Aprendizaje Automático: Proyecto Final
Devanagari Handwritten Characters

David Cabezas Berrido y Patricia Córdoba Hidalgo

Índice

Nota:
Durante la resolución de los problemas, hay momentos en los que hemos tenido que volver atrás y rectificar decisiones. Intentamos plasmar los motivos que llevan a cada decisión, pero no hemos reflejado todo en el código, sólo la versión definitiva.

Sí que hay secciones de código (selección de parámetros, por ejemplo) que hemos considerado interesante mantenerlas aunque no se ejecuten. Se pueden ejecutar dichas funciones en el cuaderno (en principio sólo se declaran), pero no es nada recomendable (los tiempos de ejecución son muy altos) así que en la versión definitiva intentamos mantener el equilibrio entre representar una gran proporción del trabajo que hemos realizado y que la ejecución sea rápida y cómoda para ir siguiéndola.

Enfocamos este proyecto como la simulación de un problema real: sólo basamos nuestras decisiones en el desempeño sobre el conjunto de training, con la excepción de que disponemos de un conjunto de test para contrastar y valorar los resultados obtenidos.

1. Problema

El problema a resolver consiste en clasificar caracteres de la escritura Devanagari. Nuestros datos son imágenes de estos símbolos escritos a mano, cada uno de ellos con una etiqueta especificando el símbolo que representa la imagen.

Inicialemente, nuestro espacio de características $\mathcal{X}$ está formado por imágenes de 32×32 píxeles cada una, con un marco de 2 píxeles por cada uno de los 4 lados. El conjunto de etiquetas, Y son los 46 caracteres que consideramos, 36 letras y 10 dígitos (del 0 a 9). La función objetivo f que buscamos aproximar es aquella que a una cuadrícula de píxeles representando un símbolo Devanagari manuscrito le haga corresponder la clase del símbolo correspondiente.

2. Dataset. Conjuntos de train y test

El conjunto de datos de los que disponemos
https://archive.ics.uci.edu/ml/datasets/Devanagari+Handwritten+Character+Dataset consta de 92000 instancias (imágenes etiquetadas), 2000 de cada una de las 46 clases. Los datos vienen divididos en train: 78200 instancias (85 %), 1700 por clase; y test: 13800 instancias (15 %), 300 por clase. No existen datos perdidos y las clases están perfectamente balanceadas.

Existen algunos artículos y proyectos relativos a este dataset, por lo que mantener esta división entre train y test nos permitirá comparar los resultados que logren nuestros modelos y procesamiento con los resultados de otros diseñados por terceros.

En la descripción del dataset se informa de que estos datos proceden de documentos escritos, pero desconocemos su procedencia y sus autores. En problemas de reconocimiento de símbolos manuscritos existe la dificultad de que un modelo aprenda a reconocer símbolos de un único autor o un conjunto de autores, lo que conlleva sobreajuste y a estimaciones por validación poco fiables, ya que la muestra de entrenamiento no termina de ser representativa y el modelo se adapta a ese sesgo. En nuestro caso, desconocemos si los datos de train y test corresponden a símbolos trazados por distintas personas o no, con lo que tenemos otra razón más para respetar la partición de conjuntos de train y test existente.

Extraemos un conjunto de validación del 20 % de los datos de entrenamiento para consultar la eficacia práctica de ciertas decisiones que tomamos durante el preprocesamiento y comparar la bondad de los modelos. Esto nos deja con 62560 (80 %) datos para entrenar cada alternativa y 15640 datos para validar. Debido a la abundancia de datos, podemos esperar que los scores obtenidos al validar sean representativos de la bondad real de un modelo o preprocesamiento, así que decidimos no usar validación cruzada, ya que es computacionalmente muy costosa y el ajuste de los modelos es lento debido a la cantidad de datos.

Para ello usamos la función `train_test_split` de *sklearn* e indicamos mediante la opción `stratify` que queremos preservar la proporción de elementos de cada clase en cada uno de los conjuntos, de forma que sigan perfectamente balanceadas.

2.1. Formato de los datos

En el fichero `png_to_np.py`, guardamos las imágenes, originalmente en formato `png`, como array. Para ello leemos cada imagen como array de escala de grises usando la función `imread` de *matplotlib* y eliminamos los dos píxeles de marco por cada lado, quedándonos con matrices de 28×28 de valores flotantes entre 0 y 1 representando la intensidad de gris en cada pixel. El resultado es guardado en disco, lo hacemos con la función `saveGrey`, que usa la función `savez_compressed` de *numpy* para almacenarlo en formato `npz`.

