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

Proyecto de prácticas. Reconocimiento de dígitos manuscritos: MNIST

Jorge Civera, Javier Iranzo, Alfons Juan, Francisco Casacuberta, Enrique Vidal
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Última actualización: 30/11/2021- 20:23:53

Índice

1. Objetivos

En el marco de un aprendizaje basado en proyectos, este proyecto de prácticas es la continuación del realizado en la asignatura de Percepción del pasado curso académico. Al igual que en Percepción, 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 desglosa en subobjetivos básicos que se pueden entender como fases o hitos de este proyecto:

1. Implementar el algoritmo EM para mixtura de gaussianas.
2. Evaluar el clasificador de mixtura de gaussianas y su interacción con la técnica de reducción de dimensionalidad *Principal Component Analysis* PCA en la tarea MNIST.
3. Comprender los conceptos teóricos de *Support Vector Machines* (SVM) mediante su aplicación a pequeñas tareas de clasificación.
4. Evaluar el clasificador basado en SVM en la tarea MNIST.
5. Comprender y aplicar el preproceso necesario a un conjunto de datos para entrenar y evaluar una red neuronal.
6. Evaluar un clasificador basado en redes neuronales en la tarea MNIST, y su interacción con la técnica PCA.

Para la realización de este proyecto de prácticas se supone que previamente has adquirido experiencia en el uso de `octave`, `gnuplot` y *shell scripts*, tanto en la asignatura de Sistemas Inteligentes como en la asignatura de Percepción. Si no es el caso, puedes refrescar tus conocimientos y habilidades sobre dichas herramientas recurriendo al tutorial incluido en el Apéndice A. Este tutorial es el mismo que se utilizó en la asignatura de Percepción. Asimismo, te recomendamos que refresques la descripción de la tarea MNIST que encontrarás en el Apéndice B. La tarea MNIST, así como otros pequeños conjuntos de datos de prueba que se utilizaran en este proyecto, están disponibles en PoliformaT a través del fichero `data.tgz`.

Como regla general, se recomienda leer en su totalidad la sección correspondiente al clasificador con el que se trabajará en las siguientes sesiones. Esto permite centrarse en el trabajo a desarrollar en la sesión de laboratorio y aprovechar mejor el tiempo.

Finalmente, la evaluación del proyecto de la asignatura consta de un total de 3 puntos que se distribuyen de manera uniforme con 1 punto para cada tipo de clasificador.

2. Mixtura de gaussianas

En esta primera parte del proyecto estudiaremos una generalización del clasificador gaussiano visto en la asignatura de Percepción (ver Apéndice C), el clasificador de

mixtura de gaussianas. Las mixturas, ya sean de gaussianas o de cualquier otra distribución, nos permiten introducir una instanciación concreta del algoritmo EM estudiado en teoría. Por ello, primero desarrollaremos la teoría asociada a la estimación de los parámetros de una mixtura de gaussianas en el marco del algoritmo EM, y seguidamente veremos su implementación con las peculiaridades prácticas que ello conlleva.

Los ficheros necesario para desarrollar esta parte del proyecto están disponibles en Poliformat en el fichero `mixgaussian.tgz`.

2.1. Algoritmo EM

Las mixturas de distribuciones de probabilidad, en nuestro caso condicionada a la clase $p(\mathbf{x} \mid c)$, se desarrollan introduciendo una variable oculta k que indica la componente de la mixtura que genera la muestra $\mathbf{x}$

$$p(\mathbf{x} \mid c) = \sum_{k=1}^{K} p(\mathbf{x}, k \mid c) = \sum_{k=1}^{K} p(k \mid c)\, p(\mathbf{x} \mid k, c) \tag{1}$$

donde $p(k \mid c)$ es la probabilidad a priori de la componente condicionada a la clase, y $p(\mathbf{x} \mid k, c)$ es una distribución de probabilidad condicionada no sólo a la clase, sino también a la componente dentro de esa clase. En nuestro caso $p(\mathbf{x} \mid k, c)$ está modelizada mediante una distribución gaussiana

$$p(\mathbf{x} \mid k, c) \sim \mathcal{N}_D(\boldsymbol{\mu}_{ck}, \Sigma_{ck}), \quad c = 1, \ldots, C \quad k = 1, \ldots, K$$

pero podría haber sido modelizada mediante una distribución multinomial o cualquier otra distribución de probabilidad.

Observad como la componente de cada clase tendrá su propia media y matriz de covarianzas. El conjunto de parámetros de este modelo de mixturas es

$$\boldsymbol{\Theta} = (P_1, \cdots, P_C, \boldsymbol{\Theta}_1, \cdots \boldsymbol{\Theta}_C)\,,$$

donde

$$\boldsymbol{\Theta}_c = (p_{c1}, \cdots, p_{cK}, \mu_{c1} \cdots \mu_{cK}, \Sigma_{c1}, \cdots, \Sigma_{cK})\,.$$

La estimación de estos parámetros sería trivial si para cada muestra supiéramos que componente de cada clase la generó, es decir, si conocieramos el valor de z_n. En ese caso, la estimación sería:

$$\hat{P}(c) = \frac{N_c}{N} \tag{2}$$

$$\hat{p}_{ck} = \frac{\sum\limits_{n\,:\,c_n=c \wedge z_n=k} 1}{N_c} = \frac{N_{ck}}{N_c} \tag{3}$$

$$\hat{\boldsymbol{\mu}}_{ck} = \frac{1}{N_{ck}} \sum_{n\,:\,c_n=c \wedge z_n=k} \boldsymbol{x}_n \tag{4}$$

$$\hat{\Sigma}_{ck} = \frac{1}{N_{ck}} \sum_{n\,:\,c_n=c \wedge z_n=k} (\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_{ck})(\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_{ck})^t \tag{5}$$

donde N_{ck} es el número de muestras de la componente k de la clase c. Es decir, simplemente tendríamos que calcular los parámetros de cada componente de cada clase con las muestras que sabemos, gracias a z_n, que pertenecen a dicha componente de esa clase.

Sin embargo, desconocemos qué componente generó cada muestra, es decir, el valor de cada z_n está oculto. Por ello, como has estudiado en teoría, el algoritmo EM nos permite estimar de manera iterativa los parámetros de un modelo (paso M) donde hay variables ocultas, calculando una distribución de probabilidad sobre dichas variables ocultas (paso E).

El paso E en la iteración t de un modelo de mixturas dada la estimación actual de los parámetros $\boldsymbol{\Theta}^{(t)}$ es:

$$\begin{aligned} z_{nk}^{(t)} &= p(z_n = k \mid \mathbf{x}_n, c_n = c; \boldsymbol{\Theta}^{(t)}) \\ &= \frac{p(z_n = k,\, \mathbf{x}_n,\, c_n = c; \boldsymbol{\Theta}^{(t)})}{p(\mathbf{x}_n, c_n = c; \boldsymbol{\Theta}^{(t)})} \\ &= \frac{p(z_n = k,\, \mathbf{x}_n,\, c_n = c; \boldsymbol{\Theta}^{(t)})}{\sum_{k'=1}^{K} p(z_n = k',\, \mathbf{x}_n,\, c_n = c; \boldsymbol{\Theta}^{(t)})} \\ &= \frac{p(z_n = k \mid c_n = c; \boldsymbol{\Theta}^{(t)})\, p(\mathbf{x}_n \mid z_n = k,\, c_n = c; \boldsymbol{\Theta}^{(t)})}{\sum_{k'=1}^{K} p(z_n = k' \mid c_n = c; \boldsymbol{\Theta}^{(t)})\, p(\mathbf{x}_n \mid z_n = k',\, c_n = c; \boldsymbol{\Theta}^{(t)})} \\ &= \frac{p(k \mid c_n)\, p(\mathbf{x}_n \mid k,\, c_n)}{\sum_{k'=1}^{K} p(k' \mid c_n)\, p(\mathbf{x}_n \mid k',\, c_n)} \end{aligned} \tag{6}$$

Es decir, la probabilidad de la muestra $\mathbf{x}$ de acuerdo a la distribución de probabilidad de la componente k de la clase c. También, se puede interpretar como el grado de pertenencia de la muestra $\mathbf{x}$ a la componente k de la clase c. Desde el punto de vista del paso M, que ahora detallaremos, z_{nk} es el peso o contribución parcial de una muestra a una componente, de forma que una misma muestra puede contribuir parcialmente a la estimación de los parámetros de varias componentes.

En el paso M se estiman los parámetros del modelo teniendo una estimación del grado de pertenencia de cada muestra de entrenamiento a cada componente. La estimación de la probabilidad a priori de cada clase queda fuera de la estimación por el algoritmo EM (ver Ec. 2). El resto de parámetros siguen una estimación similar a las Ecs. 3, 4 y 5, pero teniendo en cuenta la idea de que una muestra contribuye parcialmente acorde a z_{nk} a la estimación de los parámetros de la componente k en la clase c:

$$\hat{p}_{ck}^{(t+1)} = \frac{1}{N_c} \sum_{n\,:\,c_n = c} z_{nk}^{(t)} \tag{7}$$

$$\hat{\boldsymbol{\mu}}_{ck}^{(t+1)} = \frac{1}{\sum\limits_{n\,:\,c_n = c} z_{nk}^{(t)}} \sum_{n\,:\,c_n = c} z_{nk}^{(t)}\, \boldsymbol{x}_n \tag{8}$$

$$\hat{\Sigma}_{ck}^{(t+1)} = \frac{1}{\sum\limits_{n\,:\,c_n = c} z_{nk}^{(t)}} \sum_{n\,:\,c_n = c} z_{nk}^{(t)} (\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_{ck})(\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_{ck})^t \tag{9}$$

2.2. Implementación

Antes de pasar a la implementación es necesario tratar la inicialización del algoritmo EM. La manera más habitual de inicializar el algoritmo EM es definir una inicialización de los parámetros del modelo $\boldsymbol{\Theta}^{(0)}$ que dependen de la variable oculta, pero también sería posible definir una inicialización de la distribución de probabilidad sobre la variable oculta.

