AA: Práctica 3

David Cabezas Berrido

Índice

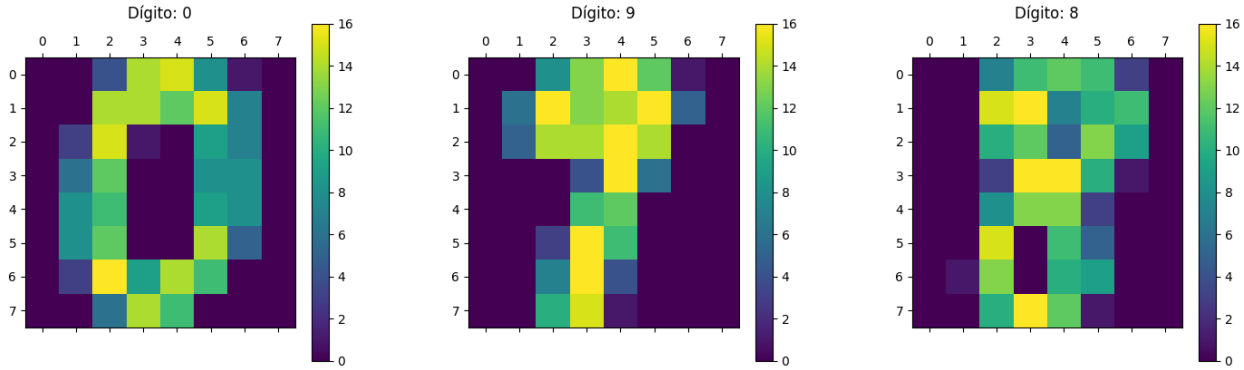
1. Clasificación de dígitos manuscritos

1.1. Problema

Se nos pide clasificar imágenes de dígitos escritos a mano para reconocer el dígito que representan (del 0 al 9). Disponemos de ejemplos clasificados para aprender, por lo que podemos enfocarlo como un problema de aprendizaje supervisado. Concretamente se trata de un problema de clasificación en el que tenemos 10 clases, los dígitos del 0 al 9.

En el archivo `opdigits.names` encontramos información sobre los datos que nos proporcionan. Constan de un conjunto con 3823 instancias para training y 1791 para test, cada instancia tiene 64 atributos que representan el número de bits coloreados (entre 0 y 16) en cada una de las 64 casillas que forman una cuadrícula de 8×8.

Podemos visualizar Ías instancias como matrices en lugar de vectores para comprender mejor el formato de los datos.

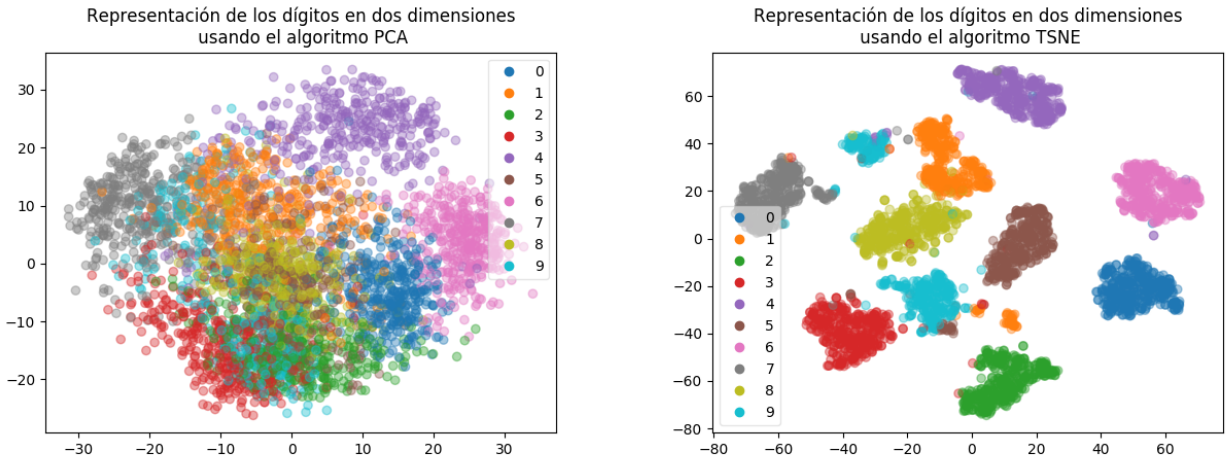


(a) Instancia correspondiente al 0 (b) Instancia correspondiente al 9 (c) Instancia correspondiente al 8

