Aprendizaje Automático (2017-2018)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 3.

Simón López Vico

19 de mayo de 2018

1. Problema de clasificación

1. **Comprender el problema a resolver.**

   El problema a realizar se basa en el reconocimiento óptico de un conjunto de datos de dígitos manuscritos.

   La base de datos ha sido realizada por un total de 43 personas, 30 contribuyeron al conjunto de datos *train* y 13 para el conjunto de *test*.

   Los mapas de bits de 32×32 se dividen en bloques no solapables de 4×4 y se cuenta el número de píxeles con valor igual a 1 en cada bloque. Esto genera una matriz de entrada de 8×8 donde cada elemento es un entero en el rango $[0, 16]$; esto reduce la dimensionalidad y da invariancia a pequeñas distorsiones.

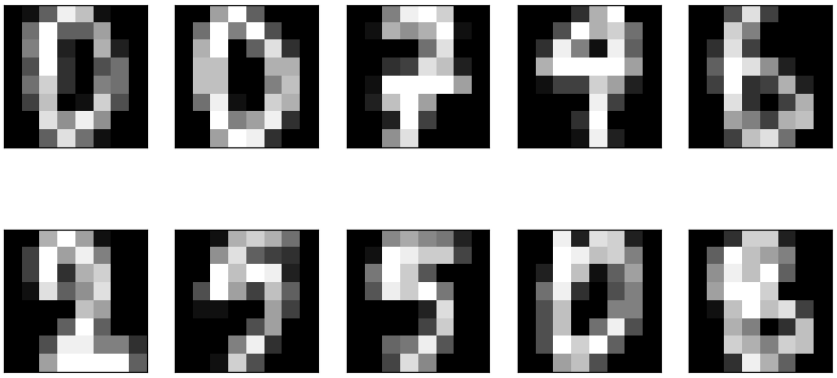


Figura 1: Representación de algunos dígitos.

   La base de datos sobre la que trabajaremos tiene 5620 instancias, con un total de 64 atributos de tipo entero.

2. **Preprocesado los datos: por ejemplo categorización, normalización, etc.**

   Hemos usado normalización para el prepocesado de los datos. Para ello, hemos dividido cada elemento por 16, pues es el máximo valor en el rango numérico de los atributos, quedando todos nuestros datos normalizados en el intervalo $[0, 1]$.

   Dicha normalización la hemos implementado mediante el siguiente código:

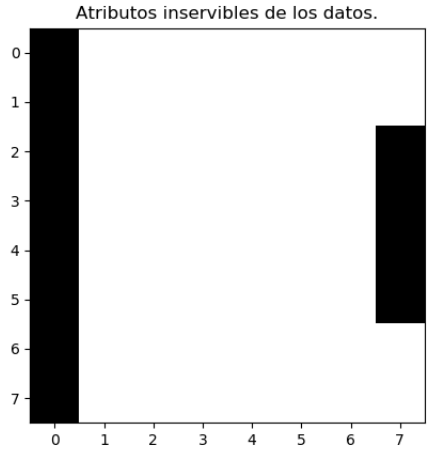
```
def readData(file_x, file_y):
x = np.load(file_x)
y = np.load(file_y)
x_ = np.empty(x.shape, np.float64)
```

```
    for i in range(0,x.shape[0]):
        for j in range(0,x.shape[1]):
            x_[i][j] = np.float64(1.0*x[i][j]/16.0)

    return x_, y
```

Por otra parte, eliminaremos atributos inservibles de nuestro conjunto de datos mediante las funciones de de *Sklearn* `VarianceThreshold(threshold)`, que devolverá un selector de las características con una varianza menor a `threshold`, y `fit_transform`.

```
# Buscamos característias inservibles que eliminar
inutil = VarianceThreshold(threshold=(0.001))
X_train = inutil.fit_transform(X_train)
```

En concreto, se eliminarán 12 atributos. Podremos plotear las características eliminadas, siendo las siguientes:



Por último, agregaremos complejidad al modelo realizando operaciones sobre los atributos de los que ya disponemos, generando nuevos atributos no lineales. Para ello, usaremos características polinomiales de la siguiente manera:

```
# Usamos características polinomiales para añadir información
poly = PolynomialFeatures(2)
X_train= poly.fit_transform(X_train)
```

Así, añadiremos operaciones de polinomios de grado 2 sobre nuestros atributos como nuevas características para nuestros datos.