En nuestro caso, inicializaremos los parámetros correspondientes a la probabilidad a priori, la media y matriz de covarianza de cada componente. Seguidamente se muestra el código utilizado para la inicialización:

```
24 sigma=cell(C,K);
25 for c=classes'
26   ic=find(c==classes);
27   % Initialization of component priors p(k|c) as uniform distro
28   pkGc{ic}(1:K)=1/K;
29   % Initialization of K component means mu_kc as K random samples
30   % from class c
31   idc=find(xl==c);
32   Nc=rows(idc);
33   mu{ic}=X(idc(randperm(Nc,K)),:)';
34   % Initialization of K component covariance sigma_kc as K class
35   % covariance matrix divided by the number of components K
36   sigma(ic,1:K)=alpha*cov(X(idc,:),1)/K+(1-alpha)*eye(D);
37 end
```

La línea 24 define una matriz de celdas $C \times K$ para almacenar las matrices de covarianza de cada una de las C clases y K componentes por clase. En el bucle desde la línea 25 a la línea 37 se inicializan los parámetros de cada clase. La línea 28 inicializa la probabilidad de cada componente como una distribución uniforme. La línea 33 define la media de cada componente como una muestra aleatoria de la clase c. Finalmente, en la línea 36 se inicializa la matriz de covarianzas de cada componente como la matriz de covarianzas de la clase dividida por el número de componentes `cov(X(idc,:),1)/K`, que es suavizada mediante *flat smoothing* con la matriz identidad. Como puedes deducir está inicialización es arbitrárea, pero ha demostrado funcionar bien en la práctica.

Al igual que en el clasificador gaussiano haremos uso de una función auxiliar para calcular la probabilidad de cada muestra[1], pero en este caso para cada componente k. Esto se corresponde con el logaritmo del numerador de la Ec. 6, es decir, $\log p(k \mid c) + \log p(\mathbf{x} \mid k, c)$:

```
121 function [zk] = compute_zk(ic,k,pkGc,mu,sigma,X)
122   D=columns(X);
```

[1] De cada muestra, pero todas ellas a la vez, independientemente y en paralelo, mediante operaciones matriciales aprovechando las capacidades de Octave.

```
123   I=pinv(sigma{ic,k});
124   cons=log(pkGc{ic}(k));
125   cons=cons-0.5*D*log(2*pi);
126   cons=cons-0.5*logdet(sigma{ic,k});
127   cons=cons-0.5*mu{ic}(:,k)'*I*mu{ic}(:,k);
128   lin=X*I*mu{ic}(:,k);
129   qua=-0.5*sum((X*I).*X,2);
130   zk=qua+lin+cons;
131 end
```

Observarás que es casi idéntica a la función auxiliar del clasificador gaussiano, a excepción de la línea 125 que incluye el término constante porque vamos a hacer una estimación de la probabilidad a posteriori y no únicamente una clasificación como en el clasificador gaussiano.

Una vez han sido inicializados los parámetros, el algoritmo EM ejecuta los pasos E y M de manera iterativa hasta convergencia (o un número máximo de iteraciones). En nuestra implementación la condición de convergencia es que el incremento relativo de la log verosimilitud entre dos iteraciones consecutivas no supere cierto umbral

$$\frac{\mid L(\boldsymbol{\Theta}; \boldsymbol{X})^{(t+1)} - L(\boldsymbol{\Theta}; \boldsymbol{X})^{(t)} \mid}{\mid L(\boldsymbol{\Theta}; \boldsymbol{X})^{(t)} \mid} < \epsilon$$

donde

$$L(\boldsymbol{\Theta}; \boldsymbol{X}) = \sum_n \log p(c_n) + \log p(\mathbf{x}_n \mid c_n)$$

donde $p(\mathbf{x}_n \mid c_n)$ se define en la Ec. 1. De manera similar, $p(\mathbf{x}_n \mid c_n)$ es el denominador de la Ec. 6, es decir, que calcularemos $p(\mathbf{x}_n \mid c_n)$ como el sumatorio sobre las K componentes

$$p(\mathbf{x}_n \mid c_n) = \sum_{k'=1}^{K} p(k' \mid c_n)\, p(\mathbf{x}_n \mid k',\, c_n)$$

invocando la función `compute_zk` para cada componente. En cada iteración del algoritmo EM se ejecutan los pasos E y M por cada clase, estimando z_{nk} y $\boldsymbol{\Theta}^{(t+1)}$, respectivamente:

```
50   for c=classes'
51     % E step: Estimate znk
52     ic=find(c==classes);
53     idc=find(xl==c);
54     Nc=rows(idc);
55     Xc=X(idc,:);
56     z=[];
57     for k=1:K
58       z(:,k)=compute_zk(ic,k,pkGc,mu,sigma,Xc);
59     end
60     % Robust computation of znk and log-likelihood
```

```
61      maxz=max(z,[],2);
62      z=exp(z-maxz);
63      sumz=sum(z,2);
64      z=z./sumz;
65      L=L+Nc*log(pc(ic))+sum(maxz+log(sumz));
66
67      % M step: parameter update
68      % Weight of each component
69      sumz=sum(z);
70      pkGc{ic}=sumz/Nc;
71      mu{ic}=(Xc'*z)./sumz;
72      for k=1:K
73        covar=((Xc-mu{ic}(:,k)')'*((Xc-mu{ic}(:,k)').*z(:,k)))/sumz(k);
74        % Smoothing covariance matrix with identity matrix
75        sigma(ic,k)=alpha*covar+(1-alpha)*eye(D);
76      end
77    end
78
79    % Likelihood divided by the number of training samples
80    L=L/N;
```

Las líneas 51 a 65 definen el paso E, mientras que las líneas 67 a 77 implementan el paso M.

En cuanto al paso E, la línea 58 estima la log probabilidad de cada muestra de la clase c para cada componente k dando lugar a cada vector columna de la matriz `z`. Seguidamente en las líneas 61 a 64 se realiza la estimación de z_{nk} de la Ec. 6, pero de manera *robusta* para manejar valores pequeños de log probabilidad[2].

$$
\begin{aligned}
z_{nk}^{(t)} &= \frac{p(k \mid c) \cdot p(\mathbf{x} \mid k,\, c)}{\sum_{k'=1}^{K} p(k' \mid c) \cdot p(\mathbf{x} \mid k',\, c)} \\
&= \frac{\frac{p(k|c)\cdot p(\mathbf{x}|k,\,c)}{\max_{k''} p(k''|c)\cdot p(\mathbf{x}|k'',\,c)}}{\sum_{k'=1}^{K} \frac{p(k'|c)\cdot p(\mathbf{x}|k',\,c)}{\max_{k''} p(k''|c)\cdot p(\mathbf{x}|k'',\,c)}} \\
&= \frac{\exp\left(\log\left(\frac{p(k|c)\cdot p(\mathbf{x}|k,\,c)}{\max_{k''} p(k''|c)\cdot p(\mathbf{x}|k'',\,c)}\right)\right)}{\sum_{k'=1}^{K} \exp\left(\log\left(\frac{p(k'|c)\cdot p(\mathbf{x}|k',\,c)}{\max_{k''} p(k''|c)\cdot p(\mathbf{x}|k'',\,c)}\right)\right)} \\
&= \frac{\exp\left(\log p(k \mid c) + \log p(\mathbf{x} \mid k,\, c)\right) - \max_{k''} \log p(k'' \mid c) + \log p(\mathbf{x} \mid k'',\, c))}{\sum_{k'=1}^{K} \exp\left(\log p(k' \mid c) + \log p(\mathbf{x} \mid k',\, c) - \max_{k''} \log p(k'' \mid c) + \log p(\mathbf{x} \mid k'',\, c)\right)}
\end{aligned}
$$

Es decir, para cada muestra se divide la log probabilidad de cada componente por la log probabilidad de aquella componente con máxima log probabilidad. Después se

[2] https://en.wikipedia.org/wiki/LogSumExp

calcula el logaritmo compensado con la exponenciación y se normaliza por la suma del total de las componentes.

Sobre el código, la línea 61 calcula la componente de máxima log probabilidad, restamos el máximo de la log probabilidad en la línea 62, calculamos la suma para todas las componentes en la línea 63 y normalizamos en la línea 64. Nótese que estas operaciones se hacen a la vez para todas las muestras de la clase c aprovechando la capacidad de Octave de trabajar con matrices y vectores de manera más eficiente.

La log verosimilitud para las muestras de la clase c se calcula en la línea 65 aprovechando el cálculo robusto realizado

$$L(\boldsymbol{\Theta}; \boldsymbol{X}) = \sum_c L(\boldsymbol{\Theta}_c; \boldsymbol{X}_c)$$

donde

$$\begin{aligned} L(\boldsymbol{\Theta}_c; \boldsymbol{X}_c) &= \sum_{n\,:\,c_n=c} \log P(c_n) + \log p(\mathbf{x}_n \mid c_n) \\ &= N_c \log P(c_n) + \sum_{n\,:\,c_n=c} \log \sum_{k=1}^{K} p(\mathbf{x}_n, k \mid c_n) \end{aligned}$$

siendo $p(\mathbf{x}_n \mid c_n)$ el denominador de z_{nk} que se calcula en la línea 63. Sin embargo, debido al cálculo robusto a $p(\mathbf{x}_n \mid c_n)$ le ha sido restado el máximo de cada componente, y por tanto para compensar debemos sumar el máximo de cada componente como se observa en la línea 65. En la línea 80, una vez se han procesado todas las muestras, se puede observar como la log verosimilitud se normaliza por el número de muestras, pero es algo opcional.

En cuanto al paso M (líneas 69-76) es una implementación directa de las Ecs. 7, 8 y 9 aprovechando las operaciones matriciales en Octave.

La clasificación de cada muestra es un cálculo muy similar a la estimación de la log verosimilitud, ya que la función discriminante que se calcula es la probabilidad de cada muestra de acuerdo a los parámetros de cada una de las clases involucradas.

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

Finalmente, el fichero `mixgaussian.m` de PoliformaT contiene la implementación descrita anteriomente. Por favor, dedica unos minutos a leer el código y comprobar que comprendes la implementación ya realizada.

Ejercicio opcional (no entregable). Para realizar una primera comprobación del correcto funcionamiento del clasificador de mixtura de gaussianas, compara su funcionamiento para el caso de una única componente por mixtura con el clasificador gaussiano. Para ello, realiza un experimento con la misma partición que se define en el experimento del clasificador gaussiano descrita en el Apéndice C utilizando el script `mixgaussian-exp.m`.

2.3. Tarea MNIST

En esta sección vamos a aplicar el clasificador de mixtura de gaussianas a la tarea MNIST para estudiar si sus resultados son competitivos respecto a otros clasificadores.

Ejercicio 1 (0.5 puntos). Realiza un experimento para evaluar el error del clasificador en función del número de componentes por mixtura ($K = 1, 2, 5, 10, 20, 50, 100$), para un número de dimensiones PCA variable ($D = 1, 2, 5, 10, 20, 50, 100$) y probando algunos de los valores de α de suavizado *flat smoothing* que mejores resultados han proporcionado en el clasificador gaussiano. Para ello, puedes partir del script `mixgaussian-exp.m` para crear el script `pca+mixgaussian-exp.m` que mantenga la misma partición en entrenamiento y validación, y realice la exploración de parámetros propuesta.

