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

Proyecto de prácticas. Reconocimiento de dígitos manuscritos: MNIST

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

Última actualización: 10/05/2022- 14:00:01

Índice

1. Objetivos

El principal objetivo de este proyecto de prácticas es la implementación y evaluación de diversos clasificadores estudiados en teoría sobre la tarea real de reconocimiento de dígitos manuscritos MNIST.

Este objetivo principal se subdivide en subobjetivos básicos que se pueden entender como fases o hitos de este proyecto:

1. Implementar la técnica de reducción de dimensionalidad PCA.
2. Visualizar los vectores de proyección PCA.
3. Realizar proyecciones lineales mediante vectores de proyección PCA.
4. Evaluar el efecto que tiene PCA sobre la tasa de error del clasificador basado en vecinos más cercanos.
5. Implementar el clasificador multinomial y gaussiano.
6. Resolver los problemas prácticos que nos podemos encontrar durante el desarrollo del clasificador multinomial y gaussiano.
7. Evaluar los clasificadores multinomial y gaussiano en función de sus parámetros.

La evaluación del proyecto de la asignatura consta de hasta 2 puntos en ejercicios obligatorios, y hasta 1 punto en ejercicios opcionales que el alumnado podrá elegir libremente. En total, la nota de prácticas constituye 3 puntos de la nota final de la asignatura.

El presente boletín debe leerse en su integridad previamente a la asistencia a la práctica para tener una visión completa del proyecto. Para realizar el proyecto de esta práctica se asume que el alumno ha realizado hasta el ejercicio 5.3 de la Práctica 0.

2. Principal Component Analysis

Para la implementación de la técnica de reducción de dimensionalidad PCA es necesario la utilización de la función `eig`[1], que como comentamos en el boletín introductorio, se encuentra entre las funciones de algebra lineal `numpy.linalg`. Esta función calcula los valores y vectores propios de una matriz devolviendo números complejos cuya parte real nos quedaremos mediante la función `real`.

La función `eig` no garantiza que los vectores propios que devuelve estén ordenados de mayor a menor por valor propio asociado. Por ello, será necesaria la utilización de la función `argsort`[2], también presentada en el boletín introductorio. Tened en cuenta que la función `argsort` ordena de menor a mayor y será necesario revertir el orden.

[1] https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html
[2] https://numpy.org/doc/stable/reference/generated/numpy.argsort.html

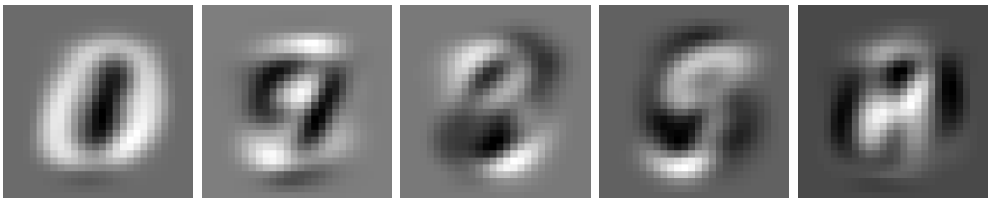
Ejercicio 2.1 Implementa la función `pca` que recibe como entrada los datos de entrenamiento `X` dispuestos por filas y devuelve el vector media `m` y la matriz de proyección `W` donde los vectores propios están dispuestos por columnas de mayor a menor valor propio asociado:

```
def pca(X):
  #
  # HERE YOUR CODE
  #
  return m,W;
```

Recuerda que esta función puede guardarse en un fichero `pca.py`. Los pasos a seguir para implementar la función `pca` son:

1. Calcula la media `m` de los datos de entrenamiento `X`.
2. Resta la media a todos los datos de entrenamiento para obtener la matriz `Xm`.
3. Calcula la matriz de covarianza de dimensiones $D \times D$ como el producto de la transpuesta de la matriz `Xm'` por la matriz `Xm`.
4. Calcula los valores y vectores propios de la matriz de covarianzas resultante utilizando la función `eig`.
5. Ordena los vectores propios de mayor a menor valor propio asociado mediante la función `argsort` para así definir la matrix `W`.

Ejercicio 2.2 Comprueba la correcta implementación de PCA representando gráficamente los 5 primeros eigenvectores. Recuerda que la visualización de vectores fue descrita en la Práctica 0. La representación gráfica de los 5 primeros vectores propios del conjunto de entrenamiento de MNIST es la siguiente:



3. Clasificador basado en vecinos más cercanos

En esta sección vamos a estudiar una implementación básica de un clasificador basado en vecinos más cercanos. También estudiaremos el efecto que tiene la técnica de reducción de dimensionalidad PCA sobre la tasa de error de un clasificador basado en vecinos más cercanos.

Para ello se dispone en PoliformaT de dos ficheros necesarios para el uso de este clasificador: `knn.py` y `L2dist.py`. Estos ficheros deben descargarse de PoliformaT.

