



**Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza**

APRENDIZAJE AUTOMÁTICO

Practica 6: Análisis de componentes principales

Autora:
Nerea Gallego Sánchez (801950)

7 de diciembre de 2022

Índice

| | |
|--|----|
| 1. Introducción y objetivos | 2 |
| 2. Reducción de la dimensión en MNIST | 2 |
| 3. Compresión de imágenes utilizando SVD | 10 |

1. Introducción y objetivos

El objetivo de la práctica es utilizar técnicas de reducción de dimensión basadas en el análisis de componentes principales.

Para ello se utilizan técnicas de PCA para la reconstrucción de dígitos y su clasificación.

En la segunda parte se utiliza SVD para comprimir una imagen.

2. Reducción de la dimensión en MNIST

En esta parte se debe utilizar el mismo conjunto de imágenes que se utilizó en la práctica anterior. Para ello se utiliza el algoritmo de clasificación bayesiana construido en la práctica anterior. Las funciones que ya se construyeron en la práctica anterior son: [entrenarGaussianas](#), [clasificacionBayesiana](#) y [errorRegularizacionMulticlase](#).

```
1 % Dados los atributos de entrada, la salida esperada, la cantidad de clases
2 % que se quieren predecir, si se quiere usar bayes ingenuo o no y el
3 % parametro de regulariacion que se quiere usar.
4 % Devuelve el modelo para realizar el entrenamiento.
5 function modelo = entrenarGaussianas( Xtr, ytr, nc, NaiveBayes, landa )
6 % Entrena una Gaussiana para cada clase y devuelve:
7 % modelo{i}.N      : Numero de muestras de la clase i
8 % modelo{i}.mu     : Media de la clase i
9 % modelo{i}.Sigma  : Covarianza de la clase i
10 % Si NaiveBayes = 1, las matrices de Covarianza seran diagonales
11 % Se regularizaran las covarianzas mediante: Sigma = Sigma + landa*eye(D)
12 modelo = struct('N',{},{},'mu',{},{},'Sigma',{},{ });
13     for i = 1:nc
14         % Cantidad de muestras que hay de esa clase
15         modelo(i).N = sum(ytr == i);
16         % Guarda las muestras de la clase i en Xp
17         Xp = Xtr(ytr == i,:);
18         % Media para la clase i
19         modelo(i).mu = mean(Xp);
20         % Matriz de covarianza
21         modelo(i).Sigma = cov(Xp);
22         modelo(i).Sigma = modelo(i).Sigma + landa * eye(size(Xp,2));
23         % Si se quiere usar Bayes ingenuo se queda con la matriz diagonal de
24         % covarianzas.
25         if NaiveBayes == 1
26             modelo(i).Sigma = diag(diag(modelo(i).Sigma));
27         end
28     end
29 end
```

```
1 % Dado el modelo y los datos de entrada, devuelve la clase predicha para
2 % cada muestra.
3 function yhat = clasificacionBayesiana(modelo, X)
4 % Con los modelos entrenados, predice la clase para cada muestra X
5     y = [];
6     for i=1:10
7         % Calcula la verosimilitud de las muestras para una clase.
8         y = [y gaussLog(modelo(i).mu, modelo(i).Sigma,X)];
9         % Probabilidad a posteriori
10        y(:,i) = y(:,i).*(modelo(i).N/size(X,1));
11    end
12    % Clase predicha
13    [~,yhat] = max(y,[],2);
14 end
```

```
1 % Dada la entrada, la salida esperada y el modelo, devuelve la
2 % media de casos erroneos que se obtienen.
3 function err = errorRegularizacionMulticlase(X,y,t)
4     % Calcula la prediccion obtenida con la matriz de pesos
5     ypred = clasificacionBayesiana(t,X);
6     % Se suma la cantidad de filas que son distintas de las que hay en el
7     % vector de salidas esperadas, y se calcula la media de error.
```

```

8     err = sum(y ~= ypred) / size(y,1);
9 end

```

El primer apartado del primer ejercicio pide encontrar la mínima cantidad de dimensiones con las que debes quedarte para no alterar significativamente los resultados de la clasificación.