Recuerda que el objetivo de este experimento es determinar los valores de los parámetros del clasificador (K, D y α) que minimizan el error de clasificación en el conjunto de validación. En el ejercicio 2, estos valores óptimos de los parámetros serán utilizados para entrenar y evaluar un clasificador final en los conjuntos oficiales MNIST de entrenamiento y test, respectivamente.

Representa gráficamente las tasas de error obtenidas en el conjunto de validación. Como punto de partida puedes utilizar el fichero `gaussian-exp.gnp` que utiliza el fichero de resultados `gaussian-exp.out` para generar la gráfica `gaussian-exp.eps` (todos ellos disponibles en PoliformaT). Te recomendamos que generes una gráfica independiente por cada valor de suavizado α que hayas evaluado. A su vez, cada gráfica tendrá tantas curvas como valores de PCA diferentes hayas probado. Finalmente, cada curva mostrará la evolución del error de clasificación (eje y) en función del número de componentes por mixtura (eje x). Para mejorar la legibilidad de la representación gráfica de los resultados puedes descartar aquellas curvas asociadas a valores de PCA cuyas tasas de error sean muy elevadas.
Pista: Las tasas de error en MNIST en el conjunto de validación para una partición 90 % entrenamiento - 10 % validación (semilla aleatoria para el barajado número `23`*) con proyección a 20 dimensiones de PCA, y 1, 2 y 5 componentes por mixtura en convergencia con suavizado α =1e-4 son 6.17 %, 5.35 % y 4.35 %, respectivamente.*

Ejercicio 2 (0.2 puntos). Como hemos comentado anteriormente, en este ejercicio utilizaremos los valores óptimos de los parámetros del clasificador para entrenar y evaluar un clasificador final en los conjuntos oficiales MNIST de entrenamiento y test, respectivamente. Para ello te recomendamos que tomes como punto de partida el script `pca+mixgaussian-exp.m`, modificándolo adecuadamente para generar el script

`pca+mixgaussian-eva.m` que también deberá recibir como entrada el conjunto de test de MNIST. Recuerda que toda estimación de (la probabilidad de) error de un clasificador final, debe ir acompañada de sus correspondientes intervalos de confianza al 95 %. Discute los resultados obtenidos comparándolos con los obtenidos con el clasificador gaussiano y con los reportados en la tarea MNIST con clasificadores que conozcas.

Ejercicio 3 (0.3 puntos). Modifica el código de aprendizaje del clasificador con mixturas de Gaussianas con el fin de mejorar el error en validación. Algunas ideas que puedes evaluar son la terminación temprana del proceso iterativo del algoritmo EM, un número de componentes diferente en cada clase o una inicialización alternativa del modelo.

3. Máquinas de Vectores Soporte

En esta parte del proyecto se experimenta con SVM mediante la librería `LibSVM`[3] desarrollada para Octave. Los ejecutables de esta librería compilados para la versión de Octave instalada en los laboratorios está disponible en PoliformaT en el fichero `svm_apr.tgz`.

Para poder emplear dicha librería, descomprime el fichero `svm_apr.tgz`, que generará el directorio `svm_apr` con los ejecutables `svm-train.mex` y `svm-predict.mex`. Simplemente necesitarás añadir la ruta del directorio `svm_apr` al `PATH` desde Octave mediante el comando:

```
addpath("svm_apr");
```

Si deseas utilizar esta librería en tu propio ordenador, deberás compilar tu propia versión de ambos ejecutable a partir del código fuente.

3.1. Ejemplo de entrenamiento y clasificación con `LibSVM`

Según la teoría de SVM, dado un conjunto de aprendizaje S, formado por N vectores y sus correspondientes etiquetas de clase, el sistema de aprendizaje obtiene los *multiplicadores de Lagrange*, α_n óptimos asociados a cada vector x_n de S, $1 \leq n \leq N$. Los multiplicadores no nulos definen los vectores de S que constituyen el *soporte* de la función discriminante lineal del clasificador (en dos clases, por el momento). Concretamente, a partir de los multiplicadores no-nulos de los correspondientes vectores soporte y de sus etiquetas de clase, se obtienen fácilmente el vector de pesos $\boldsymbol{\theta}$ y el término independiente o *umbral*, θ_0, que definen la función discriminante lineal del clasificador aprendido.

Por tanto, una vez obtenidos los multiplicadores no nulos, se puede construir de forma trivial un clasificador que prediga la etiqueta de clase asociada a cualquier vector de validación o test.

`LibSVM`, implementa el proceso de aprendizaje mediante `svmtrain.mex`, y la clasificación mediante `svmpredict.mex`. Un detalle importante de esta librería es que los multiplicadores obtenidos por el proceso de aprendizaje, y los que usan en clasificación, pueden ser positivos o negativos. Se asume que los positivos corresponden a vectores soporte de la clase $+1$ y los negativos, a los de la -1. A todos los efectos, cualquiera de estos multiplicadores (que el toolkit denota como `sv_coef`), puede considerarse como el producto del verdadero multiplicador de Lagrange, por la etiqueta de clase del vector soporte correspondiente, es decir, $c_n\alpha_n$.

3.1.1. Corpus de ejemplo

Para introducir la funcionalidad básica de la librería `LibSVM` en Octave se propone emplear el corpus de `hart` que se encuentra en el paquete `data.tgz` disponible en PoliformaT. Es un corpus de dos clases donde los datos se representan en dos dimensiones

[3] http://www.csie.ntu.edu.tw/∼cjlin/libsvm

($\mathbb{R}^2$). Este conjunto de datos *no* es linealmente separable en su espacio de representación original, pero sí lo es en un espacio transformado mediante un *kernel de base radial* (RBF), cuya dimensionalidad es mucho mayor. Los pasos a seguir podrían ser los siguientes.

Primero ejecutamos Octave en el directorio `svm_apr` resultado de descomprimir el paquete `svm_apr.tgz`. Una vez en Octave, cargamos los ficheros con los datos y las etiquetas de clase del conjunto de entrenamiento:

```
octave:1> load data/hart/tr.dat ; load data/hart/trlabels.dat
```

Una vez cargados los datos podemos visualizarlos mediante:

```
octave:2> plot (X(:,1),X(:,2),"x")
```

donde el primer parámetro son los valores de la primer dimensión, el segundo parámetro son los valores de la segunda dimensión, y el tercer parámetro es una cadena de formato que especifica la representación gráfica de los puntos. En el Apéndice D encontrarás más información sobre el comando `plot`.

Si se examinan `trlabels.dat` y `tslabels.dat`, puede observarse que las etiquetas de clase son "1" y "2". `LibSVM` admite más de dos clases, que pueden denotarse mediante etiquetas numéricas. En el caso de dos clases, estas etiquetas se convierten internamente en $+1$ y -1 y esto se refleja en los signos de los multiplicadores (`sv_coef`), como se ha explicado anteriormente. Podemos visualizar separadamente las muestras de cada clase mediante:

```
octave:3> plot(X(xl==1,1),X(xl==1,2),"x",X(xl==2,1),X(xl==2,2),"s")
```

donde los tres primeros parámetros corresponden a la representación gráfica de la clase "1", y los tres siguientes parámetros, a la de la clase "2". Se puede observar cómo estas muestras *no* son linealmente separables en su espacio original $\mathbb{R}^2$.

3.1.2. Entrenamiento

Como se ha mencionado anteriormente, la función `svmtrain` implementa el proceso de aprendizaje. Se puede obtener una breve ayuda sobre el uso de `svmtrain` invocándo la función sin argumentos:

```
octave:4> svmtrain
Usage: model = svmtrain(training_label_vector,
               training_instance_matrix, 'libsvm_options');
libsvm_options:
[...]
```

Como resultado podemos ver los diferentes parámetros que podemos usar para el aprendizaje: tipo de kernel, parámetro $\mathcal{C}$, grado del kernel si es polinomial, etc. Por ejemplo, para realizar un entrenamiento con kernel tipo RBF (`-t 2`) con parámetro $\mathcal{C} = 1$ (`-c 1`):

```
octave:5> res = svmtrain(xl, X, '-t 2 -c 1');
.*.*
optimization finished, #iter = 2298
nu = 0.174579
obj = -108.403744, rho = -0.096434
nSV = 398, nBSV = 91
Total nSV = 398
```

El resultado del proceso de entrenamiento se ha almacenado en la variable `res`, que es un tipo estructurado que contiene, entre otros: los parámetros del modelo, los índices de los vectores que han resultado ser vectores soporte, el multiplicador de Lagrange (multiplicado por la correspondiente etiqueta de clase, $+1$ o -1) asociado a cada vector soporte, etc. El contenido completo de `res` puede visualizarse mediante:

```
octave:6> res
```

Se puede acceder a cada uno de los campos de `res` mediante el operador ".". Por ejemplo se pueden mostrar los índices de los vectores soporte mediante:

```
octave:7> res.sv_indices
```

Y para mostrar los multiplicadores de Lagrange (multiplicados por las correspondientes etiquetas de clase, $+1$ o -1, en el caso de 2 clases):

```
octave:8> res.sv_coef
```

Ejercicio opcional (no entregable). Para la tarea `hart` de este ejemplo, representa gráficamente los vectores soporte obtenidos mediante la función `svmtrain`, superponiéndolos a las muestras de entrenamiento.