El fichero `knn.py` contiene la función:

```
def knn(X,xl,Y,yl,k):
  ...
  return err;
```

que implementa un clasificador por `k` vecinos más cercanos. Más concretamente, esta función devuelve el error de clasificación de este clasificador usando `X` como muestras de entrenamiento (por filas) e `Y` como muestras de test (por filas). Las etiquetas de clase de entrenamiento y test almacenadas en `xl` y `yl`, respectivamente, se proporcionan tal cual se cargan de fichero, es decir, como un vector.

El parámetro `k` indica que para la clasificación de cada muestra de test se considerarán las `k` muestras de entrenamiento más cercanas a esta muestra de test. De forma que la clase mayoritaria entre las `k` muestras de entrenamiento más cercanas define la clase de la muestra de test.

El fichero `L2dist.py` incluye una función que implementa el cálculo de distancia L2 (Euclídea):

```
def L2dist(X,Y):
  ...
  return D;
```

Esta función devuelve la matriz `D` de distancia L2 entre las N muestras de `X` y las M muestras de `Y`. La función `L2dist(X,Y)` se invoca desde la función `knn(X,xl,Y,yl,k)`.

Ejercicio obligatorio (0.5 puntos). Estudia el comportamiento del clasificador basado en el vecino más cercano (`k`=1) en combinación con PCA con el objetivo de obtener una gráfica como la que aparece en la Figura 1.

Para ello, te proporcionamos la primera parte del script `pca+knn-exp.py` que a partir del conjunto de datos `trdata` y `trlabels` evalúa un rango de valores de dimensionalidad de PCA `ks` dedicando un porcentaje a entrenamiento y otro a validación:

```
 1 #!/usr/bin/python3
 2
 3 import sys
 4 import math
 5 import numpy as np
 6 from pca import pca
 7 from knn import knn
 8
 9 if len(sys.argv)!=6:
10   print('Usage: %s <trdata> <trlabels> <ks> <%%trper> <%%dvper>'
     % sys.argv[0]);
11   sys.exit(1);
12
13 X= np.load(sys.argv[1])['X'];
14 xl=np.load(sys.argv[2])['xl'];
```

```
15 ks=np.fromstring(sys.argv[3],dtype=int,sep=' ');
16 trper=int(sys.argv[4]);
17 dvper=int(sys.argv[5]);
18
19 N=X.shape[0];
20 np.random.seed(23); perm=np.random.permutation(N);
21 X=X[perm]; xl=xl[perm];
22
23 # Selecting a subset for train and dev sets
24 Ntr=round(trper/100*N);
25 Xtr=X[:Ntr]; xltr=xl[:Ntr];
26 Ndv=round(dvper/100*N);
27 Xdv=X[N-Ndv:]; xldv=xl[N-Ndv:];
...
```

Los resultados necesarios para realizar la Figura 1 se obtienen con la siguiente ejecución (desde el intérprete de comandos de `bash`):

```
./pca+knn-exp.py train-images-idx3-ubyte.npz train-labels-idx1-ubyte.npz \
"1 2 5 10 20 50 90" 90 10
```

Recuerda que la barra (\) no se debe escribir y únicamente indica que el comando sigue en la siguiente línea. Recuerda incluir en este script el código necesario para calcular la tasa de error del clasificador basado en el vecino más cercano sin aplicar PCA.

Las líneas 9-17 capturan los parámetros de entrada del script: `trdata` es el fichero con el conjunto de datos (muestras por filas); `trlabels`, fichero con el conjunto de etiquetas de clase; `ks`, vector con el rango de dimensiones PCA a evaluar; `trper`, porcentaje del conjunto de datos para entrenamiento; y `dvper` para validación. Las líneas 19-21 realizan el barajado por filas de los datos y etiquetas de clase. Las líneas 23-27 realizan la partición en entrenamiento y validación.

Completa el script `pca+knn-exp.py` con el código necesario para calcular PCA del conjunto de entrenamiento, realizar la proyección a k dimensiones de los conjuntos de entrenamiento y validación, y efectuar la llamada al clasificador basado en el vecino más cercano (`k`=1) con estos conjuntos para obtener la tasa de error en el conjunto de validación que se representa gráficamente en la Figura 1. Recuerda que también debes calcular la tasa de error con la dimensionalidad original de los datos.

A la vista de los resultados obtenidos en el ejercicio anterior, entrena un clasificador final que utilice el conjunto de datos de entrenamiento (incluyendo el conjunto de validación) con los mejores parámetros y calcule la tasa de error en el conjunto de test.

Para ello, te proporcionamos la primera parte del script `pca+knn-eva.py` que a partir del conjunto de datos de entrenamiento `trdata` y `trlabels`, el conjunto de test `tedata` y `telabels`, y el valor óptimo de la dimensionalidad de PCA `k`, debe calcular la tasa de error en el conjunto de test aplicando PCA y sin aplicar PCA.