3. **Selección de clases de funciones a usar.**

   Para la realización del problema hemos decidido usar regresión logística.

4. **Definición de los conjuntos de training, validación y test usados en su caso.**

   Los conjuntos de training y de test vienen dados en los archivos `optdigits_tra_X.npy, optdigits_tra_y.npy` y `optdigits_tes_X.npy, optdigits_tes_y.npy` respectivamente.

   Usaremos validación para evaluar nuestro clasificador y calcular `c` para evitar sobreajuste; en concreto, usaremos "Validación Cruzada" (*K-fold*).

   Dicha técnica consiste en separar los datos de train en K conjuntos, escogiendo uno de ellos como datos de prueba y el resto como datos de entrenamiento; estas K particiones será disjuntas con la distribución de etiquetas equilibrada (*Stratified*). Dicho proceso se repite K veces, eligiendo cada vez uno de los subconjuntos como datos test. Para terminar, realizaremos la media de los resultados obtenidos en cada iteración para obtener un único resultado.

Vamos a crear varios modelos, cada uno con un `c` diferente, y a validar nuestros datos sobre cada uno de estos modelos, quedándonos con la información del que mejores resultados nos dé. El parámetro `c` será la inversa de la "*regularization strenght*" (fuerza de regularización); los valores más pequeños especificarán una mayor regularización.

El código usado para implementar dicha técnica será el siguiente:

```
# Aplicamos K-FOLD
k = 2 # Número de subconjuntos a generar
kfold = StratifiedKFold(n_splits=k) # Inicializamos el K-Fold
kfold.get_n_splits(X_train,y_train) # Separamos los conjuntos de entrenamiento
mejor_C = 0
mejor_media = 0
ces = []
medias = []

for i in range(-5,5):
    suma = 0
    c = 10**i # El crecimiento de nuestro c será exponencial

    for train_index, test_index in kfold.split(X_train,y_train):
        X_train_, X_test_ = X_train[train_index], X_train[test_index]
        y_train_, y_test_ = y_train[train_index], y_train[test_index]

        # Aplicamos regresión logística con el parámetro c
        lr = LogisticRegression(C=c)
        # Ajustamos el modelo a partir de los datos
        lr.fit(X_train_, y_train_)
        # Calculamos error respecto de test
        predictions = lr.predict(X_test_)
        # Sumamos el porcentaje de acierto
        suma += lr.score(X_test_, y_test_)

    media = 1.0*suma/k

    # Guardamos la c y la media para la gráfica
    ces.append(c)
    medias.append(media)

    # Si mejora el porcentaje de acierto con el nuevo c
    if media > mejor_media:
        mejor_media = media
        mejor_C = c
```

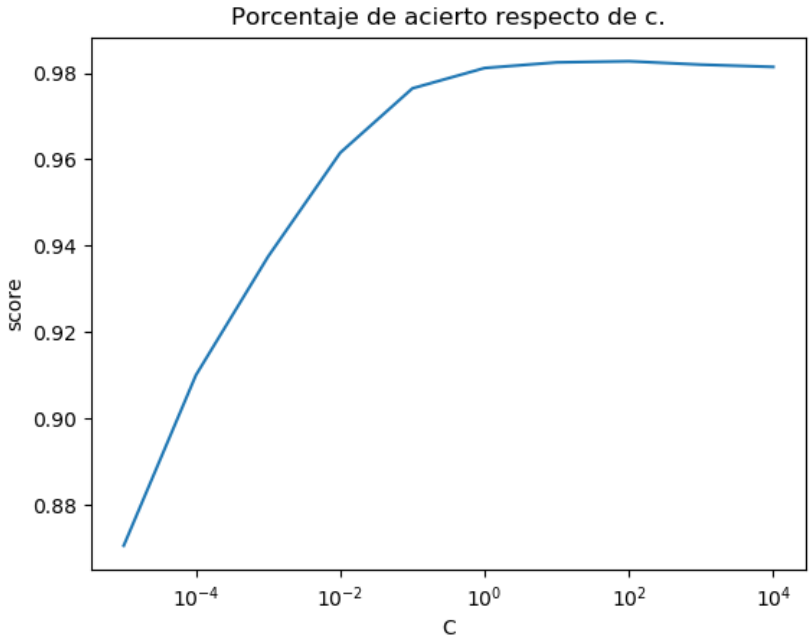
Dicho código incluye la implementación del siguiente apartado.