3.1.3. Clasificación y estimación de la tasa de acierto

A partir del modelo entrenado en la etapa anterior almacenado en la variable `res`, se pueden clasificar los vectores de un conjunto de validación o test. Primeramente, ese necesario cargar este conjunto:

```
octave:9> load data/hart/ts.dat ; load data/hart/tslabels.dat
```

La clasificación se realiza mediante la función `svmpredict`. Como en el caso de `svmtrain`, para ver todas las opciones de `svmpredict` basta invocar `svmpredict` sin argumentos. Tanto en `svmtrain` como en `svmpredict` podremos poner entre comillas simples cualquiera de las opciones que admite la librería `LibSVM`.

```
octave:10> svmpredict(yl,Y, res,' ');
Accuracy = 98.1% (981/1000) (classification)
```

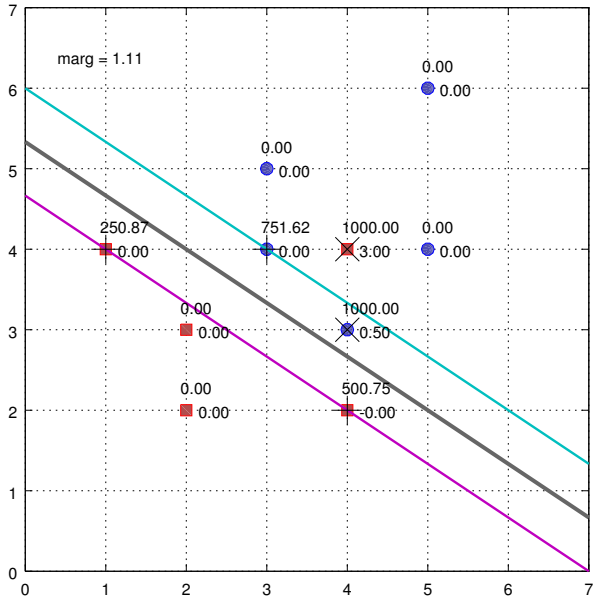
Ejercicio 1 (0.4 puntos). En el subdirectorio `data/mini` se encuentran dos pequeños conjuntos de datos de entrenamiento en dos dimensiones: `(trSep.dat, trSeplabels.dat)` y `(tr.dat, trlabels.dat)`. El primero es linealmente separable (no es necesario *kernel*) y el segundo no. Para cada uno de estos conjuntos[4]:

- Obtén los SVM sin kernel (es decir, kernel tipo lineal). Para simular la optimización estándar del caso separable basta usar un valor grande de $\mathcal{C}$ ($\mathcal{C} = 1000$).
- Determina a) los multiplicadores de Lagrange, α, asociados a cada dato de entrenamiento, b) los vectores soporte, c) el vector de pesos y umbral de la función discriminante lineal, y d) el margen correspondiente.
- Calcula los parámetros de la frontera lineal (recta) de separación;
- Representa gráficamente los vectores de entrenamiento, marcando los que son vectores soporte, y la recta separadora correspondiente.

Además, para el conjunto no-separable utilizando diversos valores relevantes de $\mathcal{C}$:

- Determina los valores de tolerancia de margen, ζ, asociados a cada dato de entrenamiento.
- Marca los vectores soporte "erróneos" en la representación gráfica.

Pista: Un ejemplo "ideal" de representación gráfica para el caso no-separable, obtenido con $\mathcal{C} = 1000$*, se muestra en la siguiente figura:*



[4]Te recomendamos que consultes el Apéndice E para un breve recordatorio sobre la teoría de SVM.

3.2. Tarea MNIST

Como se puede observar en la web de MNIST la aplicación de SVM a MNIST proporciona mejores resultados que la combinación PCA con un clasificador cuadrático.

Ejercicio 2 (0.4 puntos). Realiza un experimento donde se evalúe el error de clasificación en función de los parámetros del clasificador basado en SVM. Más concretamente, explora los valores del parámetro $\mathcal{C}$ (`-c` 1, 10, 100...) y el tipo de de kernel (`-t` 0, 1, 2, 3). Para aquellos tipos de kernel que lo permitan, explora sus parámetros específicos. Por ejemplo, en el caso del kernel polinomial (`-t` 1), explora el grado del polinomio (`-d` 1, 2, 3, 4, 5, ...); y en el caso de los kernels gaussiano y sigmoid (`-t` 2 y `-t` 3, respectivamente), explora el parámetro gamma (`-g` 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, ...).

Al igual que en el ejercicio 1 de mixturas de gaussianas, te recomendamos que elabores un script `pca+svm-exp.m` a partir del script `pca+mixgaussian-exp.m`, que realice la exploración de parámetros descrita utilizando un conjunto limitado valores de PCA ($D = 50, 100, 200$). Para acelerar la exploración de parámetros utiliza una partición entrenamiento-validación de un orden de magnitud inferior, es decir, 9 % para entrenamiento y 1 % para validación. En el caso de SVM, es imprescindible normalizar los datos dividiendo por su valor máximo.

En función del número de resultados obtenidos como consecuencia de la exploración de los valores de los parámetros, utiliza una representación adecuada de los mismos, ya sea gráfica o tabular, que muestre no solo el mejor resultado obtenido, sino también otros resultados relevantes que permitan poner de manifiesto la tendencia a mejorar o empeorar del modelo según varían los valores de los parámetros considerados.

Ejercicio 3 (0.2 puntos). Una vez determinado los valores óptimos de los parámetros del clasificador basado en SVM, entrena y evalúa un clasificador final en los conjuntos oficiales MNIST de entrenamiento y test, respectivamente. Para ello te recomendamos que tomes como punto de partida el script `pca+svm-exp.m`, modificándolo adecuadamente para generar el script `pca+svm-eva.m` que también deberá recibir como entrada el conjunto de test de MNIST. Recuerda que toda estimación de (la probabilidad de) error de un clasificador final, debe ir acompañada de sus correspondientes intervalos de confianza al 95 %. Discute los resultados obtenidos comparándolos con los obtenidos con el clasificador de mixtura de gaussianas y con los reportados en la tarea MNIST, especialmente los basados en SVM.

4. Redes Neuronales Multicapa

En esta última parte del proyecto trabajaremos con redes neuronales multicapa (MLP) utilizando la librería PyTorch[5] de Python que han introducido en las sesiones de teoría.

4.1. Tarea MNIST

Como se puede observar en la web de MNIST la utilización de redes neuronales puede llegar a proporcionar los mejores resultados en esta tarea. En PoliformaT os hemos dejado el fichero `mlp_exp.ipynb` que contiene un Jupyter Notebook con el código base para realizar un experimento en MNIST que, como en sesiones anteriores, utiliza un 90 % del conjunto de entrenamiento oficial para entrenamiento y el 10 % restante para validación. También dispones en PoliformaT del mismo código como fichero Python `mlp_exp.py`.

La arquitectura de la red MLP del código base está inspirada por una propuesta por Geoffrey E. Hinton de 2005 con 2 capas ocultas de 500 y 300 neuronas con función de activación ReLU, utilizando como función de pérdida la entropía cruzada y algoritmo de optimización Adam. La tasa de error en validación está alrededor de un 2 % cuando se entrena durante 20 épocas con un tamaño de batch de 100 muestras.

Antes de ejecutar el código base, se debe definir la variable de entorno:

```
export PYTHONPATH=$PYTHONPATH:$HOME/asigDSIC/ETSINF/apr/mlp/pylib
```

Después se puede ejecutar el código desde el Jupyter Notebook o directamente el código Python desde terminal. Comprueba que la tasa de error en validación es la esperada.

4.2. Código base

A continuación se explican los detalles del código utilizando el número de línea del fichero `mlp_exp.py`. Primeramente, en la línea 9 se carga el paquete `torch` que incluye la funcionalidad necesaria para diseñar, entrenar y clasificar utilizando redes neuronales; y en la línea 10 se importa el paquete `torchvision` que utilizaremos para cargar el conjunto de datos MNIST.

```
 9 import torch
10 import torchvision
```

El paquete `torchvision` posibilita utilizar otros conjuntos de datos bajo la clase `dataset`. El constructor de cada conjunto de datos permite indicar mediante su parámetros:

- `root`: el directorio donde se almacenará localmente el conjunto de datos creándolo si es necesario.

[5] pytorch.org

- `train`: seleccionar el conjunto de datos de entrenamiento `train=True` o de test `train=False`.
- `transform`: aplicar un preproceso a los datos.
- `download`: indicar si se debe descargar de Internet `download=True` si no está disponible en el directorio `root`.

En las líneas 21-33 se puede observar como se cargan los conjuntos oficiales de entrenamiento y test de MNIST, y se transforman a tensores.

```
21 # MNIST Dataset (Images and Labels)
22 training_dataset = torchvision.datasets.MNIST(
23     root='./data',
24     train=True,
25     transform=torchvision.transforms.ToTensor(),
26     download=True
27 )
28 test_dataset = torchvision.datasets.MNIST(
29     root='./data',
30     train=False,
31     transform=torchvision.transforms.ToTensor(),
32     download=True
33 )
```

Tras la carga de datos procederemos a realizar la partición del conjunto de entrenamiento oficial en conjunto de entrenamiento (90 %) y validación (10 %). Para ello, en la línea 42 utilizamos el paquete `torch.utils.data` cuyo método `random_split` realiza una partición aleatoria del tensor `training_dataset` en tantos conjuntos como dimensiones tenga `lenghts` y cuyo tamaño viene dado por el valor de la dimensión correspondiente.

```
42 train_dataset, val_dataset = \
     torch.utils.data.random_split(training_dataset, lengths=[54000, 6000])
```

En el mismo paquete, la clase `DataLoader` nos permite crear un objeto iterable a partir de nuestros datasets que está listo para ser utilizado en el entrenamiento de una red neuronal. Como se puede observar en las líneas 54-59 del código, organizamos el conjunto de datos en *batches* que agrupan 100 muestras (`batch_size = 100`) barajando los datos tras cada época para el conjunto de entrenamiento (`shuffle=True`), pero no para el conjunto de validación (`shuffle=False`).

```
54 batch_size = 100
55
56 # Preparamos los conjuntos de entrenamiento y validación en lotes y
57 # barajamos el conjunto de entrenamiento
```

```
58 train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
     batch_size=batch_size, shuffle=True)
59 val_loader = torch.utils.data.DataLoader(dataset=val_dataset,
     batch_size=batch_size, shuffle=False)
```

En las líneas 85-112 se define la clase `MLP` que nos permite a través de su constructor definir la arquitectura de una red *multilayer perceptron*, también conocida como red *feedforward*, en base a sus parámetros:

- `input_size`: dimensionalidad de los datos de entrada.
- `layers_data`: lista de capas como pares de número de neuronas y función de activación.
- `num_classes`: número de clases, es decir, número de neuronas en la capa de salida.

Adicionalmente, los parámetros `learning_rate` y `optimizer` caracterizan parcialmente el entrenamiento de la red MLP definida.