Este fichero solo se ejecuta una vez para cambiar el formato de los datos. Tras esto, podemos usar los datos guardados en disco para las sucesivas ejecuciones del código. Proporcionamos los datos ya transformados.

3. Métrica de error

La métrica de error que usaremos para contrastar diferentes modelos, hiperparámetros y alternativas será la accuracy, que es la proporción de etiquetas que predice correctamente el modelo. Usaremos esta métrica para ver realmente cuál es el desempeño del modelo y cómo de bien clasifica las etiquetas. Como los datos están balanceados, es una métrica adecuada de la bondad del modelo, además de fácil de interpretar.

También podemos visualizar la matriz de confusión. 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.

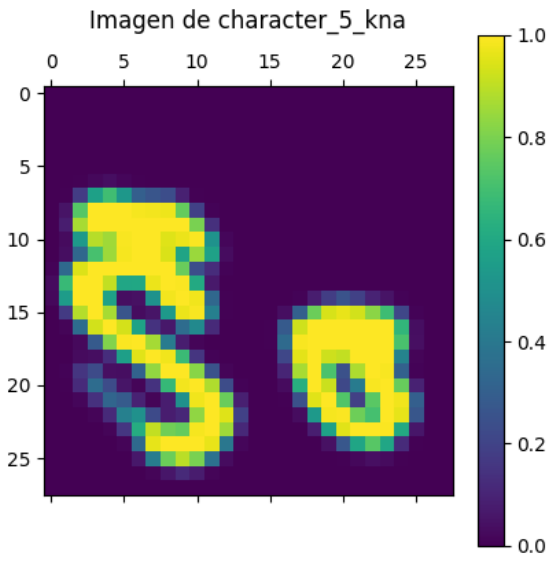
4. Preprocesamiento

El preprocesamiento se realiza imagen a imagen, siendo el resultado sólo dependiente de la propia imagen y por tanto paralelizable. Para cada imagen realizamos dos operaciones: centrado y reescalado, y downsampling. Tras este proceso, nos quedan 196 características (14×14) y comprobamos con la función `VarianceThreshold` de *sklearn* que no hay características con varianza 0 en el conjunto de train (todas aportan algo de información). Consideramos que no es necesario normalizar las variables, ya que todas tienen la misma naturaleza (intensidad de gris en un píxel) y la misma escala (entre 0 y 1).

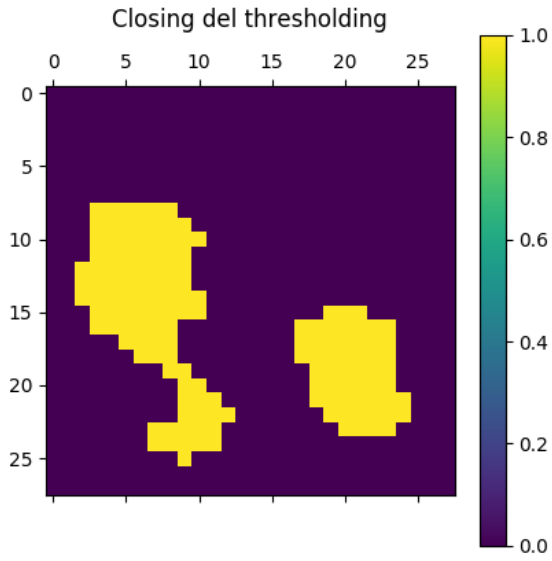
4.1. Centrado y reescalado

Esta operación la realiza la función `centerAndResize`. Ante una imagen, calculamos un umbral con la ayuda del método de Otsu para thresholding (usando la función `threshold_otsu` de la librería *skimage*).

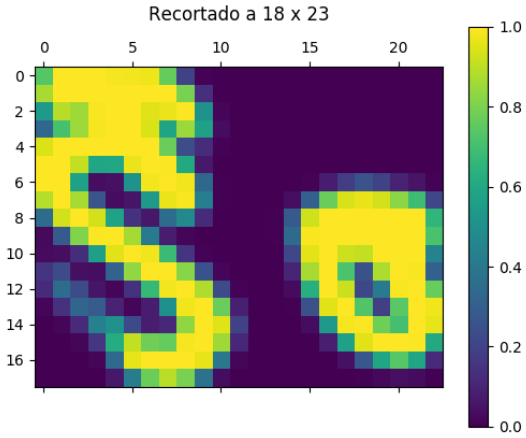
Para calcular la caja englobante del carácter, usamos el closing (función `closing` de *skimage*) de la imagen resultante de considerar los píxeles con intensidad superior a este umbral. Recortamos el exterior de la caja y reescalamos la imagen a `WIDTH` $\times$ `WIDTH` para trabajar con un tamaño de imagen unificado (función `resize` de *skimage*).