5. **Discutir la necesidad de regularización y en su caso la función usada para ello.**

   Para comprobar la necesidad de regularización, crearemos varios modelos (como hemos comentado en el apartado anterior), cada uno de ellos con un "*regularization strenght*" $\lambda = \frac{1}{c^i}$, donde i tomará valores enteros del intervalo $[-5, 5]$.

   Para cada conjunto test obtenido mediante la validación cruzada, calcularemos el acierto respecto del ajuste realizado con la variable c, calculando una media de aciertos por cada c. Finalmente, escogeremos el parámetro c que mejores resultados nos haya dado (en nuestro caso, $c = 100$).

   En la siguiente gráfica, podemos ver como cambia el porcentaje de acierto para cada c^i:



Se aprecia que la gráfica alcanza su máximo en $C = 10^2$.

El código para implementar la regularización es el especificado en el apartado anterior.

6. **Definir los modelos a usar y estimar sus parámetros e hyperparámetros.**

   Los distintos modelos usados consistirán en una regresión logística variando el parámetro c asociado a ella.

   El único parámetro es el valor c, el cual va cambiando creando un nuevo modelo de regresión logística. Respecto a los hyperparámetros (parámetro cuyo valor se establece antes de que comience el proceso de aprendizaje), tenemos fijado el valor $k = 2$, el cual establece el número de subconjuntos a generar mediante la validación cruzada.

7. **Selección y ajuste modelo final.**

   Como modelo final, seleccionaremos la regresión logística con el parámetro c que mejor porcentaje medio nos haya dado en la regularización, siendo $c = 100$.

   Para ajustar el modelo, haremos como en los anteriores casos:

```
# Ajustamos el modelo a partir de los datos
lr = LogisticRegression(C=mejor_C)
lr.fit(X_train, y_train)
```

8. **Discutir la idoneidad de la métrica usada en el ajuste.**

9. **Estimación del error E_{out} del modelo lo más ajustada posible.**

   Para obtener un E_{out} lo más ajustado posible, escogeremos el modelo final y calcularemos la precisión sobre los datos test facilitados por el profesor (`optdigits_tes_X.npy, optdigits_tes_y.npy`).

   Usaremos el siguiente código, en el que se calcula la precisión (*score*) respecto de los datos test, por lo que $E_{out} = 1 - score$.

```
# Leemos los datos de test y los preprocesamos igual que los de train
X_test, y_test = readData('optdigits_tes_npy/optdigits_tes_X.npy',
                          'optdigits_tes_npy/optdigits_tes_y.npy')
X_test = inutil.transform(X_test)
X_test = poly.fit_transform(X_test)
```

```
# Calculamos el score con dicho ajuste
predictions = lr.predict(X_test)
score = lr.score(X_test, y_test)

print()
print('Valor de acierto con el mejor c: ', score)
print('E_out=', 1-score)
```

