empírico muy interesante consiste en analizar la evolución del número de errores de clasificación tanto en el conjunto de entrenamiento como en el conjunto de validación tras cada iteración durante el proceso de entrenamiento. Para ello es necesario modificar la función `perceptron.py` de forma que su declaración sea:

```
def perceptron(tr,dv,b=0.1,a=1.0,K=200)
```

donde `tr` y `dv` son los conjuntos de entrenamiento y validación, respectivamente, y se devuelven vectores `Etr` y `Edv` que almacenan en cada iteración el número de errores de clasificación en el conjunto de entrenamiento y validación, respectivamente. Realiza un experimento con $\alpha = 0.001$, $b = 10.0$ y $K = 200$.

La representación gráfica del porcentaje de error de clasificación en el conjunto de entrenamiento (5 %) y validación (5 %) en función de la iteración te permitirá observar la convergencia del algoritmo visualmente. Para ello puedes utilizar la librería Matplotlib, más concretamente su interfaz `pyplot`[5]:

```
import matplotlib.pyplot as plt
```

que, tras invocar a la función `perceptron`, realiza la representación gráfica con las siguientes instrucciones:

```
plt.plot(np.arange(0,k+1),Etr/Ntr*100,'--xk',label='Train');
plt.plot(np.arange(0,k+1),Edv/Ndv*100,'-+k',label='Dev');
plt.xlabel('Iteration'); plt.ylabel('Error');
plt.xticks(np.arange(0,k,10));
plt.legend();
fn='perceptron.a%.1e.b%.1e.K%d.tr%d.dv%d.eps' % (a,b,K,trper,dvper);
plt.savefig(fn, format='eps')
```

[5] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html

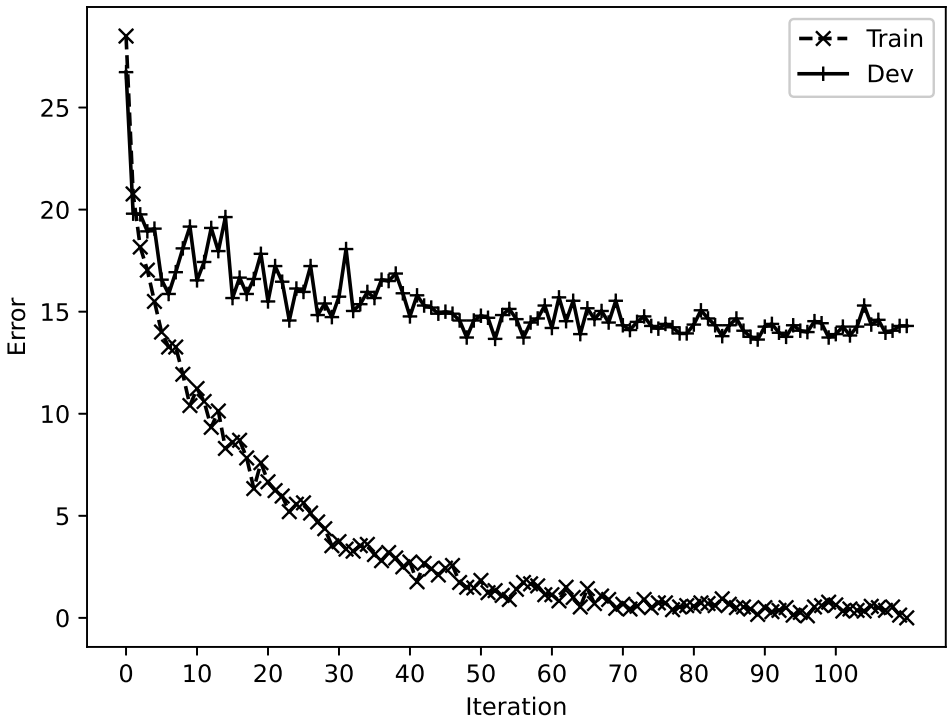

Figura 2: Evolución del porcentaje de error de clasificación (eje y) en los conjuntos de entrenamiento (abajo) y validación (arriba) en función del número de iteraciones (eje x) del algoritmo Perceptron hasta convergencia.

siendo `Ntr` y `Ndv`, el número de muestras en el conjunto de entrenamiento y validación, respectivamente. La representación gráfica que deberías obtener se muestra en la Figura 2.