Figura 1: Algunas instancias de los datos

La población X constituye el conjunto de vectores de 64 enteros entre 0 y 16 que representan la cuadricula resultante de aplicar la trasformación antes comentada; el conjunto de clases Y constituye los posibles dígitos: 0,1,2,3,4,5,6,7,8,9; y la función objetivo f es la que asigna a cada vector de X la clase del dígito que representa.

Cabe preguntarnos si con los datos que tenemos podemos entrenar un buen modelo, pues se ha perdido parte de la información al agrupar los bits por cuadrículas. Para ello, podemos usar la función PCA (Principal Component Analysis) para proyectar las dos características que más me ayudan a distinguir los datos. Luego partimos de esa proyección y aplicamos algoritmo TSNE (T-distributed Stochastic Neighbor Embedding) para proyectar los datos en dos dimensiones de forma que para cada dato sus vecinos más cercanos queden proyectados cerca. Ambos algoritmos se encuentran progamados en *sklearn*.



(a) Proyección 2D mediante PCA

(b) Proyección 2D mediante T-SNE

Figura 2: Visualización en 2D de los datos

Sólo con las características principales ya podemos discernir ligeramente entre los datos de diferentes clases. Y si tenemos en cuenta todas ellas, los dígitos correspondientes a la misma clase se encuentran bastante agrupados exceptuando algunas instancias sueltas.

1.2. Conjuntos de training y test

Los datos que nos proporcionan vienen ya separados en conjuntos de training y test, y es importante que mantengamos esta división.

El motivo es que los datos de training corresponden a dígitos hechos a mano por 30 personas diferentes y los datos de test a los de otras 13 personas.

Siempre es importante que en training y en test se utilicen conjuntos disjuntos de datos para que E_{test} sea un estimador de E_{out} lo más representativo posible. Pero además en este problema es importante que los dígitos usados en test estén hechos por personas diferentes a las que generaron los dígitos de train, ya que es presumible que el modelo reconozca mejor los dígitos trazados por las mismas personas que trazaron los dígitos de entrenamiento.

Este hecho provoca que la estimación de E_{out} realizada con Cross-Validation sea demasiado optimista, por ser los datos de validación correspondientes a las mismas personas que los de entrenamiento.

1.3. Clases de funciones a usar y Preprocesamiento

La clase de funciones que usaremos son los polinomios de grado 2, ya que permiten obtener un modelo mucho más complejo y potente que utilizar simplemente características lineales. Para añadir características polinomiales usamos PolynomialFeatures.

El motivo por el que presento esta sección junto con procesamiento es que la introducción de características polinomiales eleva el número de variables al grado del polinomio, esto es computacionalmente costoso y provoca que acabemos con demasiadas características. De hecho, introducir características polinómicas de grado mayor que 2 es ya demasiado costoso, por lo que no usamos grados más altos. Por este motivo he decidido añadir la indroducción de características polinomiales al preprocesamiento.