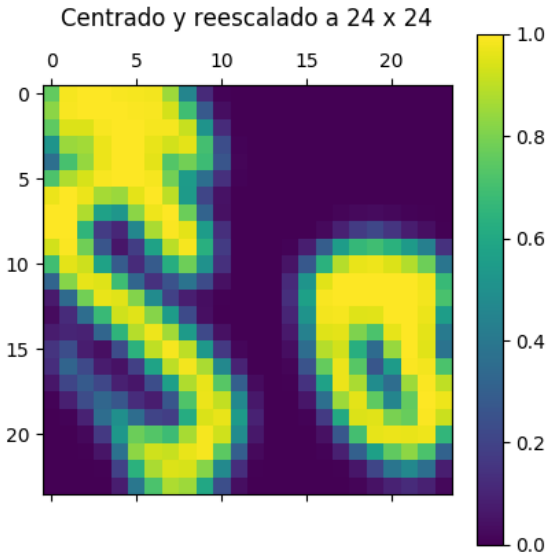
(a) Muestra antes del preprocesado



(b) Thresholding y closing



(c) Recortado de la caja englobante



(d) Reescalado

Figura 1: Proceso de centrado y reescalado sobre una instancia correspondiente al carácter kna (usando `WIDTH` = 24)

4.2. Downsampling

Para reducir la dimensionalidad, realizamos un downsampling o reducción por bloques de la imagen, agrupando cada bloque de 4 píxeles en uno usando la media de sus valores. Esta reducción se lleva a cabo con la función `block_reduce` de *skimage*.

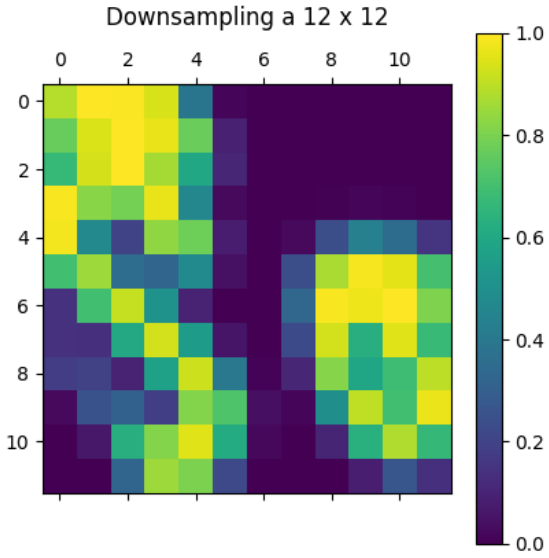


Figura 2: Resultado del preprocesamiento (tras combinar centrado y reescalado con downsampling usando `WIDTH` = 24)

4.3. Utilidad del preprocesamiento

Queremos comprobar que el preprocesamiento que hemos realizado realmente ayudará al posterior entrenamiento de los modelos y mejorará los resultados. Usando los 3 modelos escogidos para el ajuste con sus respectivos hiperparámetros previamente seleccionados (hablaremos de esto más detenidamente en secciones posteriores) veremos la accuracy obtenida sobre nuestro subconjunto de validación en cada una de las siguientes comparaciones.

Durante el preprocesamiento, hemos tomado dos decisiones que podrían afectar al desempeño posterior de los modelos. Estas dos decisiones son: el valor de la variable `WIDTH` y usar o no `block_reduce` para reducir la dimensionalidad.

Para el valor de la variable `WIDTH` hemos consderado 28, ya que muchos caracteres ocupan toda la imagen y no se llegan a recortar, y 24, para intentar encontrar un tamaño intermedio entre las que se recortan y las que no.

La decisión de reducir dimensionalidad con `block_reduce` es arriesgada porque se produce una pérdida de información de los datos. Tenemos que contrastar si esa pérdida de información es significativa atendiendo a las ventajas que supone hacerla: simplifica el problema (menor dimensionalidad) y reduce los tiempos de ejecución.