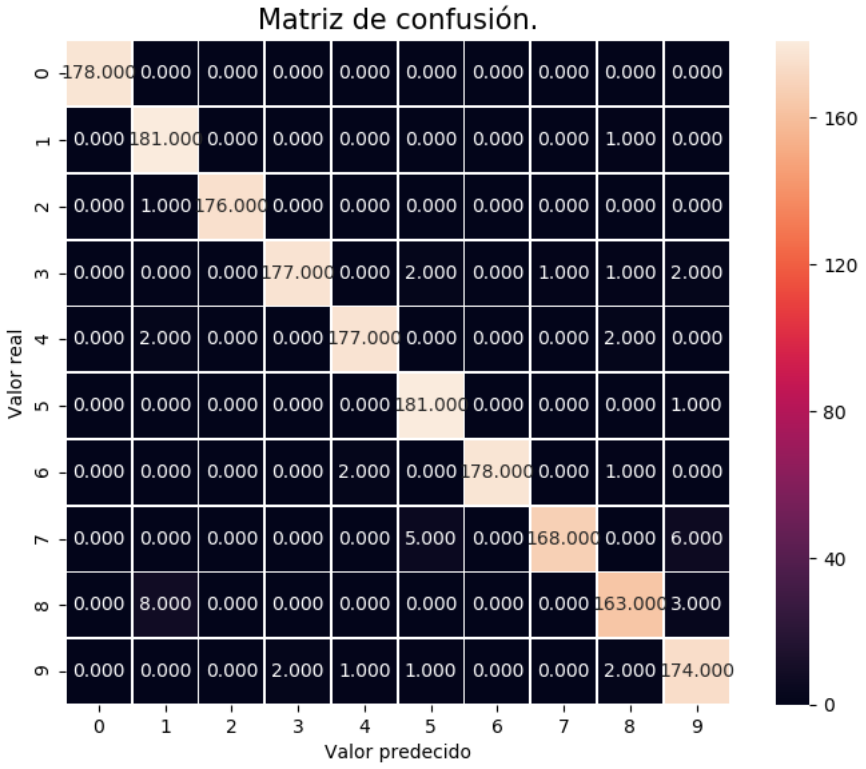
Ejecutando dicho código, obtendremos como resultados:

$$score = 0{,}9755147468002225 \rightarrow E_{out} = 0{,}024485253199777457$$

10. **Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.**

    El modelo elegido realiza un buen ajuste de los datos, pues estamos obteniendo un porcentaje de precisión del 97 % sobre los datos de test.

    Para comprobar la calidad del ajuste, generaremos una matriz de confusión. En ella podremos ver la cantidad de veces que ha fallado nuestro modelo entre los valores reales y los valores predecidos, pudiendo así deducir nuevas características que añadir a nuestro conjunto de entrenamiento para distinguir entre ciertos casos. La matriz de confusión generada es la siguiente:



Como podemos ver, los valores de la diagonal son mayores que los del resto; esto es porque el modelo predice bastante bien, y da muchos más aciertos que fallos.

Podemos comprobar que nuestro modelo ha confundido 8 veces el número 1 con el 8, y 6 veces el número 7 con el 9, por lo que no sería una mala idea añadir un atributo al conjunto de datos mediante el que se diferenciasen dichas cifras, para así obtener mejores resultados.

2. Problema de regresión.

1. **Comprender el problema a resolver.**

   Para el problema de regresión usaremos la base de datos Airfoil Self-Noise, la cual contendrá un conjunto de datos obtenidos por la NASA a partir de una serie de ensayos aerodinámicos y acústicos de perfiles de paletas bidimensionales y tridimensionales realizados en un túnel aerodinámico anecoico (diseñado para absorber en su totalidad las reflexiones producidas por ondas acústicas o electromagnéticas en cualquiera de las superficies que la conforman).

   Cada dato estará formado por 6 atributos numéricos reales, siendo el último la etiqueta, y serán los siguientes:

   - Frecuencia.
   - Ángulo de ataque.
   - Longitud de cuerda.
   - Velocidad de flujo libre.
   - Grosor de desplazamiento del lado de aspiración.
   - Nivel de presión sonora escalonado.

   A continuación se muestra distribución de los datos respecto de sus atributos de dos en dos:

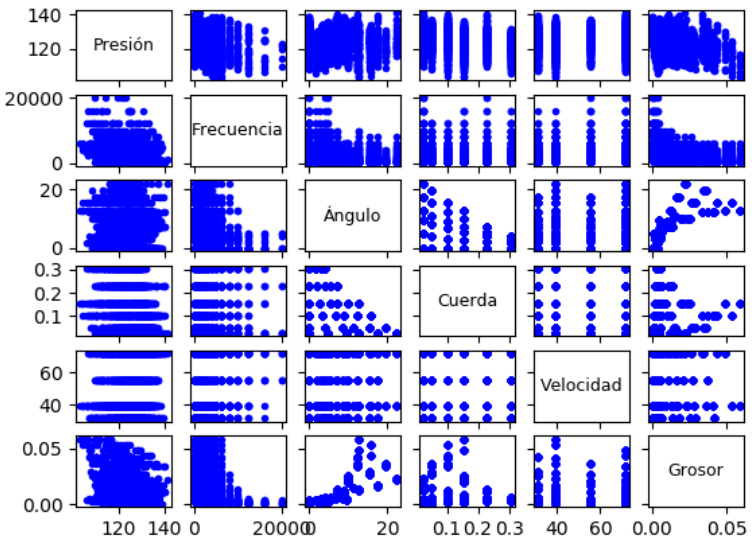


Figura 2: Datos del conjunto Airfoil Self-Noise.