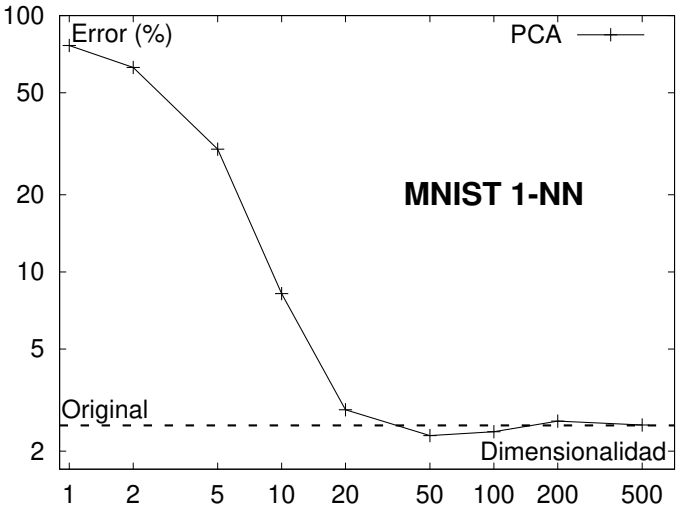


Figura 1: Resultados comparativos entre PCA y espacio original en MNIST utilizando el clasificador de 1 vecino más cercano (1-NN).

```
#!/usr/bin/python3

import sys
import math
import numpy as np
from pca import pca
from knn import knn

if len(sys.argv)!=6:
  print('Usage: %s <trdata> <trlabels> <tedata> <telabels> <k>' % sys.argv[0]);
  sys.exit(1);

X= np.load(sys.argv[1])['X'];
xl=np.load(sys.argv[2])['xl'];
Y= np.load(sys.argv[3])['Y'];
yl=np.load(sys.argv[4])['yl'];
k=int(sys.argv[5]);
...
```

Compara tus resultados con los aportados en la web de MNIST para *K-Nearest Neighbors*.

Ejercicios opcionales A continuación se plantean una serie de ejercicios opcionales con los cuales se puede sumar hasta 0.5 puntos a la nota de prácticas:

- Como se puede observar en la web de MNIST para *K-Nearest Neighbors* se proporcionan resultados con distancia *L3* que mejoran los resultados de la distancia *Euclidean (L2)* para la dimensionalidad original de los datos. Implementa otras distancias de la familia L_p:

$$d_p(\boldsymbol{a}, \boldsymbol{b}) = \left(\sum_{i=1}^{D} |a_i - b_i|^p \right)^{\frac{1}{p}}$$

  al menos *L0*, *L1* y *L3*, y estudia su comportamiento en un clasificador basado en el vecino más cercano (k=1) combinado con una reducción de dimensionalidad PCA (0.25 puntos).

- La distancia Euclídea ponderada suele proporcionar resultados que mejoran los conseguidos por la distancia Euclídea no ponderada. Más concretamente, la distancia Mahalanobis-diagonal por clase presenta un compromiso entre complejidad de implementación y buenos resultados. Implementa esta distancia y estudia su comportamiento en un clasificador basado en el vecino más cercano (k=1) combinado con una reducción de dimensionalidad PCA (0.5 puntos).

  Nota: Dado que para algunas dimensiones obtendrás varianzas igual a cero, será necesario que suavices dichas varianzas mediante flat smoothing con la varianza de la gaussiana estandarizada (ver Tema 5 de teoría).

- Implementa el algoritmo de edición de Wilson y estudia su comportamiento en un clasificador basado en el vecino más cercano (k=1) combinado con una reducción de dimensionalidad PCA (0.5 puntos).

4. Implementación de un clasificador multinomial

Antes de abordar la implementación del clasificador multinomial es necesario recordar que, tanto el clasificador multinomial como el clasificador gaussiano que se verá en la siguiente sección, son instanciaciones del clasificador de Bayes:

$$\begin{aligned} c^*(x) &= \operatorname*{argmax}_{c=1,\dots,C} P(c \mid x) \\ &= \operatorname*{argmax}_{c=1,\dots,C} P(c)\, p(x \mid c) \\ &= \operatorname*{argmax}_{c=1,\dots,C} \log P(c) + \log p(x \mid c) \end{aligned}$$

donde la f.d. condicional $p(x \mid c)$ se modeliza mediante una distribución concreta, ya sea multinomial, gaussiana, u otra cualquiera. En esta sección, la f.d. condicional $p(x \mid c)$ será una distribución multinomial D-dimensional:

$$
\begin{aligned}
c^*(\boldsymbol{x}) &= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log P(c) + \log p(x \mid c) \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log P(c) + \log \frac{x_+!}{x_1! \cdots x_D!} \prod_{d=1}^{D} p_{cd}^{x_d} \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log P(c) + \log \frac{x_+!}{x_1! \cdots x_D!} + \sum_{d=1}^{D} x_d \log p_{cd}
\end{aligned}
$$