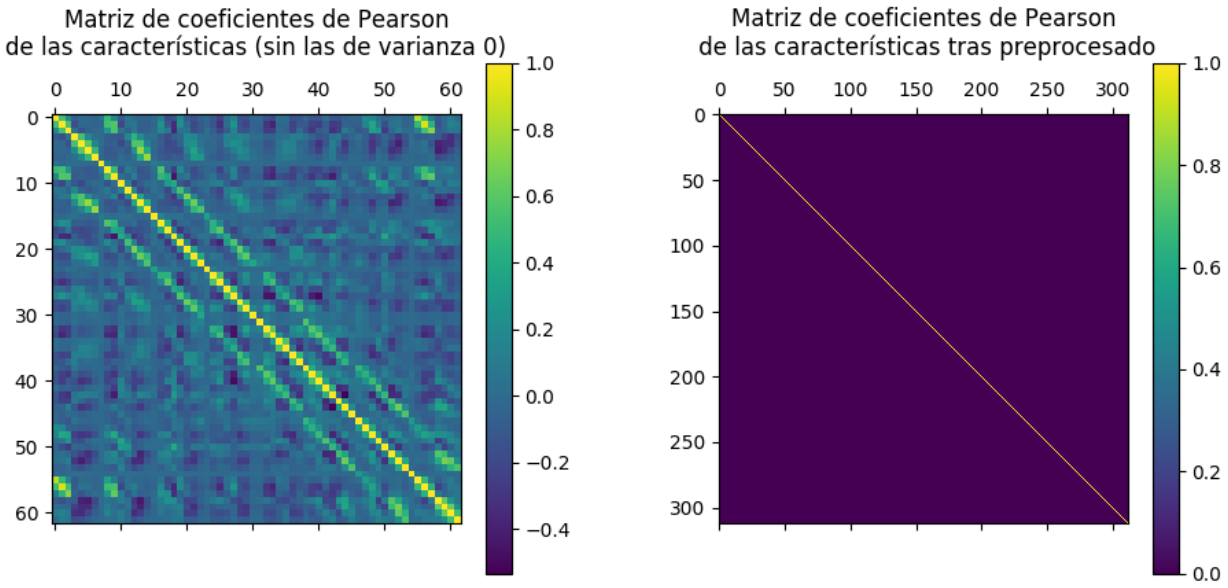
Para el preprocesamiento, creamos un Pipeline que primero elimina con Variance Threshold las características con varianza menor que un umbral, en este caso 0.005. Estas características apenas ayudan a distinguir las instancias de la muestra. En segundo lugar añade características de grado 2, ahora el número de características es igual al cuadrado de las que quedaban tras la primera selección en lugar de 64^2. A continuación, realiza una estandarización

con StandardScaler y escala las variables para dejar todas con media 0 y varianza 1. Por último utiliza otra vez PCA, pero esta vez en lugar de proyectar un número fijo de variables, selecciona el menor número posible que expliquen cierto porcentaje de la variabilidad de la muestra, en este caso un 97.5 %.

Ajustamos el Pipeline con los datos de train y aplicamos las transformaciones tanto a los de train como a los de test. Partíamos de 64 características y nos quedamos con 312. Teniendo en cuenta que $64^2 = 4096$, sacrificando un pequeño porcentaje de la información nos quedamos con un conjunto de variables bastante manejable.

Tanto el objeto Pipeline como los que realizan cada paso del preprocesado se encuentran en *sklearn*.

Para apreciar el resultado del preprocesado podemos visualizar la matriz de coeficientes de correlación de Pearson, que indica la medida en que unas características determinan otras. Si una característica está determinada por el resto, se podría eliminar sin perder información. Por tanto interesa que esta matriz sea diagonal, que cada variable se determine únicamente a ella misma.



(a) Antes del preprocesado

(b) Después del preprocesado

Figura 3: Matrices de coeficientes de correlación de Pearson

Como podemos ver, fuera de la diagonal todos los coeficientes son practicamente 0 como queríamos.

1.4. Métricas

Mirando el fichero `opdigits.names` percibimos que el número de ejemplos de cada clase está balanceado tanto en training (entre 376 y 389) como en test (entre 174 y 183). En esta situación, la precisión (accuacy, la proporción de elementos bien clasificados) es un métrica adecuada de la bondad del modelo.

Ésta será la métrica que estimaremos con validación cruzada y compararemos con la que consiga en el conjunto de test.

También podemos visualizar la matriz de confusión (figuras 4 y 6). Es una matriz cuadrada de orden el número de clases con números naturales como entradas, en la que el valor de la posición (i, j) representa el número de ejemplos de la clase i-ésima que han sido clasificados por el modelo como elementos de la clase j-ésima. Claramente interesa que la matriz de confusión sea diagonal, ya que las entradas de la forma (i, i) representan éxitos a la hora de clasificar y las entradas (i, j) con $j \neq i$ representan errores. De hecho, la accuracy se obtiene dividiendo la traza de la matriz de confusión (el número de ejemplos bien clasificados) entre el número total de ejemplos.