En la línea 93 se inicializan las capas de la red (`self.layers` como una lista de módulos. En las líneas 96-99 se recorre la lista de capas (número de neuronas - función de activación) para ir añadiendo cada capa a la lista de módulos como un módulo `nn.Linear` con el número de variables de entrada `input_size` y variables de salida `output_size` seguido de la función de activación `activation_function` como otro módulo. Finalmente, en la línea 101 se añade la capa de salida. Las líneas 102-106 definen atributos del entrenamiento de la red: cálculo en CPU o GPU, factor de aprendizaje, optimizador y criterio de entrenamiento.

```
 85 from torch import nn, optim
 86 from torch.nn.modules import Module
 87
 88 class MLP(nn.Module):
 89     # layers_data is a list of pairs: number of neurons and activation function
 90     def __init__(self, input_size, layers_data: list, num_classes,
          learning_rate=1e-3, optimizer=optim.Adam):
 91         super().__init__()
 92
 93         self.layers = nn.ModuleList()
 94         self.input_size = input_size
 95         # Layer and activation function are appended in a list
 96         for output_size, activation_function in layers_data:
 97             self.layers.append(nn.Linear(input_size, output_size))
 98             input_size = output_size
 99             self.layers.append(activation_function)
100         # Finally, the output layer is appended
101         self.layers.append(nn.Linear(input_size, num_classes))
102         self.device = torch.device('cuda' if torch.cuda.is_available()
```

```
                                                else 'cpu')
103         self.to(self.device)
104         self.learning_rate = learning_rate
105         self.optimizer = optimizer(params=self.parameters(), lr=learning_rate)
106         self.criterion = nn.CrossEntropyLoss()
```

La función `forward` (líneas 108-122) define como se calcula la salida de la red a partir de la entrada de datos al pasar por cada capa.

```
108     def forward(self, input_data):
109         for layer in self.layers:
110             output_data = layer(input_data)
111             input_data=output_data
112         return output_data
```

Tras haber definido la clase `MLP` se puede instanciar una arquitectura de red concreta. En las líneas 115-118, como se ha mencionado anteriormente, la red tiene 2 capas ocultas de 500 y 300 neuronas con función de activación ReLU utilizando el factor de aprendizaje (1*e*–3) y algoritmo de optimización (*Adam*) por defecto. La dimensionalidad de entrada y salida de la red corresponden con el tamaño de imagen y número de clases de MNIST.

```
115 input_size=28 * 28
116 M1, M2 = 500, 300
117 num_classes=10
118 mlp = MLP(input_size, [(M1, nn.ReLU()), (M2, nn.ReLU())], num_classes)
```

El entrenamiento de la red por un número de épocas está definido en las líneas 135-162. Como se puede observar en la línea 139, el entrenamiento se realiza iterando sobre `train_loader` que es un `DataLoader` que vuelve batches de entradas y etiquetas de clase. Estas entradas y etiquetas se pasan al dispositivo de cálculo en las líneas 142-143, y las imágenes de entrada se convierten de matriz a vector en la línea 146. Seguidamente, se realiza la inferencia en la línea 149, se calcula la diferencia entre la salida esperada y la estimada como la función de pérdida de la línea 151, lo cual nos permite calcular el gradiente en la línea 153 y actualizar los valores de los parámetros en la línea 155. Finalmente, se resetean los gradientes a cero y se acumula la función de pérdida en las líneas 157 y 160, respectivamente.

```
135 for epoch in range(20):
136     total_loss = 0.0
137
138     # Loop over the training set in batch mode
139     for (inputs, labels) in train_loader:
140
141         # Transfering data to GPU or CPU
142         inputs = inputs.to(mlp.device)
```

```
143         labels = labels.to(mlp.device)
144
145         # Converting from 28x28 data samples to 768 data samples
146         inputs = inputs.view(-1, 28*28)
147
148         # Forward pass
149         outputs = mlp(inputs)
150         # Computing loss function
151         loss = mlp.criterion(outputs, labels)
152         # Computing gradient
153         loss.backward()
154         # Updating parameter values with gradient
155         mlp.optimizer.step()
156         # Reset gradient
157         mlp.optimizer.zero_grad()
158
159         # Accumulated loss function over an epoch
160         total_loss += loss.item()
161
162     print("Epoch %d, Loss=%.4f" % (epoch+1, total_loss/len(train_loader)))
```

Por último, la función `error` (líneas 168-183) estima el error empírico del modelo `model` sobre el conjunto de datos almacenado en `data_loader` con el dispositivo de cómputo `device`. En la línea 172 se itera sobre el conjunto de datos realizando un proceso similar al entrenamiento hasta la línea 176. Sin embargo, la línea 177 obtiene en la variable `predicted` como el *argmax* de los valores de la capa salida ignorando el valor máximo. En la línea 179 se acumula el número de errores en cada batch comparando las etiquetas que predice el modelo con las reales para ver cuando difieren.

```
168 def error(model, data_loader, device):
169     with torch.no_grad():
170         errors = 0
171         total = 0
172         for inputs, labels in data_loader:
173             inputs = inputs.to(device)
174             inputs = inputs.view(-1, 28*28)
175
176             outputs = model(inputs)
177             _, predicted = outputs.max(1)
178
179             errors += (predicted.cpu() != labels).sum().item()
180             total += labels.size(0)
181
182     err = errors / total
```

```
183     return err
```

Las líneas 188 y 194 calculan el error en el conjunto entrenamiento y validación tras entrenar el modelo.

```
188 err=error(mlp, train_loader, mlp.device)
189 print("Tasa de error en entrenamiento: %.2f%%" % (err*100))
190
[...]
193
194 err=error(mlp, val_loader, mlp.device)
195 print("Tasa de error en validación: %.2f%%" % (err*100))
```

Ejercicio 1 (0.25 puntos). A partir del código base proporcionado, representa gráficamente la evolución del error en entrenamiento y validación en función del número de épocas. Si es necesario, incrementa el número máximo de iteraciones para poder observar el fenómeno de sobreentrenamiento.

Ejercicio 2 (0.5 puntos). Para ajustar los parámetros de la red MLP, estudia el comportamiento de la tasa de error en el conjunto de validación en función de:

- Algoritmo de optimización: SGD, Adadelta, Adagrad, Adam, etc.
- Función de activación: ReLU, Sigmoid, Tanh, etc.
- Número de capas ocultas: 1, 2, 3, etc.
- Número de neuronas por capa, por simplicidad, en múltiplos de 100 hasta 800.
- Criterio de parada: máximo de iteraciones, variación de error en validación, etc.

En función del número de resultados obtenidos como consecuencia de la exploración de los valores de los parámetros, utiliza una representación adecuada de los mismos, ya sea gráfica o tabular, que muestre no solo el mejor resultado obtenido, sino también otros resultados relevantes que permitan poner de manifiesto la tendencia a mejorar o empeorar del modelo según varían los valores de los parámetros considerados.

Ejercicio 3 (0.25 puntos). Tras el ajuste de parámetros en el conjunto de validación, utiliza los valores óptimos de los parámetros del clasificador para entrenar y evaluar un clasificador final en los conjuntos oficiales MNIST de entrenamiento y test, respectivamente. Recuerda que toda estimación de (la probabilidad de) error de un clasificador final, debe ir acompañada de sus correspondientes intervalos de confianza al 95 %. Discute los resultados obtenidos comparándolos con los obtenidos en los clasificadores estudiados y con otros clasificadores basados en redes neuronales reportados en la tarea MNIST.

A. Tutorial sobre Octave

A.1. Introducción

Varias de las técnicas empleadas en Aprendizaje Automático y Reconocimiento de Formas (RF) emplean cálculos vectoriales y matriciales, como por ejemplo en las implementaciones de clasificadores basados en la distribución gaussiana. Por tanto, una herramienta que implemente de forma sencilla estos cálculos matriciales puede simplificar notáblemente la implementación de sistemas de RF.

Una de las herramientas comerciales más potentes en cálculo matricial es MATLAB. Asimismo existe una herramienta de código libre que presenta capacidades semejantes: *GNU Octave*.

GNU Octave es un lenguaje de alto nivel interpretado definido inicialmente para computación numérica. Entre otras, posee capacidades de cálculo numérico para solucionar problemas lineales y no lineales. También dispone de herramientas gráficas para visualizar datos y resultados. Se puede usar de forma interactiva y/o programada mediante *scripts* en un lenguaje interpretado. La sintaxis y semántica de Octave es prácticamente idéntica a MATLAB, lo que hace que los programas sean fácilmente portables entre ambas plataformas.

Octave está en continuo crecimiento y puede descargarse y consultarse su documentación y estado en su web `http://www.gnu.org/software/octave/`. Aunque está definido para funcionar en GNU/Linux, también es portable a otras plataformas como OS X y MS-Windows (los detalles pueden consultarse en la web mencionada).

A.2. Órdenes básicas de *Octave*

Octave puede ejecutarse desde la línea de órdenes o desde el menú de aplicaciones del entorno gráfico. Para ejecutar desde la línea de órdenes se abre un terminal y se escribe:

```
octave
```

Generalmente, obtenemos una salida similar a:

```
GNU Octave, version 3.8.2
Copyright (C) 2014 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-redhat-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
```

```
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1>
```

La última línea tendrá un cursor indicando que se esperan órdenes de Octave. Estamos pues en el modo **interactivo**.

A.2.1. Aritmética básica

Octave acepta a partir de este momento expresiones aritméticas sencillas (operadores `+`, `-`, `*`, `/` y `^`, este último para exponenciación), funciones trigonométricas (`sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`), logaritmos (`log`, `log10`), exponencial neperiana (`e^n` ó `exp(n)`) y valor absoluto (`abs`). Como respuesta a estas expresiones Octave da valor a la variable predefinida `ans`, y la muestra. Pero los resultados también pueden asignarse a otras variables. Por ejemplo:

```
octave:1> sin(1.71)
ans =  0.99033
octave:2> b=sin(2.16)
b =  0.83138
```

Para consultar el valor de una variable, basta con escribir su nombre, aunque también puede emplearse la función `disp`, que muestra el contenido de la variable omitiendo su nombre:

```
octave:3> b
b =  0.83138

octave:4> ans
ans =  0.99033

octave:5> disp(b)
 0.83138
```

Las variables pueden usarse en otras expresiones:

```
octave:6> c=b*ans
c =  0.82334
```

Puede evitarse que se muestre el resultado de cada operación añadiendo (`;`) al final de la operación:

```
octave:7> d=ans*b*5;
octave:8> disp(d)
 4.1167
```

A.2.2. Operadores básicos en vectores y matrices

Para la notación matricial en Octave se usan los corchetes (`[ ]`); en su interior, las filas se separan por punto y coma (`;`) y las columnas por espacios en blanco ( ) o por comas (`,`). Por ejemplo, para crear un vector fila de dimensión 3, un vector columna de dimensión 2 y una matriz de 3×2, se puede hacer:

```
octave:9> v1=[1 3 -5]
v1 =
   1   3  -5

octave:10> v2=[4;2]
v2 =
   4
   2

octave:11> m=[3,-4;2 1;-5 0]
m =
   3  -4
   2   1
  -5   0
```

Siempre y cuando las dimensiones de los elementos vectoriales y matriciales implicados sean apropiados, sobre ellos se pueden aplicar operadores de suma (`+`), diferencia (`-`) o producto (`*`). El operador potencia (`^`) puede aplicarse sobre matrices cuadradas. Los operadores producto, división y potencia tienen la versión "elemento a elemento" (`.*`, `./`, `.^`). Por ejemplo:

```
octave:12> mv=v1*m
mv =
   34   -1

octave:13> mx=m.*5
mx =
   15  -20
   10    5
  -25    0

octave:14> v3=v1*v2
error: operator *: nonconformant arguments (op1 is 1x3, op2 is 2x1)

octave:15> v3=v1*[3;5;6]
v3 = -12

octave:16> mvv=[3;5;6]*v1
```

```
mvv =
    3    9  -15
    5   15  -25
    6   18  -30
```

Los operadores de comparación (`>`, `<`, `>=`, `<=`, `==`, `!=`) se pueden aplicar "elemento a elemento". Como resultado se obtiene una matriz binaria con 1 en las posiciones en las que se cumple la condición y 0 en caso contrario:

```
octave:17> m>=0
ans =
   1   0
   1   1
   0   1

octave:18> m!=0
ans =
   1   1
   1   1
   1   0
```

Esos operadores se pueden emplear en la comparación de vectores y matrices de dimensiones congruentes. La matriz binaria resultante contiene los resultados de las comparaciones de los pares de elementos en la misma posición en ambas estructuras.