Eliminando términos constantes y expresado en términos de función discriminante lineal:

$$
\begin{aligned}
c^*(\boldsymbol{x}) &= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ g_c(\boldsymbol{x}) \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \boldsymbol{w}_c \, \boldsymbol{x} + w_{c0}
\end{aligned}
$$

donde

$$
\begin{aligned}
\boldsymbol{w}_c &= \log \boldsymbol{p}_c \\
w_{c0} &= \log p(c)
\end{aligned}
$$

La estimación máximo-verosímil de los parámetros del clasificador multinomial es:

$$
\hat{p}(c) = \frac{N_c}{N} \qquad\qquad \hat{\boldsymbol{p}}_c = \frac{1}{\sum\limits_{n \,:\, c_n = c} \sum\limits_{d} x_{nd}} \sum_{n \,:\, c_n = c} \boldsymbol{x}_n
$$

En el caso de la tarea MNIST, $\boldsymbol{x}_n$ es el vector de niveles de gris de la imagen n y x_{nd}, el nivel de gris del píxel d de la imagen n.

Antes de realizar un experimento, es necesario suavizar los parámetros correspondientes a los prototipos multinomiales pues existen píxeles cuyo nivel de gris es siempre cero. En este caso se propone aplicar el *suavizado de Laplace* (ver Tema 5 de teoría) a nuestros prototipos multinomiales:

$$
\tilde{\boldsymbol{p}}_{cd} = \frac{\hat{\boldsymbol{p}}_{cd} + \epsilon}{\sum\limits_{d} (\hat{\boldsymbol{p}}_{cd} + \epsilon)} \qquad \epsilon > 0
$$

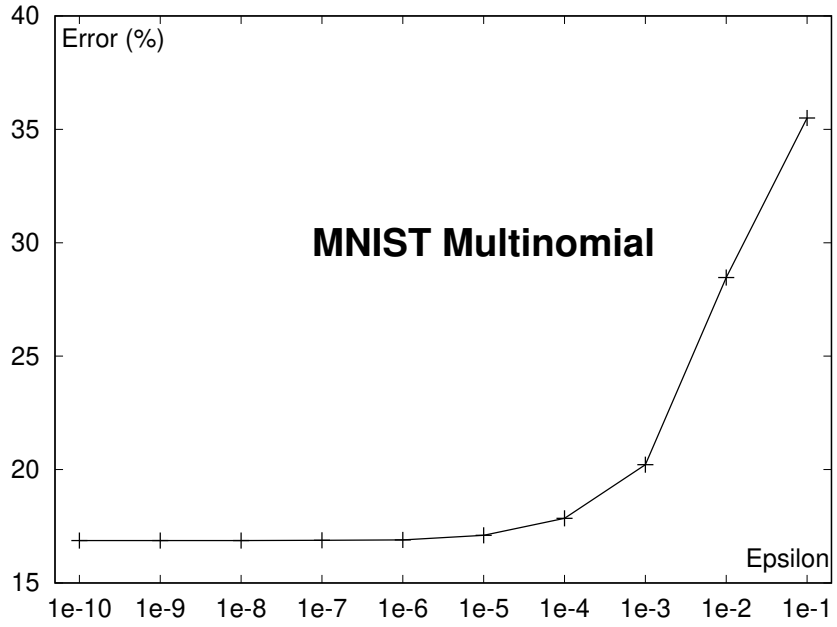


Figura 2: Error del clasificador multinomial en función del parámetro de suavizado ϵ.

Ejercicio obligatorio (0.5 puntos). Implementa la función `multinomial` en el fichero `multinomial.py`:

```
def multinomial(Xtr,xltr,Xdv,xldv,epsilons):
  # HERE YOUR CODE
  return edv;
```

que recibe como entrada los conjuntos de datos y etiquetas de clase de entrenamiento, `Xtr` y `xltr`, y los conjuntos de datos y etiquetas de clase de validación, `Xdv` y `xldv`, un vector de posibles valores ϵ de suavizado de Laplace; y devuelve la tasa de error de clasificación en el conjunto de validación `edv` para cada uno de los valores de suavizado del vector `epsilons`.

Realiza un experimento en MNIST para estudiar la evolución de la tasa de error en el conjunto de validación en función del valor ϵ utilizado con el objetivo de obtener una gráfica como la mostrada en la Figura 2.

Para ello debes implementar un script `multinomial-exp.py`, muy similar al script `pca+knn-exp.py` que invoca a la función `multinomial`. Dedica un 90 % de los datos de entrenamiento de MNIST al conjunto de entrenamiento y un 10 % al conjunto de validación.