1.5. Técnica de ajuste del modelo

Aunque la métrica usada es la precisión, el modelo que usamos es el de Regresión Logística Multietiqueta para estimar la probabilidad de pertenencia a cada clase (la regla de clasificación será SoftMax), minimizamos la Pérdida Logarítmica.

$$E(\mathbf{w}) = E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \frac{-1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \ln \sigma(\mathbf{w}_k^T \mathbf{x}_n)$$

Donde cada $\mathbf{w}_k$ es la fila k-ésima de una matriz de pesos $\mathbf{w}$ de dimensión $K \times d$, con K el número de clases (10) y d el número de características (312). Y las etiquetas y_n están codificadas como vectores one-hot ($y_{nk} = 1$ si y_n corresponde a la clase k y 0 en caso contrario).

Para minimizar esta función de pérdida utilizamos el algoritmo de Gradiente Descendente Estocástico, en otras ocasiones ya hemos comprobado su eficiacia a la hora de minimizar este tipo de funciones. De hecho el motivo de que usemos esta función de pérdida y no la proporción de fallos (1-accuracy) es la comodidad que tiene esta función para minimizarla con esta técnica. Puedo calcular fácilmente el gradiente respecto a cada una de las filas de la matriz $\mathbf{w}$.

$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \frac{1}{N} \sum_{n=1}^{N} (\sigma(\mathbf{w}_j^T \mathbf{x}_n) - y_{nj}) \mathbf{x}_n$$

Necesito elegir un learning rate η y un tamaño de minibatch adecuados, estos son hiperparámetros del modelo que discutieremos más adelante.

1.6. Regularización

Las funciones cuadráticas que hemos introducido añaden un número elevado de características y bastante complejidad al modelo, lo que lo hace propenso al sobreajuste. Para evitar esto, debemos introducir algún tipo de regularización.

En el preprocesado hemos eliminado bastantes atributos que no aportaban apenas información sobre la variabilidad de la muestra, luego cabe esperar que la mayoría de atributos que hemos seleccionado sean relevantes. Es por ello que utilizaremos la Regularización Ridge, reduciendo el cuadrado de la norma euclídea o norma de Frobenius de la matriz de pesos $\mathbf{w}$. La función de pérdida queda ahora

$$E_{aug}(\mathbf{w}) = E(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

donde $\|\mathbf{w}\|_2^2 = \sum_{k=1}^{K} \sum_{j=1}^{d} \mathbf{w}_{kj}^2$ y λ es un hiperparámetro del modelo sobre el que hablaremos más adelante.
Esta fórmula es fácil de derivar, obteniendo

$$\nabla_{\mathbf{w}_j} E_{aug}(\mathbf{w}) = \nabla_{\mathbf{w}_j} E(\mathbf{w}) + \lambda 2 \mathbf{w}_j$$

1.7. Modelos

Como ya hemos comentado, usaremos un modelo de Regresión Logística multiclase, ya que es el más adecuado de los modelos lineales que conocemos para resolver un problema de clasificación no binario.

El modelo nos permite estimar para una instancia la probabilidad de pertenencia a cada clase $j = 1, \ldots, K$ mediante la fórmula:

$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{w}_j \mathbf{x}}}{\sum_{k=1}^{K} e^{\mathbf{w}_k \mathbf{x}}}$$

A la hora de predecir utilizaremos la regla SoftMax, que para una instancia $\mathbf{x}$ predice la clase con mayor probabilidad.

Para aprovechar varias funcionalidades de *sklearn*, implementamos nuestro propio objeto estimador (heredando de la clase BaseEstimator) que depende de los hiperparámetros que hemos comentado anteriormente. Debemos implementar como mínimo los métodos `fit` y `predict`, `fit` recibe los datos de entrenamiento y ajusta la matriz de pesos $\mathbf{w}$ minimizando la Pérdida Logarítmica aumentada con SGD; `predict` recibe una instancia y aplica la regla SoftMax para determinar la clase a la que pertenece.

Para evitar que el entrenamiento sea computacionalmente muy costoso, he limitado el número de evaluaciones totales a 50000, a la hora de aprender los datos se agrupan según el tamaño de minibatch. Esto quiere decir que si el tamaño de minibatch es el doble, se realizarán la mitad de iteraciones.