Para tranponer una matriz o vector se usa el operador de transposición (`'`):

```
octave:19> m2=m'
m2 =
   3   2  -5
  -4   1   0
```

El indexado de los elementos se hace entre paréntesis. Para vectores puede indicarse una posición o lista de posiciones, mientras que para una matriz se espera una fila o secuencia de filas seguida de una columna o secuencia de columnas:

```
octave:20> v1(2)
ans =  3

octave:21> v1([2 3])
ans =
   3  -5

octave:22> v2(2)
ans =  2
```

```
octave:23> m3=[1 2 3 4;5 6 7 8;9 10 11 12]
m3 =
    1    2    3    4
    5    6    7    8
    9   10   11   12

octave:24> m3([1 3], [1 4])
ans =
    1    4
    9   12
```

Para indicar todas las filas o columnas, se puede emplear (`:`):

```
octave:25> m3(:,[1 3])
ans =
    1    3
    5    7
    9   11
```

Los rangos se denotan como (`i:f`), donde `i` es el índice inicial y `f` el final. Se puede emplear la notación (`i:inc:f`), donde `inc` indica el incremento, que por omisión es 1.

```
octave:26> m3([1 3],1:3)
ans =
    1    2    3
    9   10   11

octave:27> m3([1 3],1:2:4)
ans =
    1    3
    9   11
```

Para indicar el último índice de una dimensión se puede emplear (`end`):

```
octave:28> m3([1 3],end)
ans =
    4
   12
```

A.2.3. Funciones básicas en vectores y matrices

Octave aporta múltiples funciones para operar con vectores y matrices. Las más importantes son:

- `size(m)`: devuelve número de filas y columnas de la matriz (en el caso de un vector, una de las dimensiones tendrá tamaño 1)

- `eye(f,c)`, `ones(f,c)`, `zeros(f,c)`: dan la matriz identidad, todo unos y nula, respectivamente, de tamaño `f`×`c`; si se pone un solo número, da la matriz cuadrada correspondiente
- `sum(v)`, `sum(m)`: da la suma de los elementos del vector o matriz; en el caso de la matriz, devuelve el vector resultante de las sumas por columnas; si se le pasa un segundo argumento (`sum(m,n)`), éste indica la dimensión a sumar (1 para columnas, 2 para filas).
- `max(v)`, `max(m)`: indica el valor máximo del vector o el vector con los máximos por columna de la matriz; si se pide que devuelva dos resultados (`[r1,r2]=max(v)`, `[r1,r2]=max(m)`), el primer resultado almacena los valores y el segundo su posición
- `det(m)`: determinante de `m`
- `eig(m)`: vector de valores propios de `m` o su versión matricial diagonal
- `diag(v)`: crear matriz diagonal con los valores de `v`
- `inv(m)`: inversa de la matriz `m` si esta es no singular
- `trace(m)`: traza de la matriz `m`
- `sort(v)`: vector ordenado con los valores del vector `v`
- `repmat(m,f,c)`: crea una matriz de `f`×`c` bloques de `m`; si `c` se omite, será de `f`×`f`
- `find(v)`, `find(m)`: se le pasa un vector o matriz e indica aquellos elementos que no son cero (índices absolutos empezando en 1 y haciendo el recorrido por cada columna y por filas ascendentes); si se le piden dos resultados (`[r1,r2]=find(v)`, `[r1,r2]=find(m)`) el primer resultado almacena fila y el segundo columna; se puede aplicar sobre resultados de operaciones lógicas a fin de verificar elementos de la matriz o vector que cumplen una condición. Por ejemplo, mediante la función (`rem`) que obtiene el resto del primer operador dividido por el segundo, se pueden obtener las posiciones de los elementos pares:

```
octave:29> [r,c]=find(rem(m3,2)==0)
r =
   1
   2
   3
   1
   2
   3

c =
   2
```

```
2
2
4
4
4
```

A.2.4. Carga y salvado de datos

La introducción de datos de forma manual no es apropiada para grandes cantidades de datos. Por tanto, Octave aporta funciones que permiten cargar de y salvar en ficheros. El salvado se hace mediante la orden `save`:

```
octave:30> save "m3.dat" m3
```

El fichero tiene el siguiente contenido:

```
# Created by Octave 3.0.5, DATE <user@machine>
# name: m3
# type: matrix
# rows: 3
# columns: 4
 1 2 3 4
 5 6 7 8
 9 10 11 12
```

Los ficheros de datos a cargar deben seguir este formato, indicando en la línea "`# name:`" el nombre de la variable en la que se cargarán los datos. Por ejemplo, ante un fichero `maux.dat` cuyo contenido es:

```
# Created by Octave 3.0.5, DATE <user@machine>
# name: A
# type: matrix
# rows: 4
# columns: 3
 1 2 -3
 5 -6 7
 -9 10 11
 -5 2 -1
```

Su carga definirá la matriz (`A`) como:

```
octave:31> load "maux.dat"

octave:32> A
A =
```

```
    1    2   -3
    5   -6    7
   -9   10   11
   -5    2   -1
```

La orden `save` puede usarse con opciones como `-text` (grabar en formato texto con cabecera, por omisión), `-ascii` (graba en formato texto sin cabecera), `-z` (graba en formato comprimido), o `-binary` (graba en binario). Por ejemplo:

```
octave:33> save -ascii "m3woh.dat" m3
```

En ocasiones, el salvado de datos puede provocar pérdida de precisión, pues por omisión se salva hasta el cuarto decimal. Para modificar esta precisión de salvado, se puede emplear `save_precision(n)`, donde `n` es el número de posiciones decimales (incluyendo el `.`) que se grabarán.

A.2.5. Funciones Octave

En Octave se pueden definir funciones que hagan tareas específicas y/o complejas. Las funciones Octave pueden recibir varios parámetros y pueden devolver varios valores de retorno (que pueden incluir vectores y matrices). La sintaxis básica es:

```
function [ lista_valores_retorno ] = nombre ( [ lista_parametros ] )
  cuerpo
end
```

Porejemplo:

```
octave:34> function [m1,m2] = addsub(ma,mb)
> m1=ma+mb
> m2=ma-mb
> end

octave:35> mat1=[1,2;3,4]
mat1 =
   1   2
   3   4

octave:36> mat2=[-1,2;3,-4]
mat2 =
  -1   2
   3  -4

octave:37> addsub(mat1,mat2)
m1 =
   0   4
```

```
   6   0

m2 =
   2   0
   0   8

ans =
   0   4
   6   0

octave:38> [mr1,mr2]=addsub(mat1,mat2)
m1 =
   0   4
   6   0

m2 =
   2   0
   0   8


mr1 =
   0   4
   6   0

mr2 =
   2   0
   0   8
```

Es habitual definir las funciones en ficheros de código Octave cuyo nombre debe ser el mismo que la función con el sufijo "`.m`" (en nuestro ejemplo sería `addsub.m`). Estos ficheros deben situarse en el mismo directorio en el que se ejecuta Octave. De esa forma, se puede acceder a las funciones sin tener que definirlas cada vez.

A.2.6. Programas Octave

Octave se puede usar de forma *no interactiva* mediante *scripts* que son interpretados por Octave. En estos *scripts* o *"programas Octave"* se pueden emplear las mismas instrucciones que en el modo interactivo. Por ejemplo, suponiendo que tenemos en el directorio actual el fichero `addsub.m` con la función previamente definida, desde cualquier terminal podemos crear (con algún editor) el fichero `test.m` con el siguiente contenido:

```
#!/usr/bin/octave -qf
a=[1,2,3;4,5,6;7,8,9]
b=[9,8,7;6,5,4;3,2,1]
c=a+b
```

```
[d,e]=addsub(a,b)
disp(c)
```

Si desde la línea de órdenes le damos permisos de ejecución (`chmod +x test.m`), podremos ejecutarlo como cualquier programa ejecutable o *shell script*:

```
$_ ./test.m
a =
   1   2   3
   4   5   6
   7   8   9

b =
   9   8   7
   6   5   4
   3   2   1

c =
   10   10   10
   10   10   10
   10   10   10

m1 =
   10   10   10
   10   10   10
   10   10   10

m2 =
  -8  -6  -4
  -2   0   2
   4   6   8

d =
   10   10   10
   10   10   10
   10   10   10

e =
  -8  -6  -4
  -2   0   2
   4   6   8

 10   10   10
 10   10   10
```

```
10   10   10
```

Estos programas también pueden ejecutarse desde la línea interactiva de Octave escribiendo su nombre (sin el sugijo "`.m`"). En este caso, se pueden poner argumentos en la línea de órdenes y puede usarse la variable `nargin` (número de argumentos) y la función `argv()` (da una lista de los valores alfanuméricos de los argumentos recibidos). Estos argumentos están en formato de cadena de caracteres; por tanto los valores numéricos deben convertirse a formato numérico, por ejemplo empleando la función `str2num(c)`.

A.3. Ejercicios propuestos

1. Realiza el producto escalar de los vectores $v_1 = (1, 3, 8, 9)$ y $v_2 = (-1, 8, 2, -3)$.

2. *a*) Obtén la matriz de dimensión 4×4 mediante producto de los vectores v_1 y v_2
   b) Calcula su determinante
   c) Calcula sus valores propios.