A la vista de los resultados obtenidos en el conjunto de validación, entrena un clasificador final con los datos de entrenamiento (incluyendo el conjunto de validación) y el mejor valor de ϵ de suavizado. Estima el error en el conjunto de test con este clasificador final. Integra la realización de esta estimación en un script `multinomial-eva.py`, que

también será similar a `pca+knn-eva.py`. Compara el resultado obtenido con los reportados en la web de MNIST.

5. Implementación de un clasificador gaussiano

En esta sección, la f.d. condicional $p(x \mid c)$ del clasificador de Bayes será una distribución gaussiana D-dimensional:

$$p(\mathbf{x} \mid c) \sim \mathcal{N}_D(\boldsymbol{\mu}_c, \Sigma_c), \quad c = 1, \ldots, C$$

Por tanto:

$$\begin{aligned}
c^*(\boldsymbol{x}) &= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log\, P(c) + \log\, p(\boldsymbol{x} \mid c) \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log\, P(c) - \frac{D}{2}\log 2\pi - \frac{1}{2}\log\,|\Sigma_c| - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_c)^t\Sigma_c^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_c) \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log P(c) - \frac{1}{2}\log\,|\Sigma_c| - \frac{1}{2}\boldsymbol{x}^t\Sigma_c^{-1}\boldsymbol{x} + \boldsymbol{\mu}_c^t\Sigma_c^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_c^t\Sigma_c^{-1}\boldsymbol{\mu}_c \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \log P(c) - \frac{1}{2}\boldsymbol{x}^t\Sigma_c^{-1}\boldsymbol{x} + \boldsymbol{\mu}_c^t\Sigma_c^{-1}\boldsymbol{x} + \left(-\frac{1}{2}\log\,|\Sigma_c| - \frac{1}{2}\boldsymbol{\mu}_c^t\Sigma_c^{-1}\boldsymbol{\mu}_c\right)
\end{aligned}$$

Eliminando términos constantes y expresado como función discriminante cuadrática con $\boldsymbol{x}$:

$$\begin{aligned}
c^*(\boldsymbol{x}) &= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ g_c(\boldsymbol{x}) \\
&= \underset{c=1,\ldots,C}{\operatorname{argmax}} \ \boldsymbol{x}^t\, W_c\, \boldsymbol{x} + \boldsymbol{w}_c^t\, \boldsymbol{x} + w_{c0}
\end{aligned}$$

donde

$$\begin{aligned}
W_c &= -\frac{1}{2}\Sigma_c^{-1} \\
\boldsymbol{w}_c &= \Sigma_c^{-1}\boldsymbol{\mu}_c \\
w_{c0} &= \log P(c) - \frac{1}{2}\log\,|\Sigma_c| - \frac{1}{2}\boldsymbol{\mu}_c^t\Sigma_c^{-1}\boldsymbol{\mu}_c
\end{aligned}$$

La estimación máximo-verosímil de los parámetros del clasificador gaussiano es ampliamente conocida:

$$\hat{P}(c) = \frac{N_c}{N}$$
$$\hat{\boldsymbol{\mu}}_c = \frac{1}{N_c} \sum_{n:c_n=c} \boldsymbol{x}_n$$
$$\hat{\Sigma}_c = \frac{1}{N_c} \sum_{n\,:\,c_n=c} (\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_c)(\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_c)^t$$

El suavizado con la matriz identidad (*flat smoothing*) es:

$$\tilde{\Sigma}_c = \alpha \cdot \hat{\Sigma}_c + (1 - \alpha) \cdot I$$

Ejercicio obligatorio (1 punto). Implementa la función `gaussian` en el fichero `gaussian.py`:

```
def gaussian(Xtr,xltr,Xdv,xldv,alphas):
  # HERE YOUR CODE
  return edv;
```

que recibe como entrada los conjuntos de datos y etiquetas de clase de entrenamiento, `Xtr` y `xltr`, los conjuntos de datos y etiquetas de clase de validación, `Xdv` y `xldv`, un vector de posibles valores α para realizar el suavizado con la matriz identidad; y devuelve la tasa de error de clasificación en el conjunto de validación `edv` para cada uno de los valores de suavizado del vector `alphas`.

Algunos puntos a tener en cuenta de cara a la implementación:

- La implementación de la estimación de los parámetros de la gaussiana de cada clase, probabilidad a priori, media y matriz de covarianzas se deberá realizar seleccionando del conjunto de entrenamiento las muestras de cada clase.
- Las medias (una por cada clase) es recomendable almacenarlas en una matriz `mu`.
- Para almacenar las matrices de covarianzas es recomendable utilizar una matriz tridimensional donde `sigma[c]` es la matriz de covarianzas de la clase `c`.
- Es recomendable implementar una función auxiliar `pxc(pc[c],mu[c],sigma[c],X)` que calcule la probabilidad conjunta $p(\boldsymbol{x}, c)$ de cada muestra en `X` dados los parámetros de la clase c.