Para buscar este valor, primero se han normalizado los datos con respecto a la media y a continuación, mediante la función *eig* de Matlab se han obtenido los vectores y valores propios del conjunto de datos inicial. También ha sido necesario ordenar la matriz con los vectores propios según los valores propios obtenidos.

```

1 Xp = normalize(X,'center','mean'); % normalizar los datos respecto de la media
2
3 % obtener la matriz de covarianza
4 sig = cov(Xp);
5 % obtener los valores y vectores propios
6 [U,A] = eig(sig);
7 % ordenar los vectores propios segun los valores propios
8 [A,I] = sort(diag(A),'descend')
9 U = U(:,I);

```

Para obtener la mínima cantidad de dimensiones con las que debes realizar el entrenamiento de manera que no se alteren los resultados, se ha decidido iterar a lo largo de los valores de k de manera que se obtenga el mínimo valor de k que tenga un error similar al obtenido en la práctica anterior ($err = 3,500000e-02$). Para ello se ha comenzado a comprobar los distintos valores de k , desde 1 buscando su mejor parámetro de regularización de manera que el mejor error de validación obtenido sea tan solo un 1% mayor que el error que se obtuvo en la práctica anterior.

El algoritmo construido itera para cada valor de k , reduce la dimensión de los datos a k . Separa los datos de entrenamiento y de validación y a continuación, busca el mejor parámetro de regularización entre unos valores de λ prefijados.

El algoritmo termina cuando encuentra un valor de k con el que se obtiene un error de validación que no supera en un 1% al error obtenido en la práctica anterior.

```

1 err = 3.5*10^(-2)
2 elegido = 0
3 errK = []
4 while k <= size(X,2) && elegido == 0
5     bestModel = 0;
6     bestErrV = inf;
7     errT = []; errV = [];
8     naiveBayes = 0;
9     Uk = U(:,1:k);
10    Z = Xp * Uk;
11    [Xtr, Ytr, Xv, Yv] = separar([Z y],0.8,[(1:k)],[k+1]);
12    for model = 1:size(lambda,1)
13        err_T = 0; err_V = 0;
14        % se calcula la matriz de pesos
15        th = entrenarGaussianas(Xtr, Ytr, 10, naiveBayes, lambda(model));
16
17        % se calculan los valores de error
18        err_T = errorRegularizacionMulticlase(Xtr,Ytr,th);
19        err_V = errorRegularizacionMulticlase(Xv,Yv,th);
20        % se guardan los valores de error obtenidos para posteriormente
21        % imprimir una grafica
22        errT = [errT err_T];
23        errV = [errV err_V];
24        % si el modelo es mejor que el anterior, se guarda
25        if err_V < bestErrV
26            bestModel = model;
27            bestErrV = err_V;
28        end
29    end
30    % si mejor error del modelo obtenido tan solo un 1% mayor que el error
31    % que se obtuvo en la practica anterior, ya se ha obtenido el numero de
32    % componentes con las que te puedes quedar sin alterar el resultado
33    errK = [errK bestErrV];
34    if(bestErrV<=err*1.01)