3. Sobre la matriz del ejercicio 2:
   a) Calcula su submatriz 2×2 formadas por las filas 1 y 3 y las columnas 2 y 3.
   b) Súmale la matriz todo unos a la submatriz resultante.
   c) Calcula el determinante de la matriz resultante.
   d) Calcula la inversa de la matriz resultante.

4. Sobre la matriz inversa resultado del ejercicio 3:
   a) Calcula su máximo valor y su posición.
   b) Calcula las posiciones (fila y columna) de los elementos mayores que 0.
   c) Calcula la suma de cada columna.
   d) Calcula la suma de cada fila.

5. Salva la matriz inversa resultante del ejercicio 3 con hasta 7 dígitos decimales; comprueba que se ha salvado correctamente.

6. Define una función Octave que reciba una matriz y devuelva la primera fila y la primera columna de esa matriz.

7. Implementa un *script* Octave que lea una matriz de un fichero `data` y grabe su traspuesta en el fichero `data_trans`.

B. Tarea de clasificación: MNIST

B.1. Introducción

La base de datos MNIST[6] 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 en formato Octave[7]:

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

En la Figura 1 se muestra una representación de un dígito cero que se corresponde con la segunda fila de datos del fichero `train-images-idx3-ubyte.mat.gz` que ha sido 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. Asimismo, 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 aquellos que desean 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.

B.2. Carga de datos

Los ficheros Octave mencionados anteriormente son ficheros *ascii* comprimidos en formato Octave y están disponibles en PoliformaT.

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

[7] Estos ficheros se generan al ejecutar el bash script `00-preprocess.sh` disponible en PoliformaT.

```
  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  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   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  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   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  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   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 cero manuscrito de MNIST.

Si examinamos `train-images-idx3-ubyte.mat.gz` desde el intérprete de comando (por ejemplo con `zless`) veremos que contiene una cabecera como esta:

```
# name: X
# type: matrix
# rows: 60000
# columns: 784
```

Donde `#name` indica el nombre de la variable (matriz Octave) en la que se cargarán los datos, `#type` es el tipo de variable, `#rows` es el número de filas y finalmente `#columns` es el número de columnas. Por lo tanto el fichero `train-images-idx3-ubyte.mat.gz` contiene una matriz representando 60.000 imágenes por filas y 784 dimensiones por columnas.

En Octave cargaremos esta matriz con la siguiente orden:

```
octave:1> load train-images-idx3-ubyte.mat.gz
```

Comprobaremos que la variable X se ha cargado, así como su tamaño:

```
octave:2> size(X)
ans =

   60000     784
```

El fichero de datos de test `t10k-images-idx3-ubyte.mat.gz` cuando se cargue se instanciará en la variable `Y`.

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.gz` y `t10k-labels-idx1-ubyte.mat.gz`, respectivamente. Cargad estos ficheros para comprobar las dimensiones de las variables.

B.3. Visualización de dígitos

Podemos visualizar la imagen de la Fig. 1 que se corresponde con la fila 2 de la variable `X` teniendo en cuenta que es una imagen de 28 x 28 píxeles:

```
octave:3> x=X(2,:);
octave:4> xr=reshape(x,28,28);
octave:5> imshow((255-xr)',[])
```

También podemos visualizar las 20 primeras imágenes de la base de datos MNIST para hacernos una idea de la dificultad de esta tarea real:

```
octave:6> for i=1:20
> xr=reshape(X(i,:),28,28); imshow((255-xr)',[]); pause(1);
> end
```

C. Clasificador gaussiano

En la asignatura de Percepción se implementó un clasificador gaussiano cuya teoría e implementación refrescamos en esta sección. El código está disponible en PoliformaT.

C.1. Estimación de parámetros y clasificación

Antes de abordar la implementación del clasificador gaussiano es necesario recordar que este clasificador es una instanciación del clasificador de Bayes:

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

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

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

Por tanto,

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

Si lo expresamos en términos de función discriminante cuadrática con $\boldsymbol{x}$:

$$c^*(\boldsymbol{x}) = \underset{c=1,\dots,C}{\operatorname{argmax}}\ g_c(\boldsymbol{x}) = \underset{c=1,\dots,C}{\operatorname{argmax}}\ \log P(c) + \boldsymbol{x}^t\, W_c\, \boldsymbol{x} + \boldsymbol{w}_c^t\, \boldsymbol{x} + w_{c0}$$

donde

$$W_c = -\frac{1}{2}\Sigma_c^{-1} \tag{10}$$

$$\boldsymbol{w}_c = \Sigma_c^{-1}\boldsymbol{\mu}_c \tag{11}$$

$$w_{c0} = -\frac{1}{2}\log |\Sigma_c| - \frac{1}{2}\boldsymbol{\mu}_c^t\Sigma_c^{-1}\boldsymbol{\mu}_c \tag{12}$$

Para simplificar esta implementación hemos dejado el término $\log P(c)$ fuera de la componente constante de la probabilidad condicional gaussiana. La estimación máximo verosímil de los parámetros del clasificador gaussiano es ampliamente conocida:

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

C.2. Implementación y experimentación

A continuación se muestra una implementación en Octave del clasificador gaussiano `gaussian.m` disponible en PoliformaT. Primero, veremos el código correspondiente a la estimación de parámetros:

```
 8 function [errY] = gaussian(X,xl,Y,yl,alphas)
 9
10 classes=unique(xl);
11 N=rows(X);
12 M=rows(Y);
13 D=columns(X);
14
15 % Parameter estimation
16 for c=classes'
17   ic=find(c==classes);
18   idx=find(xl==c);
19   Xc=X(idx,:);
20   Nc=rows(Xc);
21   pc(ic)=Nc/N;
22   muc=sum(Xc)/Nc;
23   mu(:,ic)=muc';
24   sigma{ic}=((Xc-muc)'*(Xc-muc))/Nc;
25 end
```

Las líneas 10-13 obtienen el vector de etiquetas de clases `[0 1 ... 9]`, el número de muestras de entrenamiento y evaluación, y la dimensionalidad de los datos.

A continuación sigue el bucle (líneas 15-25) donde se estimarán los parámetros de cada clase, teniendo en cuenta que `ic` es el índice (o posición) de la clase `c` dentro del vector `classes` y `idx` es el vector de índices a muestras de la clase `c` sobre las filas de la matriz `X`. Recuerda que `ic` es un índice que tomará valores en `[1 2 ... 10]`, mientras `c` es una etiqueta de clase en el rango de valores `[0 1 ... 9]`.

En las líneas 20-22 se estima la probabilidad a priori y media de la clase `c`. El vector de probabilidades a priori `pc` será un vector fila, y la matriz de medias `mu` dispone la media de cada clase como un vector columna.

En la línea 24 se estima la matriz de covarianzas de la clase `c` y se almacena en la posición `ic` de un vector de celdas. Estos vectores de celdas permiten almacenar tipos diferentes en el mismo vector, pero en nuestro caso nos será útil para crear un vector de matrices de covarianzas e indexarlo fácilmente.

Las líneas 27-43 de la función `gaussian` contienen el bucle que realizará el cálculo del error en el conjunto de evaluación para los valores de α de suavizado que contiene el vector `alphas`.

```
27 for i=1:length(alphas)
28
29   % Smoothing with identity matrix
30   for c=classes'
31     ic=find(c==classes);
32     ssigma{ic}=alphas(i)*sigma{ic}+(1-alphas(i))*eye(D);
33   end
34
35   % Compute g for each sample in the evaluation set
36   for c=classes'
37     ic=find(c==classes);
38     gY(:,ic)=log(pc(ic))+compute_pxGc(mu(:,ic),ssigma{ic},Y);
39   end
40
41   [~,idY]=max(gY');
42   errY(i)=mean(classes(idY)!=yl)*100;
43 end
44
45 end
```

Primeramente, las líneas 29-33 implementan el suavizado *flat smoothing* de la matriz de covarianzas de cada clase con la matriz identidad:

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

Seguidamente, las líneas 35-39 muestran el código que estima la función discriminante de cada clase calculando $\log P(c) + \log p(\boldsymbol{x} \mid c)$ para todas las muestras del conjunto de evaluación. Observa como la función auxiliar `compute_pxGc` implementa $\log p(\boldsymbol{x} \mid c)$ estimando para cada muestra del conjunto de evaluación la log probabilidad condicional modelizada mediante una distribución gaussiana. Por último, la línea 41 obtiene el índice de la clase `[1 2 ... 10]` cuyo valor de su función discriminante es máximo para cada muestra del conjunto de validación. La línea 42 es el cálculo de la probabilidad de error empírica, es decir, el número de errores promedio sobre el conjunto de evaluación. Observa como la indexación del vector `classes` mediante el vector de índices `idy`, es

decir, `classes(idy)`, realiza la conversión de índices de clases `[1 2 ... 10]` a etiquetas de clase `[0 1 ... 9]`.

A continuación se muestra la implementación de la función auxiliar `compute_pxGc`, invocada en la línea 38:

```
47 % Computes component-and-class conditional gaussian prob
48 function [pxGc] = compute_pxGc(mu,sigma,X)
49   I=pinv(sigma);
50   qua=-0.5*sum((X*I).*X,2);
51   lin=X*I*mu;
52   cons=-0.5*logdet(sigma);
53   cons=cons-0.5*mu'*I*mu;
54   pxGc=qua+lin+cons;
55 end
```

Esta función calcula la log probabilidad condicional para la gaussiana con media `mu` y matriz de covarianzas `sigma` de cada muestra en `X`. La línea 49 precalcula la pseudoinversa de la matriz de covarianzas que se usará posteriormente. La línea 50 se corresponde con el término cuadrático que pre y posmultiplica `X` en la Ec. 10, la línea 51 implementa el término lineal, que multiplica `X` en la Ec. 11, y las líneas 52 y 53 implementan el término constante en la Ec. 12.

Como puedes observar, para calcular la inversa de la matriz de covarianzas utilizamos la función pseudoinversa `pinv` en lugar de la inversa convencional `inv`. Esto se debe a que no se puede calcular la inversa de una matriz cuyo rango no es completo, pero sí la pseudoinversa. Es decir, nuestra matriz de covarianzas para MNIST es singular, y por tanto tiene filas (o columnas) que son linealmente dependientes unas de otras. Puedes comprobarlo rápidamente ejecutando `rank(cov(X,1))` y viendo que el rango es inferior a la dimensionalidad de los datos $D = 784$. Esto se debe mayormente a que tenemos muchas dimensiones que son siempre cero, al ser parte del fondo sobre el que está centrado el dígito.

Como consecuencia de que la matriz de covarianzas sea singular, su determinante, que se calcula en la línea 52, será cero y su logaritmo `-Inf` imposibilitando el cálculo de esta log probabilidad. Por ello, es necesario una versión robusta del cálculo del logaritmo del determinante de la matrix de covarianzas que se muestra a continuación:

```
63 function v = logdet(X)
64   lambda = eig(X);
65   if any(lambda<=0)
66     v=log(realmin);
67   else
68     v=sum(log(lambda));
69   end
70 end
```

La función `logdet` reemplaza el logaritmo de cero por el logaritmo de la constante `realmin` en Octave. Además, podrás comprobar que la función `logdet` no calcu-

la `log(det(sigma))` directamente, sino que para evitar valores excesivamente grandes (`Inf`) del determinante, este cálculo se realiza de forma robusta como la suma de logaritmos de los valores propios de la matriz de covarianzas[8].

El clasificador gaussiano se puede invocar directamente desde Octave habiendo cargado previamente los ficheros de datos (ver Apéndice B), pero lo más funcional es hacerlo desde un script Octave. Por ejemplo, el siguiente script, `gaussian-exp.m` disponible en PoliformaT implementa un barrido del parámetro de suavizado α y la correspondiente evaluación de la tasa de error sobre un conjunto de evaluación:

```
 1 #!/usr/bin/octave -qf
 2
 3 if (nargin!=5)
 4 printf("Usage: gaussian-exp.m <trdata> <trlabels> <alphas> <%%trper>...
 5 exit(1);
 6 end;
 7
 8 arg_list=argv();
 9 trdata=arg_list{1};
10 trlabs=arg_list{2};
11 alphas=str2num(arg_list{3});
12 trper=str2num(arg_list{4});
13 dvper=str2num(arg_list{5});
14
15 load(trdata);
16 load(trlabs);
17
18 N=rows(X);
19 seed=23; rand("seed",seed); permutation=randperm(N);
20 X=X(permutation,:); xl=xl(permutation,:);
21
22 Ntr=round(trper/100*N);
23 Ndv=round(dvper/100*N);
24 Xtr=X(1:Ntr,:); xltr=xl(1:Ntr);
25 Xdv=X(N-Ndv+1:N,:); xldv=xl(N-Ndv+1:N);
26
27 [edv] = gaussian(Xtr,xltr,Xdv,xldv,alphas);
28
29 printf("\n  alpha dv-err");
30 printf("\n------- ------\n");
31
32 for i=1:length(alphas)
33   printf("%.1e %6.3f\n",alphas(i),edv(i));
34 end
```

[8] https://www.adelaide.edu.au/mathslearning/play/seminars/evalue-magic-tricks-handout.pdf

Las líneas 3-6 comprueban que el número de parámetros sea el correcto, para después parsear esos parámetros:

- `trdata`: fichero de imágenes.
- `trlabs`: fichero de etiquetas de clase.
- `alphas`: vector con el rango de valores de suaviado α a evaluar.
- `trper`: Porcentaje del conjunto de imágenes dedicadas a entrenamiento.
- `dvper`: Porcentaje del conjunto de imágenes dedicadas a validación.

Las líneas 15 y 16 realizan la carga de los ficheros de imágenes y etiquetas de clase, respectivamente. Las líneas 18-20 se emplean para barajar aleatoriamente las imágenes con un semilla predefinida (valor `23`) que garantiza que el barajado, aunque aleatorio, será siempre el mismo con esa semilla. Las líneas 22-25 realizan la partición en entrenamiento y validación, dedicando el `trper` % de las muestras desde el principio del fichero a entrenamiento, y el `dvper` % desde el final del fichero a validación. Seguidamente, en la línea 27 se invoca a la función `gaussian` con los conjuntos de entrenamiento y validación, y el vector de valores de α. Para cada uno de estos valores de α se devuelve la tasa de error del clasificador gaussiano suavizado con ese valor de α calculada en el conjunto de validación. Finalmente, las líneas 29-34 imprimen la tasa de error en validación para cada valor de α.

La ejecución del script desde línea de comandos del terminal:

```
./gaussian-exp.m train-images-idx3-ubyte.mat.gz
train-labels-idx1-ubyte.mat.gz
"[1e-8 1e-7 1e-6 1e-5 1e-4 1e-3 1e-2 1e-1 2e-1 5e-1 9e-1 1e1]" 90 10
```

obtiene como resultado:

```
  alpha dv-err
------- ------
1.0e-08 19.550
1.0e-07 18.933
1.0e-06 14.083
1.0e-05  6.317
1.0e-04  4.267
1.0e-03  6.383
1.0e-02 10.000
1.0e-01 11.967
2.0e-01 12.200
5.0e-01 13.550
9.0e-01 18.683
1.0e+01 93.867
```

Como se puede observar, la mejor tasa de error se consigue con $\alpha = 10^{-4}$. La evaluación final con el valor de α óptimo entrenando con todo el conjunto de entrenamiento y evaluando en el conjunto de test se puede realizar con el script `gaussian-eva.m`:

```
 1 #!/usr/bin/octave -qf
 2
 3 if (nargin!=5)
 4 printf("Usage: gaussian-eva.m <trdata> <trlabels> <tedata> <telabels>...
 5 exit(1);
 6 end;
 7
 8 arg_list=argv();
 9 trdata=arg_list{1};
10 trlabs=arg_list{2};
11 tedata=arg_list{3};
12 telabs=arg_list{4};
13 alpha=str2num(arg_list{5});
14
15 % Loading data
16 load(trdata);
17 load(trlabs);
18 load(tedata);
19 load(telabs);
20
21 [ete] = gaussian(X,xl,Y,yl,alpha);
22
23 printf("\n  alpha te-err");
24 printf("\n------- ------\n");
25 printf("%.1e %6.3f\n",alpha,ete);
```

Si ejecutamos este script desde la línea de comandos del terminal:

```
./gaussian-eva.m train-images-idx3-ubyte.mat.gz train-labels-idx1-ubyte.mat.gz
t10k-images-idx3-ubyte.mat.gz t10k-labels-idx1-ubyte.mat.gz 1e-4
```

obtenemos la tasa de error en el conjunto de test:

```
  alpha te-err
------- ------
1.0e-04  4.180
```

Esta última tasa de error es la que se puede comparar con las reportadas en la web de MNIST[9].

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

D. Función `plot` en Octave

La función `plot`[10] permite la representación gráfica bidimensional de secuencias de puntos definidas por sus coordenadas `x` e `y` como vectores:

```
plot(x, y, format, ...)
```

donde `format` son los argumentos de formato que se define por una cadena con cuatro partes opcionales `<linestyle><marker><color><;displayname;>` que indican el tipo de línea ('-' continua, '–' discontinua, ':' punteada, etc.), el tipo de punto ('+' cruz, 'o' círculo, 's' cuadrado, etc.), el color ('r' rojo, 'g' verde, 'b' azul, etc.) y la etiqueta de la leyenda, respectivamente. Por ejemplo:

```
plot (tr(:,1),tr(:,2),"sr")
```

que representa gráficamente el conjunto de entrenamiento como cuadrados rojos. También cabe la posibilidad de añadir a la función `plot` modificadores de propiedades:

```
plot(x, y, format, property, value, ...)

plot(x, y, property, value, ...)
```

donde algunos valores útiles en esta práctica para `property` son `markersize`, `linewidth`, `markerfacecolor`, `color`, etc., y cuyos valores (`value`) se pueden consultar en el manual de la función `plot`.

Asimismo, será necesario añadir anotaciones[11] sobre la gráfica representada. Algunas funciones que te serán de utilidad para realizar estas anotaciones son `text`, `title` y `grid`. Además, la configuración de los ejes[12] se puede realizar mediante la función `axis`. Finalmente, se puede guardar la gráfica a fichero para incluirla posteriormente en la memoria mediante la función `print`[13].

[10] https://octave.org/doc/v4.2.1/Two_002dDimensional-Plots.html
[11] https://octave.org/doc/v4.2.1/Plot-Annotations.html
[12] https://octave.org/doc/v4.2.1/Axis-Configuration.html
[13] https://octave.org/doc/v4.2.1/Printing-and-Saving-Plots.html

E. Recordatorio de teoría de SVM

Sea $\phi(\mathbf{x};\boldsymbol{\theta},\theta_0)$, una FDL obtenida mediante el método SVM, donde $\boldsymbol{\theta}$ es el vector de pesos óptimo que se calcula como:

$$\boldsymbol{\theta} = \sum_{m\in\mathcal{V}} c_m\ \alpha_m\ \mathbf{x}_m$$

siendo c_m, la etiqueta de clase, α_m, el multiplicador de Lagrange óptimo, y $\mathcal{V}$, el conjunto de vectores soporte. El peso umbral θ_0, por las condiciones de KKT se calcula para cualquier vector soporte $m \in \mathcal{V}$ correcto ($\alpha_m < \mathcal{C}$) como:

$$\theta_0 \ = \ c_m - \boldsymbol{\theta}^t\mathbf{x}_m$$

El margen es $\frac{2}{\|\boldsymbol{\theta}\|}$ y la tolerancia de margen ζ_m para un vector soporte $m \in \mathcal{V}$ se calcula como:

$$\zeta_m = 1 - c_m(\boldsymbol{\theta}^t\mathbf{x}_m + \theta_0)$$

Para el caso bidimensioanl donde $\mathbf{x},\boldsymbol{\theta} \in \mathbb{R}^2$, tenemos que la ecuación de la recta de separación asociada a ϕ se calcula como:

$$\phi(\mathbf{x};\boldsymbol{\theta},\theta_0) = 0 \quad \Rightarrow \quad \theta_1x_1 + \theta_2x_2 + \theta_0 = 0 \quad \Rightarrow \quad x_2 = -\frac{\theta_1}{\theta_2}x_1 - \frac{\theta_0}{\theta_2},$$

mientras que las ecuaciones de las rectas que definen las fronteras del margen son:

$$\phi(\mathbf{x};\boldsymbol{\theta},\theta_0) = +1 \quad \Rightarrow \quad \theta_1x_1 + \theta_2x_2 + \theta_0 = +1 \quad \Rightarrow \quad x_2 = -\frac{\theta_1}{\theta_2}x_1 - \frac{\theta_0 - 1}{\theta_2}$$

$$\phi(\mathbf{x};\boldsymbol{\theta},\theta_0) = -1 \quad \Rightarrow \quad \theta_1x_1 + \theta_2x_2 + \theta_0 = -1 \quad \Rightarrow \quad x_2 = -\frac{\theta_1}{\theta_2}x_1 - \frac{\theta_0 + 1}{\theta_2}$$