Si realizas un experimento (90 % entrenamiento, 10 % validación) sin suavizado ($\alpha = 1.0$), mediante el script `gaussian-exp.py` que debes implementar, observarás que Python imprime un mensaje de error "`numpy.linalg.LinAlgError: Singular matrix`".

Esto se debe a que las matrices de covarianzas estimadas a partir de los datos de cada clase son singulares, es decir, el rango de estas matrices no es completo ya que existen filas (o columnas) que son linealmente dependientes unas de otras. En el caso de

MNIST, esto se debe a que tenemos filas (o columnas) de la matriz de covarianzas que son todo cero, al ser estas filas (o columnas) parte del fondo de la imagen sobre el que está centrado el dígito manuscrito.

Como consecuencia de que las matrices de covarianzas sean singulares se plantean dos problemas de estimación de la probabilidad condicional gaussiana:

1. El determinante de un matriz singular es cero y su logaritmo será `-Inf`.
2. No es posible calcular la inversa de una matriz singular.

El primer problema lo resolveremos reemplazando el cero por el valor `float` más pequeño representable en nuestra máquina, que es la constante `sys.float_info.min` en Python. Adicionalmente, para que el cálculo del logaritmo del determinante sea más robusto utilizaremos la propiedad que dice que el determinante de una matriz es igual al producto de sus valores propios[3]. Por tanto, el logaritmo del determinante será la suma del logaritmo de los valores propios:

$$\log |\Sigma| = \sum_{d=1}^{D} \log \lambda_d$$

El segundo problema lo solventaremos reemplazando la inversa `inv` por la pseudoinversa `pinv` que sí que se puede calcular para una matriz singular.

Debes implementar las dos soluciones a la singularidad de las matrices de covarianzas. Es recomendable que implementes estas dos soluciones en una función `logdet(X)` que calcule el logaritmo del determinante de una matriz. Recuerda que si existe un valor propio que sea cero, no podrás calcular el logaritmo porque el determinante es cero y deberás tratarlo como hemos explicado anteriormente. Realiza un experimento con la misma partición que la del ejercicio anterior pero variando el valor de α de suavizado como se muestra en la Figura 3.

A la vista de los resultados obtenidos en el conjunto de validación, entrena un clasificador final con los datos de entrenamiento (incluyendo el conjunto de validación) y el mejor valor de α de suavizado. Estima el error en el conjunto de test con este clasificador final. Integra la realización de esta estimación en un script `gaussian-eva.py`. Compara el resultado obtenido con los reportados en la web de MNIST, especialmente en la sección *Non-Linear Classifiers*.

Ejercicio opcional (0.5 puntos). Se puede observar como el clasificador gaussiano mejora la tasa de error del clasificador multinomial, pero según la web de MNIST queda lejos de la tasa de error conseguida con *40 PCA + quadratic classifier*. En nuestro caso, el clasificador cuadrático será un clasificador gaussiano. Realiza un experimento para evaluar el error del clasificador gaussiano con diferentes valores de α en función del número de componentes PCA a las cuales se proyectan los datos originales. Representa

[3] https://www.studocu.com/en-au/document/the-university-of-adelaide/mathematics-ia/evalue-magic-tricks-handout/832365

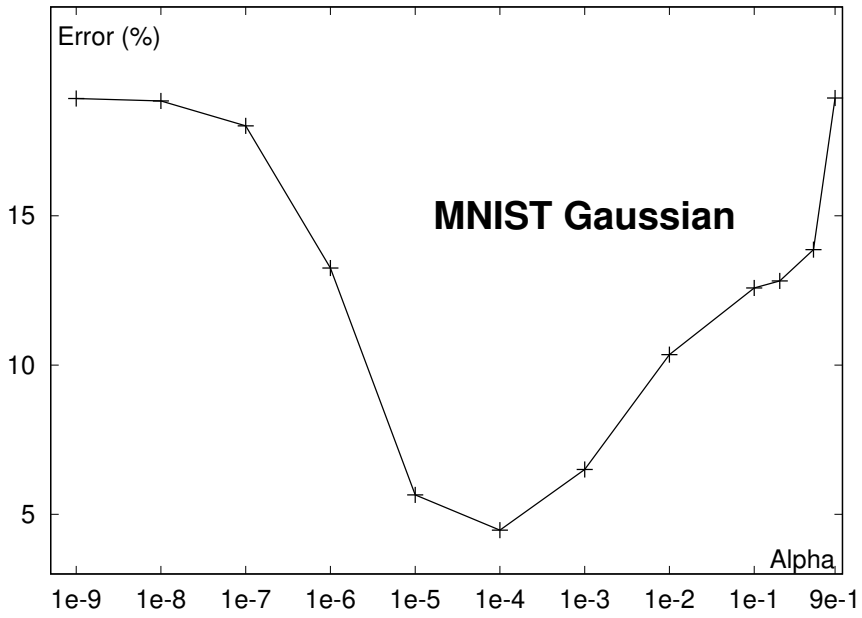


Figura 3: Error del clasificador gaussiano en función del parámetro de suavizado α.

gráficamente los resultados obtenidos donde cada curva es un valor constante de α y el eje x representa la variación en el número de componentes PCA. Finalmente, entrena un clasificador final con la mejor combinación de parámetros y evalúa en el conjunto de test para comparar el resultado con el proporcionado en la web de MNIST.