```

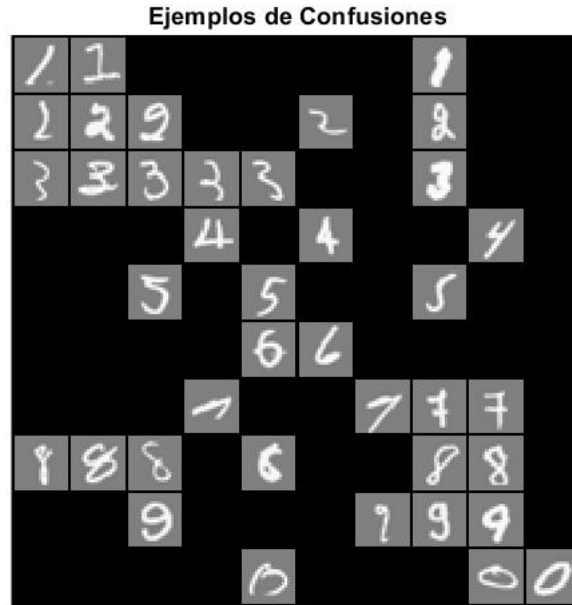


Figura 1: Confusiones obtenidas reduciendo la dimensión a 21 componentes.

```

35     elegido = 1;
36     end
37     k = k + 1
38 end

```

De esta manera se obtiene un valor de $k = 21$.

Las confusiones obtenidas con esta cantidad de componentes es la que se puede observar en la imagen 1. Se puede ver que son muy similares a las que se obtuvieron en la práctica anterior 2.

En la segunda parte se pide reducir la dimensión de los datos de manera que se mantenga el 99 % de la variabilidad. Esto se realiza buscando el mínimo valor de k que contenga el 99 % de la covarianza inicial. Esto se puede calcular con la siguiente formula: $\frac{\sum_{i=1}^k \lambda_i}{\sum_{i=1}^n \lambda_i} > 0,99$ se manera que λ_i es el valor propio i .

De esta manera se obtiene la cantidad de dimensiones con las que se van a realizar el entrenamiento.

Mediante esta fórmula se ha obtenido el valor $k = 153$.

Para comenzar a evaluar la clasificación, se ha utilizado un el algoritmo construido en la práctica anterior. Inicialmente, se han obtenido los datos a los que se les ha reducido la dimensión.

```

1 Uk = U(:,1:k);
2 Z = Xp * Uk;

```

A continuación, mediante la función `separar` se han separado el 80 % de los datos de entrenamiento para realizar una validación de los distintos parámetros de regularización.

```

1 % Dado el conjunto de datos completo, el porcentaje de datos que se quieren
2 % usar como datos de entrenamiento, las columnas de los atributos y las
3 % columnas de las salidas esperadas, devuelve el conjunto de datos de
4 % entrenamiento con sus salidas esperadas y el conjunto de datos de
5 % validacion con sus salidas esperadas.
6 function [Xtr, Ytr, Xtst, Ytst] = separar(data, porc, col_x, col_y)

```

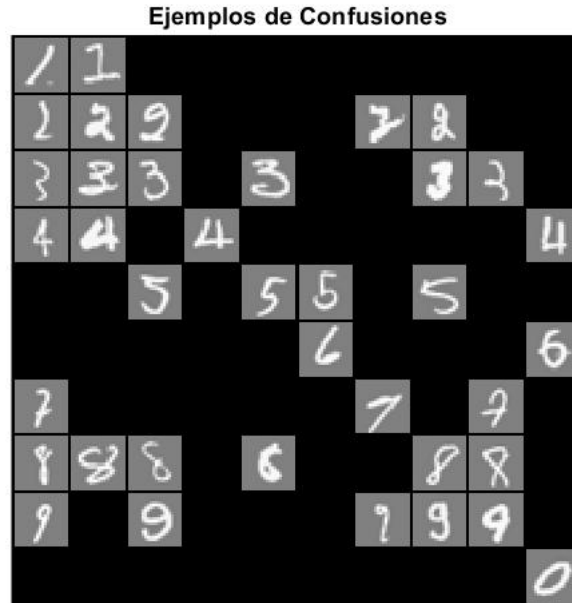


Figura 2: Confusiones obtenidas en la práctica anterior.

```

7 % Realiza una permutación de los datos para que se escojan de manera
8 % aleatoria.
9 randOrd = randperm(size(data,1));
10 perm = data(randOrd,:);
11 % Establece el límite entre los datos de entrenamiento y los de
12 % validación.
13 lim = int32(size(data,1)*porc);
14 Xtr = perm(1:lim,col_x);
15 Ytr = perm(1:lim,col_y);
16 Xtst = perm(lim+1:end,col_x);
17 Ytst = perm(lim+1:end,col_y);
18 end