Esto no significa que no tengamos que discutir sobre el mejor modelo a usar, ya que el modelo depende de varios hiperparámetros que determinan su comportamiento y debemos estimar valores adecuados para ellos.

1.8. Estimación de hiperparámetros y selección del modelo

Para decidir el mejor modelo debemos estimar los hiperparámetros, para ello utilizamos GridSearchCV, que realiza una búsqueda exhaustiva en rejilla probando todas las combinaciones de hiperparámetros en un rango que determinamos. Para decidir qué combinación de parámetros es mejor implementamos el método `score` en nuestro estimador, que calcula el accuracy sobre un conjunto de datos. El algoritmo evalúa cada combinación usando Cross Validation con 5 subdivisiones, esto es tremendamente costoso (incluso paralelizando) por lo que lo omitimos en la versión final del código (se controla con la variable booleana PARAMSELECT).

Tras algunas horas de búsqueda, la combinación que mejor score ha presentado entre las que he probado es $\lambda = 0.007391304347826088$, $\eta = 0.001$, tamaño de minibathch $= 1$. Parece que lo más efectivo es hacer un gran número de iteraciones con un sólo dato y una tasa de aprendizaje baja. Con estos hiperparámetros, la precisión media de las 5 validaciones ha sido 0.9675642473394245, más que aceptable.

Por tanto, fijamos estos hiperparámetros y seleccionamos éste como el mejor modelo. Primero medimos su efectividad sobre el conjunto de training.

Pérdida logarítmica (aumentada) en la muestra: 0.6220026439186417
Precisión sobre train: 0.9777661522364635

Matriz de confusión en la prueba sobre el conjunto de entrenamiento:

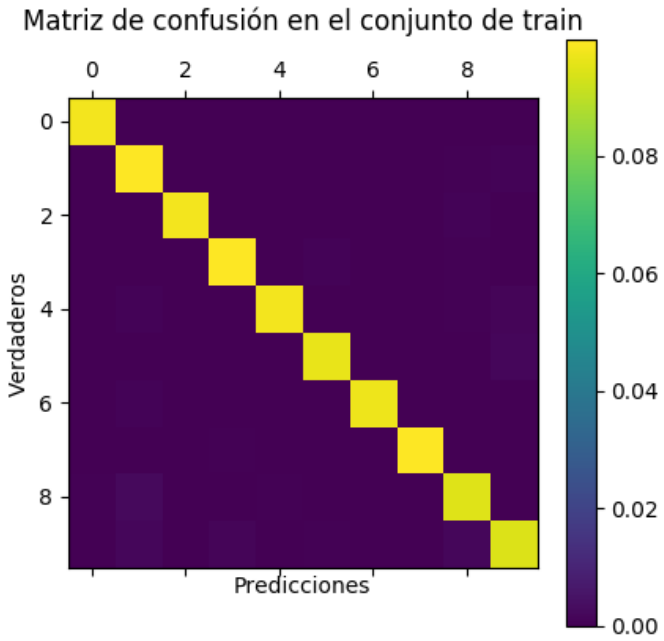


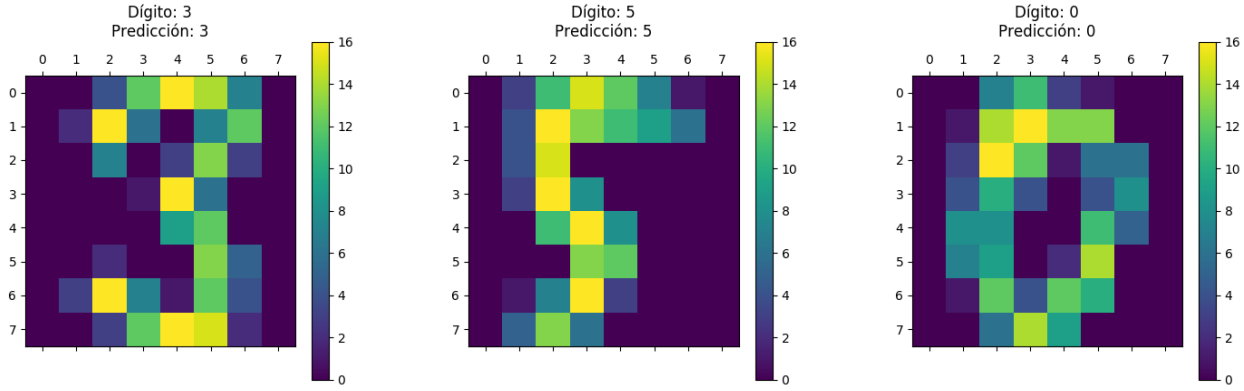
Figura 4: Matriz de confusión en la prueba con los datos de entrenamiento