6. Ejercicios opcionales

En esta sección, se plantean ejercicios opcionales adicionales que junto con el anterior ejercicio opcional puede sumar hasta 0.5 puntos a la nota de prácticas:

- Implementación y evaluación de un clasificador Bernoulli al igual que se ha hecho con el clasificador multinomial. En este caso será necesario convertir previamente las imágenes que originalmente están en escala de grises a blanco y negro (binarización). Para ello debes aplicar un umbral de binarización, es decir, el nivel de la escala gris por encima del cual un píxel se considera negro y por debajo, blanco (0.5 puntos).
- Combinación de clasificadores fuertes mediante Bagging: clasificadores de vecinos más cercanos (0.5 puntos).

Por parte del alumnado se puede proponer la implementación de otros clasificadores distintos a los propuestos en los ejercicios opcionales que requerirán la valoración y aprobación del profesorado de la asignatura.

A. Recomendaciones ejercicios opcionales k-NN

En esta sección se dan algunas sugerencias a la hora de afrontar la implementación de los ejercicios opcionales relacionados con el clasificador k-NN. Estas sugerencias son un compromiso entre eficiencia temporal/espacial y simplicidad en la implementación. Recuerda que cuando nos referimos a los prototipos del conjunto de entrenamiento, son simplemente las muestras del conjunto de entrenamiento que están dispuestas por filas en la matriz de datos.

A.1. Implementación de distancias *L0*, *L1* y *L3*

Se recomienda tomar como referencia la implementación de distancia Euclídea L2 implementada en el fichero `L2dist.py`, pero teniendo en cuenta que no será posible realizar una implementación tan eficiente. Así que el código de referencia para implementar las distancias *L0*, *L1* y *L3* es la versión no tan eficiente de *L2* que aparece comentada al final del mismo fichero:

```
1 def L2dist(X,Y):
2   D=np.zeros((X.shape[0],Y.shape[0]));
3   for i in range(X.shape[0]):
4     D[i] = np.sum(np.square(X[i]-Y),axis=1);
5   return D;
```

En la línea 3 se observa como el bucle recorre las muestras del conjunto `X` que están dispuestas por filas. En la línea 3 se calcula mediante una operación matricial la distancia de cada muestra `X[i]` a todos los prototipos en `Y`. Esto resulta en un vector fila que se asigna a la fila `i` de la matriz de distancias `D`. La matriz `D` en su posición `D[i,j]` proporciona la distancia del prototipo `X[i]` a la muestra `Y[j]`.

La implementación de las distancias *L0*, *L1* y *L3* es un modificación de esta versión de la función `L2dist`.

A.2. Implementación de distancia Euclídea ponderada

En esta sección nos centraremos en la implementación de la distancia Mahalanobis-diagonal por clase, pero la misma idea se puede aplicar para el cálculo de distancia Euclídea normalizada (Mahalanobis-diagonal) o Mahalanobis-Local simplemente cambiando la matriz de pesos `W` que describiremos a continuación.

El primer paso es implementar una función que calcule la matriz `W` correspondiente a los pesos de Mahalanobis-diagonal por clase:

```
def Md4cweight(X,xl,alpha):
  # HERE YOUR CODE
  return W;
```

La dimensionalidad de la matriz `W` coincide con la de la matriz `X`, ya que la fila `W[i]` son los pesos correspondientes al prototipo `X[i]` que dependen de la clase `xl[i]` de este prototipo.

Como sabes, Mahalanobis-diagonal por clase requiere el cálculo de la varianza de los prototipos de cada clase y para ello será necesario un bucle `for` sobre el conjunto de clases. En este bucle puedes seleccionar las muestras de cada clase con la ayuda de la función `where` de `numpy`. El cálculo de la varianza se puede realizar usando la función `var` también de `numpy`, o implementando la estimación de la varianza.

El suavizado de las varianzas estimadas a partir de los datos $\hat{\boldsymbol{\sigma}}_c$ de una clase se puede realizar directamente con varianzas unitarias $\mathbf{1}$:

$$\tilde{\boldsymbol{\sigma}}_c = \alpha \cdot \hat{\boldsymbol{\sigma}}_c + (1 - \alpha) \cdot \mathbf{1}$$

siendo $\hat{\boldsymbol{\sigma}}_c$ y $\mathbf{1}$ la diagonal de la matriz de covarianzas $\hat{\Sigma}_c$ y matriz identidad I, respectivamente.