Los resultados obtenidos (con `block_reduce`=True) son:

Modelos	`WIDTH` = **28**	`WIDTH` = **24**
Random Forest	0.9195652173913044	0.9117647058823529
Multilayer Perceptron	0.8879156010230179	0.885230179028133
Regresión Logística	0.7273657289002557	0.7289002557544757
Media (no lineales)	0.903740409	0.898497442
Media	0.844948849	0.841965047

Los resultados obtenidos (con `block_reduce`=False) son:

Modelos	`WIDTH` = **28**	`WIDTH` = **24**
Random Forest	0.9115728900255754	0.9159846547314578
Multilayer Perceptron	0.8574168797953964	0.8688618925831202
Regresión Logística	Error de memoria	Error de memoria
Media (no lineales)	0.884494885	0.892423274

A la vista de los resultados obtenidos, hemos decidido que merece la pena usar `block_reduce` y el valor de `WIDTH` que elegimos es 28. Esta configuración presenta la mayor media y el mayor máximo de las accuracy en las validaciones de los modelos.

Los resultados tras el preprocesamiento comparándolos con los datos en crudo son:

Modelos	**Antes del preprocesamiento**	**Después del preprocesamiento**
Random Forest	0.9070971867007672	0.9195652173913044
Multilayer Perceptron	0.8464194373401535	0.882161125319693
Regresión Logística	Error de memoria	0.7273657289002557

Podemos observar que con el preprocesamiento hemos conseguido reducir notablemente la dimensionalidad de los datos: pasando de $28 \times 28 = 784$ variables a $14 \times 14 = 196$. Además, mejoramos en cierta medida la precisión de los modelos.

5. Visualización de los datos mediante proyección 2D

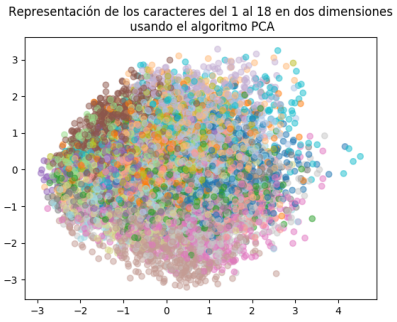
Podemos proyectar los datos en dos dimensiones para intuir de un simple vistazo hasta que punto la información de la que disponemos nos permitirá discernir unas clases de otras, y así conocer (entre otras cosas) si se ha perdido información durante el preprocesamiento. Existen algoritmos para esto: PCA (Principal Component Analysis), que proyecta las dos direcciones que más nos ayudan a discernir los datos. También TSNE (T-distributd Stochastic Neighbor Embedding), un algoritmo iterativo (lo lanzamos con la salida de PCA como proyección inicial) que proyecta los datos en dos dimensiones de forma que para cada dato sus vecinos más cercanos queden proyectados cerca.

El gran número de datos hace que estas representaciones tarden mucho tiempo en generarse y estén muy cargadas, por lo que sólo representamos el 40 % de los datos (elegidos aleatoriamente). Para ello usamos la función `train_test_split` de *sklearn* e indicamos mediante la opción `stratify` que queremos preservar la proporción de

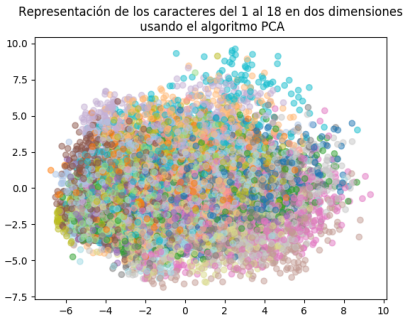
elementos de cada clase en cada uno de los conjuntos. Debido a la abundancia de datos, este 40 % es suficientemente representativo de la distribución de los datos.

Además, debido al elevado número de clases, no podemos representar cada una con un color diferente de forma que sean fácilmente distinguibles, por eso las representamos separadas en 3 gráficas diferentes: caracteres del 1 al 18, caracteres del 19 al 36 y dígitos (del 0 al 9). Nos interesa ver si las instancias de la misma clase están cerca entre sí, no la localización en el espacio, por tanto omitimos la leyenda (que además sería extremadamente larga y taparía el gráfico).

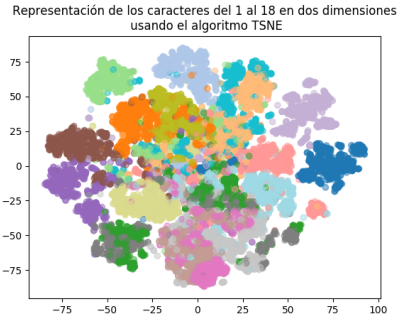
Generamos estas representaciones antes y después del preprocesado, para advertir si la reducción de dimensionalidad conlleva una pérdida considerable de información.