```

Una vez se han separado tanto los datos de entrenamiento como los de validación, se ha utilizado el algoritmo construido en la práctica anterior para buscar el mejor parámetro de regularización. Dicho algoritmo utiliza la clasificación de Bayes completo. Para empezar, crea un vector con los valores de λ que se van a probar. Estos valores se encuentran entre 10^{-6} y 10. A continuación, para cada posible valor de regularización, construye el modelo de entrenamiento y calcula el error obtenido tanto con los datos de entrenamiento como con los datos de test. En caso de que el error de validación sea el menor error obtenido hasta el momento, guarda como mejor modelo el actual. Al final, el modelo imprime en una gráfica la evolución de los errores que se obtienen con los distintos grados del parámetro lambda y muestra cuál es el valor del mejor parámetro de regularización.

```

1 lambda = logspace(-6,1,50)';
2
3 bestModel = 0;
4 bestErrV = inf;
5 errT = [];
6 errV = [];
7 naiveBayes = 0;
8 for model = 1:size(lambda,1)
9     err_T = 0;
10    err_V = 0;
11    % se calcula la matriz de pesos
12
13    th = entrenarGaussianas(Xtr, Ytr, 10, naiveBayes, lambda(model));
14    % se calculan los valores de error

```

```

15     err_T = errorRegularizacionMulticlase(Xtr,Ytr,th);
16     err_V = errorRegularizacionMulticlase(Xv,Yv,th);
17     % se guardan los valores de error obtenidos para posteriormente
18     % imprimir una grafica
19     errT = [errT err_T];
20     errV = [errV err_V];
21     % si el modelo es mejor que el anterior, se guarda
22     if err_V < bestErrV
23         bestModel = model;
24         bestErrV = err_V;
25     end
26 end
27
28 figure;
29 plot(log10(lambda(bestModel)),errV(bestModel),'go');
30 hold on
31 plot(log10(lambda), errT,'b-','LineWidth', 1);
32 plot(log10(lambda), errV, 'r-','LineWidth', 1);
33 legend('Parametro de regularizacion ideal','Error de entrenamiento','Error de validacion
34 ')
35 hold off
36
37 title('Curva del error')
38 ylabel('Error'); xlabel('Grado de lambda');
39 sprintf("El mejor modelo obtenido es con lambda = %s",lambda(bestModel))

```

El mejor parámetro de regularización obtenido es $\lambda = 1,389495e - 02$. Muy similar al obtenido en la práctica anterior ($\lambda = 3,856620e - 02$). Además, el error obtenido es $err = 2,625000e - 02$ también muy parecido al obtenido en la práctica anterior $err = 3,500000e - 02$.

En la figura 3 se puede observar la curva de error que se produce con los distintos grados que adopta el parámetro lambda. Se puede corroborar como el parámetro que produce menor error es el indicado anteriormente.

En la gráfica también se muestra como, utilizar un parámetro de regularización más pequeño produce sobreajuste, ya que el error obtenido con los datos de entrenamiento es muy pequeño, pero el error obtenido con los datos de validación es bastante mayor. Esto se debe a que la penalización que realiza a modelos complejos es pequeña. Sin embargo, si se aumenta mucho el valor del parámetro λ se obtiene subajuste, ya que penaliza los modelos complejos y lo simplifica a uno más sencillo. Los errores que se obtienen tanto con los datos de entrenamiento como de validación se equiparan pero son muy grandes.

Una vez obtenidos los datos con la reducción de dimensión de manera que se mantenga el 99% de la variabilidad, es necesario conocer que tan buenos son los resultados de la clasificación.