Tiene una precisión muy cercana a 1, más que su score en la selección de modelos, ya que estos datos son los que he usado para entrenar el modelo. La matriz de confusión es prácticamente diagonal, falla sobre todo en algunas instancias de los dígitos 8 y 9 que ha clasificado como unos.

1.9. Estimación de E_{out} por validación cruzada y comparación con E_{test}

En lugar de error he considerado la precisión, que es una medida de bondad. Usando la función `cross_val_score` de *sklearn* obtenemos una estimación por validación cruzada de la precisión del modelo. Realiza 10 subdivisiones del conjunto de entrenamiento, por lo que necesita entrenar y validar 10 veces. Este proceso es algo lento, pero la función permite paralelizarlo. La precisión media que he obtenido en las 10 validaciones es $E_{cv} = 0.964162098615231$.

Primero sacamos tres muestras aleatorias del conjunto de test, en las tres el modelo ha dado con la clase correcta.



(a) Instancia correspondiente al 3 (b) Instancia correspondiente al 5 (c) Instancia correspondiente al 0

Figura 5: Algunas instancias de los datos

Evaluamos ya el modelo sobre el conjunto de test.

Pérdida logarítmica (aumentada) en test: 0.6327756996166487
Precisión sobre test: 0.9549248747913188

Matriz de confusión en la prueba sobre el conjunto de test:

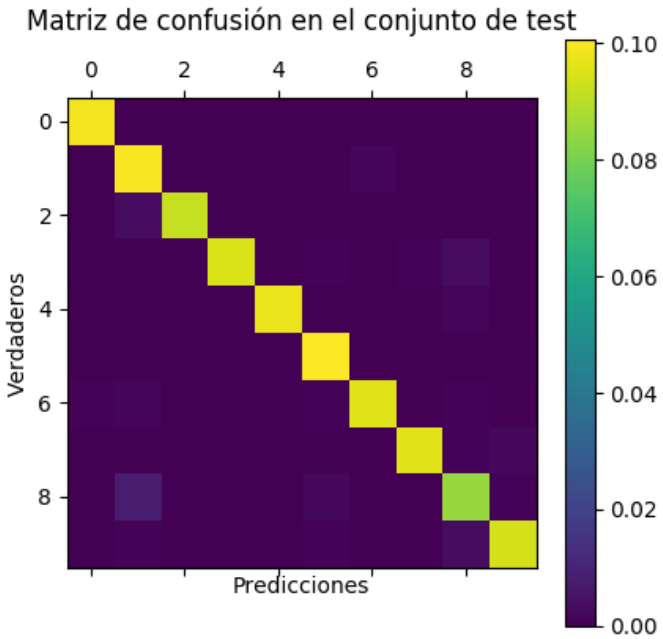


Figura 6: Matriz de confusión en la prueba con los datos de test

La precisión obtenida sobre el conjunto de test es ligeramente menor que la estimación realizada por validación cruzada. Esto se debe a la razón que comenté en la sección 1.2 de que la validación y el entrenamiento se realizan sobre datos correspondientes a dígitos trazados por las mismas 30 personas y los dígitos tienden a ser más parecidos entre ellos que los de test, que están trazados por otras 13 personas diferentes.

No obstante, la precisión obtenida es aceptable. Tampoco podemos optar a mucha más precisión puesto que hemos renunciado a algo más del 2.5 % de información sobre la variabilidad de la muestra para simplificar los datos durante el preprocesamiento.

El modelo falla sobre todo al clasificar algunos ochos como unos, lo que también ocurría en la prueba con los datos de entrenamiento.

1.10. Conclusiones