   Por último, dicha base de datos constará de 1503 instancias.

2. **Preprocesado los datos: por ejemplo categorización, normalización, etc.**

   Para empezar, nos encargaremos de escalar los datos, pues cada uno de ellos se mueve en un rango distinto (como podemos ver en la imagen de arriba), y si trabajamos sobre dichos valores la predicción no será buena; utilizaremos el siguiente código:

```
# Escalamos los datos
min_max_scaler = preprocessing.MinMaxScaler()
X = min_max_scaler.fit_transform(X)
```

   Por otra parte, añadiremos nuevos atributos de la misma manera que en el caso de clasificación, aplicando características polinómicas, pues solo disponemos de 5 atributos. El código usado es:

```
# Usamos características polinomiales para añadir información
poly = PolynomialFeatures(2)
X = poly.fit_transform(X)
```

3. **Selección de clases de funciones a usar.**

   Usaremos el modelo de regresión Ridge, y la clase de funciones a usar será la clase de funciones lineales.

4. **Definición de los conjuntos de training, validación y test usados en su caso.**

   Como podemos ver, disponemos de un solo conjunto de datos, por lo que necesitaremos separar una parte de ellos para usarla en el futuro como datos test, pero antes de ello permutaremos los datos, pues los datos se nos dan en un orden preestablecido.

```
# Permutamos los datos antes de separar en train y test
random_state = check_random_state(0)
permutation = random_state.permutation(X.shape[0])
X = X[permutation]
y = y[permutation]

# Separamos los datos en train y test
X_train = X[0:2*X.shape[0]//3]
y_train = y[0:2*y.shape[0]//3]
X_test = X[2*X.shape[0]//3:X.shape[0]]
y_test = y[2*y.shape[0]//3:y.shape[0]]
```

   A continuación, usaremos validación cruzada de la misma manera que en clasificación, aunque aquí tomaremos un k mayor, pues el algoritmo no es tan lento como el de clasificación.

   Al contrario que en clasificación, no necesitaremos que las etiquetas estén distribuidas de manera uniforme, pues toman valores reales y no es necesario clasificarlas.

   Además, se ha aumentado el rango en el que se mueve la i. El código será el siguiente:

```
# Aplicamos K-FOLD
k = 8 # Número de subconjuntos a generar
kf = KFold(n_splits=k) # Inicializamos el K-Fold
mejor_alpha = 0
mejor_media = 0
alphas = []
scores = []

for i in range(-7,7): # Antes el rango era en [-5,5]
   suma = 0
   a = i*0.001 # En este caso, alpha crece linealmente

   for train_index, test_index in kf.split(X_train):
      X_train_, X_test_ = X_train[train_index], X_train[test_index]
      y_train_, y_test_ = y_train[train_index], y_train[test_index]

      [...] # Continúa con la regularización
```

5. **Discutir la necesidad de regularización y en su caso la función usada para ello.**

   Al igual que en el caso de la clasificación, aplicaremos regularización en el bucle de la validación, usando cada uno de los subconjuntos generados para ajustar el parámetro alpha del método de regresión de Ridge.

   En este modelo, el parámetro alpha denota la cantidad de regulación, no como en el caso de clasificación que el parámetro lamda denotaba la inversa de la cantidad de regularización.