Para ello, se han utilizado los datos de test. Primero ha sido necesario normalizar los datos de test con respecto de la media. Esto se ha realizado de la misma manera que con los datos de entrenamiento. A continuación, se ha reducido la dimensión de los datos a 153 dimensiones. Con los datos de entrenamiento y el parámetro de regularización previamente calculado, se ha construido y entrenado el modelo. Por último se ha realizado la clasificación con los datos de test mediante la función [clasificacionBayesiana](#) y se ha calculado la matriz de confusión mediante la función [matrizConfusion](#).

```

1 % Dada la salida predicha, la salida esperada y la cantidad de clases,
2 % devuelve la matriz de confusi n de manera que contiene en las filas la
3 % clase real y en las columnas, la clase predicha.
4 function matrizConf = matrizConfusion(ypred,ytest,clases)
5     matrizConf = zeros(clases,clases);
6     for i = 1:clases
7         for j = 1:clases
8             matrizConf(i,j) = sum(ytest==i & ypred == j);
9         end
10    end
11 end

```

Como se puede observar en la [matriz de confusion](#) el número más problemático es el 8, seguido del 3 y del 5. De la misma manera, lo números que más se confunden con otros son el 2, y el 8.

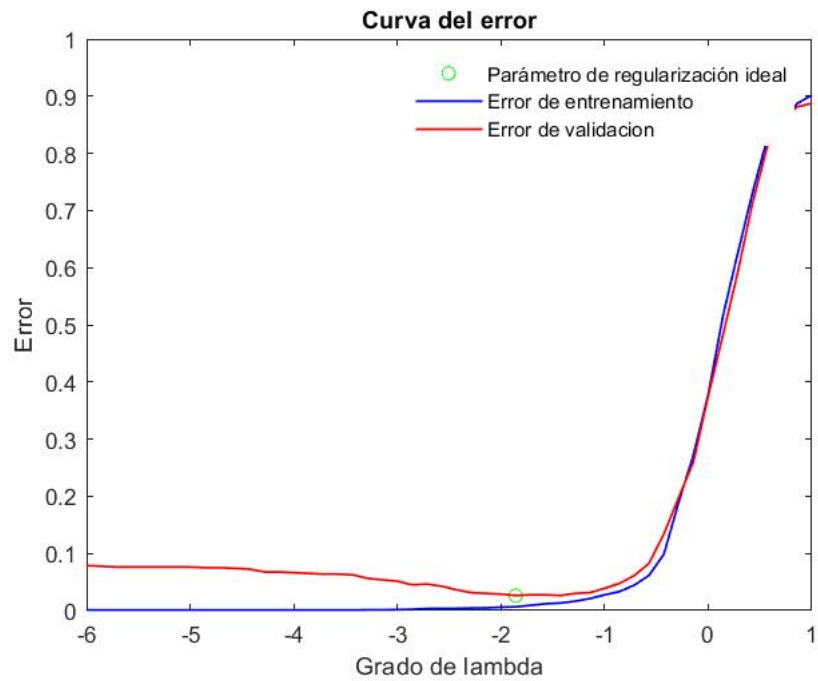


Figura 3: Curva de errores al variar el grado del parámetro lambda.

| | | Clase predicha | | | | | | | | | |
|------------|---|----------------|-----|----|----|----|----|----|-----|----|-----|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
| Clase real | 1 | 98 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 2 | 0 | 99 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 3 | 0 | 3 | 93 | 1 | 1 | 0 | 0 | 2 | 0 | 0 |
| | 4 | 0 | 1 | 0 | 97 | 0 | 0 | 0 | 0 | 1 | 1 |
| | 5 | 0 | 0 | 2 | 0 | 93 | 1 | 0 | 4 | 0 | 0 |
| | 6 | 0 | 0 | 0 | 0 | 2 | 98 | 0 | 0 | 0 | 0 |
| | 7 | 1 | 0 | 0 | 0 | 0 | 0 | 97 | 0 | 2 | 0 |
| | 8 | 1 | 2 | 2 | 0 | 1 | 0 | 0 | 93 | 1 | 0 |
| | 9 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 2 | 95 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| TOTAL | | 100 | 106 | 99 | 98 | 97 | 99 | 98 | 103 | 99 | 101 |