El segundo paso, una vez calculada la matriz `W` debes implementar la función que calcula la distancia Euclídea ponderada:

```
def wL2dist(X,Y,W):
  # HERE YOUR CODE
  return D;
```

En este punto, hay dos opciones. La primera es ir directamente a una implementación eficiente al estilo de `L2dist` mediante producto de matrices donde se integraría la matriz `W`. La segunda opción es pasar por una versión menos eficiente con un bucle `for` como se recomienda para las distancias *L0*, *L1* y *L3*, que posibilita comprender las operaciones que se están realizando antes de pasar a la versión eficiente. En esta segunda opción se recomienda realizar un bucle `for` recorriendo los prototipos en `X`:

```
1 def wL2dist(X,Y,W):
2   D=np.zeros((X.shape[0],Y.shape[0]));
3   for i in range(X.shape[0]):
4     D[i,:] = # HERE YOUR CODE
5   return D;
```

donde la matriz `D` se construye por filas y así se calcula la distancia del prototipo `X[i,:]` con su peso asociado `W[i,:]` a cada una de las muestras en `Y`.

A.3. Implementación del algoritmo de Wilson

En este caso es recomendable la utilización de vectores de índices para simplificar la implementación. Más concretamente, la función que implementa la aplicación del algoritmo de Wilson al conjunto `X` mediante el clasificador de `k` vecinos más cercanos puede ser:

```
def wilson(X,xl,k):
  # HERE YOUR CODE
  return ind;
```

siendo `ind` un vector de índices que indica que prototipos (filas) del conjunto `X` no han sido eliminados por el algoritmo. Se inicializa este vector a `ind = np.array(range(X.shape[0]))` y se van eliminando aquellos prototipos que son mal clasificados por el clasificador de `k` vecinos más cercanos. Para eliminar una muestra `i` del vector `ind` se puede utilizar la función predefinida `setdiff1d` de `numpy`.

La función `wilson` requiere de dos funciones auxiliares que aceleran y simplifican su implementación. Primero, debes implementar una función auxiliar que precalcule los vecinos más cercanos a cada prototipo en forma de vector de índices, de manera que tengamos un vector de índices a vecinos ordenados por distancia (de más a menos cerca):

```
def mnn(X,xl,m):
  # HERE YOUR CODE
  return V;
```

siendo `V` la matriz que almacena por columnas los índices de los `m` vecinos más cercanos a cada prototipo en `X`. Esta función se puede implementar fácilmente modificando la función `knn(X,xl,Y,yl,k)` que os proporcionamos como parte de código del proyecto.

En cuanto a la función `mnn`, es necesario limitar a `m` el número de vecinos que almacenamos de cada prototipo porque para MNIST almacenar todos los vecinos puede ser excesivo:

$$60000 \cdot 60000 \cdot 4 \text{ bytes}/1024^3 = 13.41 \text{ Gbytes}$$

Por otra parte, es importante comprender que para clasificar un prototipo `i` mediante el clasificador de `k` vecinos utilizaremos su vector de vecinos más cercanos precalculados y almacenados en `V[:,i]`, pero es posible que cuando los utilicemos algunos de ellos ya hayan sido descartados por Wilson. Sin embargo, los prototipos que todavía no han sido eliminados están indexados mediante el vector `ind`. Así que los vecinos de `V[:,i]` que todavía podemos utilizar para clasificar son aquellos que todavía están en `ind`. Por ello, es recomendable utilizar una segunda función auxiliar que clasifica mediante los `k` vecinos más cercanos teniendo en cuenta `V[:,i]` e `ind`:

```
def knnV(Vi,ind,xl,k):
  ...
  return c;
```

siendo `Vi=V[:,i]` y `c` la clase en la que se clasifica el prototipo `i`. No utilices la función `intersect1d` de `numpy` pues ordena numéricamente los índices en `V[:,i]`, en su lugar utiliza la función `isin` también de `numpy`.

Estas dos funciones auxiliares, `mnn` y `knnV`, junto con el pseudocódigo del algoritmo de Wilson permiten implementar una versión bastante eficiente.

El número de vecinos `m` que es suficiente precalcular en Wilson para MNIST utilizando `k=1` en `knnV` es 100, es decir, `m=100` que ocupará:

$$60000 \cdot 100 \cdot 4\text{bytes}/1024^3 = 0.22\text{Gbytes}$$

Obviamente, la determinación de este valor `m` depende de la tarea MNIST y del número de `k` vecinos empleado por el clasificador. La solución aquí propuesta es *ad hoc*, pero es suficiente para los objetivos de la asignatura.