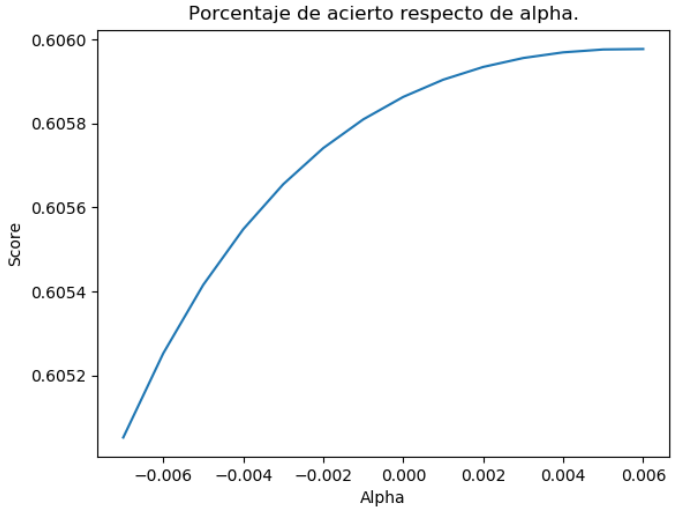
   Por tanto, ajustaremos este parámetro alpha con el siguiente código:

```
      [...] # Justo después de la validación

      # Aplicamos Ridge con el parámetro a
      lr = linear_model.Ridge(alpha = a)
```

```
            # Ajustamos el modelo a partir de los datos
            lr.fit(X_train_, y_train_)
            # Sumamos el porcentaje de acierto
            suma += lr.score(X_test_, y_test_)

        media = 1.0*suma/k

        # Guardamos alpha y la media para la gráfica
        alphas.append(a)
        scores.append(media)

        # Si mejora el porcentaje de acierto con el nuevo alpha
        if media > mejor_media:
            mejor_media = media
            mejor_alpha = a
```

De esta manera, obtendremos el valor de alpha, el cual será 0,6.

Si ploteamos los valores del porcentaje de acierto respecto de alpha, obtendremos la siguiente gráfica (la cuál alcanza su máximo en 0.6):



6. **Definir los modelos a usar y estimar sus parámetros e hyperparámetros.**

   Los distintos modelos usados consistirán en la regresión de Ridge variando el parámetro alpha asociado a ella.

   El único parámetro es el valor alpha, el cual va cambiando creando un nuevo modelo de regresión de Ridge. Respecto a los hyperparámetros, tenemos fijado el valor k = 8, el cual establece el número de subconjuntos a generar mediante la validación cruzada.

7. **Selección y ajuste modelo final.**

   Como modelo final, seleccionaremos la regresión de Ridge con el parámetro alpha que mejor porcentaje medio de acierto nos haya dado en la regularización, siendo alpha = 0,6.

   Para ajustar el modelo, haremos como en los anteriores casos:

```
# Ajustamos Ridge a partir de los datos
lr = linear_model.Ridge(alpha = mejor_alpha)
model = lr.fit(X_train, y_train)
```

8. **Discutir la idoneidad de la métrica usada en el ajuste.**

Para ver la idoneidad vamos a hacer uso de la función `score` del modelo de regresión de Ridge, que nos devuelve el coeficiente de determinación R^2, el cual toma valores entre 0 y 1, aunque debido a la implementación computacional, `score` puede llegar a devolver valores negativos. Cuanto más cercano sea el valor a 1 mejor explicará el modelo y por lo tanto menor sería el error de predicción.

9. **Estimacion del error E_{out} del modelo lo más ajustada posible.**

   Para obtener un E_{out} lo más ajustado posible, escogeremos el modelo final y calcularemos la precisión sobre los datos test generados anteriormente.

   Usaremos el siguiente código, en el que se calcula la precisión (score) respecto de los datos test, por lo que $E_{out} = 1 - score$.

```
# Calculamos el score con dicho ajuste
# con el conjunto test generado anteriormente
score = lr.score(X_test, y_test)
print()
print('Porcentaje de acierto sobre los datos test: ', score)
print('E_out=', 1-score)
input("Pulsa enter para continuar.")
```

   Ejecutando dicho código, obtendremos como resultados:

$$score = 0{,}6614911168181805 \rightarrow E_{out} = 0{,}3385088831818195$$

10. **Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.**

    El valor del precisión obtenido tras ajustar este modelo al conjunto de datos no es nada fiable, pues solo ajusta bien los datos un 66 % de las veces, por tanto no podemos considerar que es un buen ajuste que represente adecuadamente los datos muestrales.

    Además, podemos generar una gráfica de predicción, en la que la dispersión de los datos nos dará una idea de que nuestro modelo no es el más exacto para ajustar el conjunto de datos sobre el que trabajamos.