Cuadro 1: Matriz de confusión 153 componentes.

| | Precision | Recall |
|---|-----------|--------|
| 1 | 0.9800 | 0.9800 |
| 2 | 0.9340 | 0.9900 |
| 3 | 0.9394 | 0.9300 |
| 4 | 0.9898 | 0.9700 |
| 5 | 0.9588 | 0.9300 |
| 6 | 0.9899 | 0.9800 |
| 7 | 0.9898 | 0.9700 |
| 8 | 0.9029 | 0.9300 |
| 9 | 0.9596 | 0.9500 |
| 0 | 0.9901 | 1.0000 |

Cuadro 2: Tabla de precisión/recall con reducción de dimensión a 153 componentes.

También se calcula la precisión y el recall para cada una de las clases. Se ha utilizado el código escrito en la práctica anterior para calcular estas medidas.

En la tabla precision/recall se muestran los resultados obtenidos tanto de precisión como de recall para cada una de las clases.

Como se puede observar, en la tabla 2 las clases con menor precisión son el 8, seguido del 2 y el 3. Esto es debido a que en numerosas ocasiones se predicen que las entradas son de esta clase, cuando en realidad no lo son. De la misma manera, las clases con peor recall son el 8, 3, 5 y 9. Como ya hemos podido observar anteriormente en la matriz de confusión, son las clases que peor predicción tienen y por consiguiente, los número más problemáticos.

Por último, se pide visualizar las confusiones con la función *verConfusiones*.

Como se puede ver en la imagen 4 el modelo confunde algunos dígitos con otros debido a su mala grafía. Se muestran más ejemplos de aquellos números que tienen una peor predicción como pueden ser el 8 o el 3.

Los resultados obtenidos en la práctica anterior fueron los contenidos en las tablas 3 que contiene la matriz de confusión y en la tabla 4 que contiene la precisión y el recall.

| | | Clase predicha | | | | | | | | | |
|------------|---|----------------|-----|-----|----|----|-----|----|----|----|-----|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
| Clase real | 1 | 99 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 2 | 1 | 96 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| | 3 | 1 | 2 | 94 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| | 4 | 1 | 1 | 0 | 97 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 5 | 0 | 0 | 1 | 0 | 94 | 1 | 0 | 4 | 0 | 0 |
| | 6 | 0 | 0 | 0 | 0 | 0 | 99 | 0 | 0 | 0 | 1 |
| | 7 | 2 | 0 | 0 | 0 | 0 | 0 | 96 | 0 | 2 | 0 |
| | 8 | 2 | 2 | 3 | 0 | 1 | 0 | 0 | 91 | 1 | 0 |
| | 9 | 1 | 0 | 2 | 0 | 0 | 0 | 1 | 2 | 94 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| TOTAL | | 107 | 102 | 101 | 97 | 96 | 100 | 98 | 99 | 98 | 102 |

Cuadro 3: Matriz de confusión con Bayes completo.

Como ya se ha observado anteriormente, los números de confusiones son muy similares, a excepción del número 1, que tiene se ha conseguido tener mejores resultados. También se puede observar como se obtienen tasas muy similares de precisión y de recall.

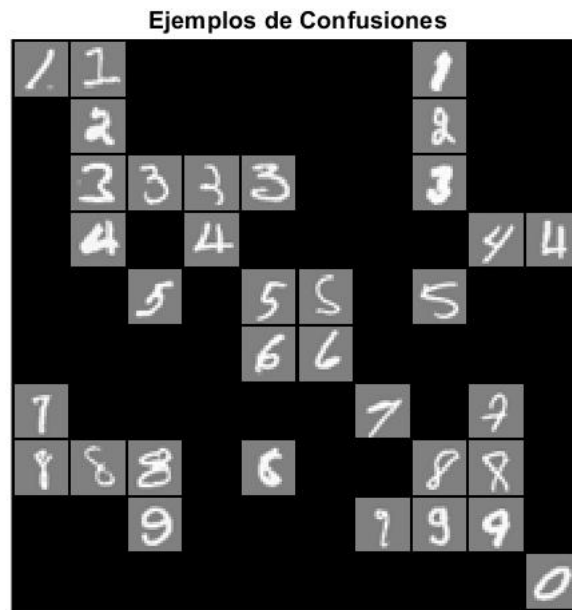


Figura 4: Ejemplos de confusión con los dígitos para la clasificación tras reducir la dimensión a 153 componentes.

| | Precision | Recall |
|---|-----------|--------|
| 1 | 0.9252 | 0.99 |
| 2 | 0.94126 | 0.96 |
| 3 | 0.9307 | 0.94 |
| 4 | 1 | 0.97 |
| 5 | 0.9792 | 0.94 |
| 6 | 0.99 | 0.99 |
| 7 | 0.9796 | 0.96 |
| 8 | 0.9192 | 0.91 |
| 9 | 0.9592 | 0.94 |
| 0 | 0.9804 | 1 |

Cuadro 4: Tabla de precisión/recall con Bayes completo.


 imagenes-p6/ngc1.jpg

Figura 5: Resultado de la compresión de la imagen.


 imagenes-p6/nerea_gallego1.jpg

Figura 6: Imagen inicial.

3. Compresión de imágenes utilizando SVD

Se pide utilizar el algoritmo de SVD para comprimir una foto de mi cara. La foto utilizada es la adjuntada en la portada.

Para empezar se ha convertido la imagen a escala de grises y se han convertido las componentes para que sean del formato double.

```
1 % Convert to B&W
2 BW = rgb2gray(I);
3
4 % Convert data to double
5 X=im2double(BW);
```

Para aplicar el algoritmo de SVD se ha utilizado la función `svd` de matlab para obtener las matrices correspondientes para realizar la compresión.

```
1 % Apply SVD
2 [U,S,V]=svd(X);
3 V = V';
```

Para comprimir la imagen y que mantenga legibilidad, se ha buscado una cantidad de componentes de manera que la imagen mantenga el 90 % de la legibilidad. Esto se ha realizado con la fórmula: $\frac{\sum_{i=1}^k \lambda_i}{\sum_{i=1}^n \lambda_i} > 0,90$

De esta manera se ha obtenido un valor de $k = 173$. Para realizar la compresión de la imagen se ha creado la función `getDescomposicionAcum` de manera que dados los vectores y valores propios de la descomposición `svd` y la cantidad de componentes que se quieren guardar, devuelve el resultado de realizar el producto con la compresión realizada.

```
1 function Xhat = getDescomposicionAcum(U,S,V,k)
2     S2 = S(1:k,:);
3     Xhat = U(:,1:k)*S2(:,1:k)*V(1:k,:);
4 end
```

En la imagen 5 se puede observar el resultado de la compresión a 173 componentes. Se puede observar como la imagen es muy similar a la imagen inicial contenida en 6.

Para comprobar si la compresión ha merecido la pena, se ha calculado la cantidad de componentes que se ahorra guardar si se guardaran las k componentes necesarias de cada matriz. Esto se ha calculado con la siguiente formula:

```
1 U2 = U(:,1:k);
2 S2 = S(1:k);
3 V2 = V(1:k,:);
4 total = size(X,1)*size(X,2);
5 new = (size(U2,1)*size(U2,2)+size(S2,1)*size(S2,2)+size(V2,1)*size(V2,2));
6 (total - new)*100/total
```

El resultado obtenido es que se ahorra un 42,0670 % de las componentes a guardar.