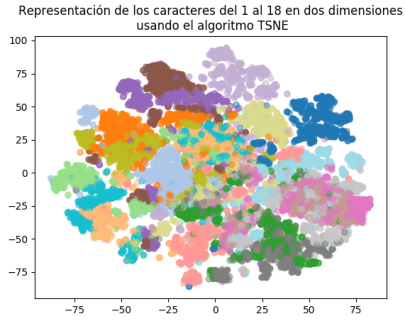
(a) PCA. Con preprocesamiento



(b) PCA. Sin preprocesamiento

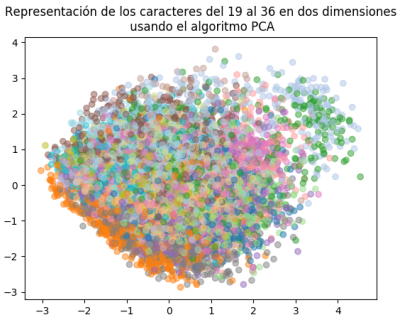


(c) TSNE. Con preprocesamiento

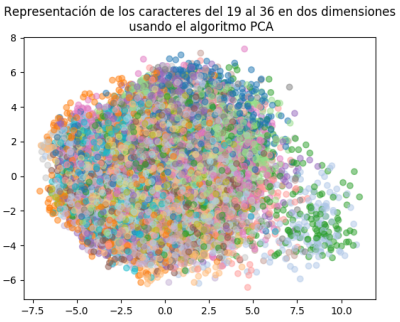


(d) TSNE. Sin preprocesamiento

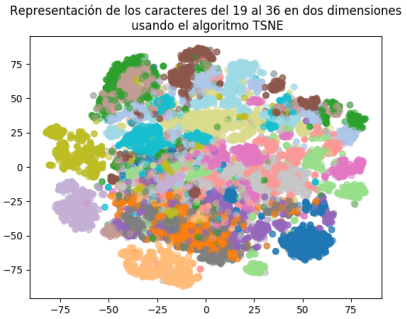
Figura 3: Proyecciones en 2D de las clases correspondientes a los caracteres del 1 al 18



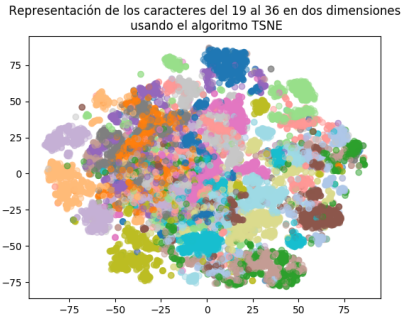
(a) PCA. Con preprocesamiento



(b) PCA. Sin preprocesamiento

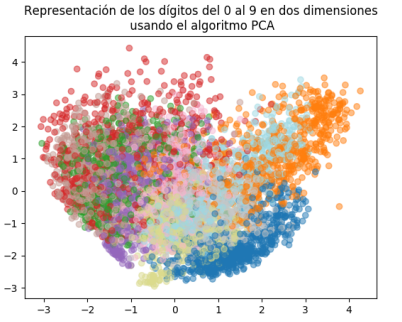


(c) TSNE. Con preprocesamiento

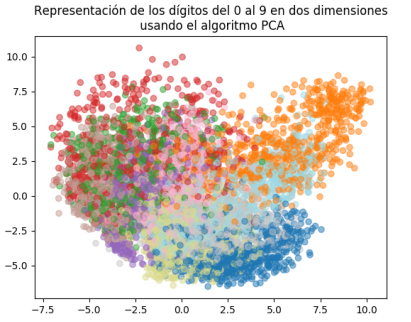


(d) TSNE. Sin preprocesamiento

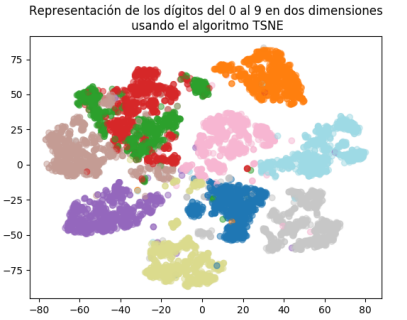
Figura 4: Proyecciones en 2D de las clases correspondientes a los caracteres del 19 al 36



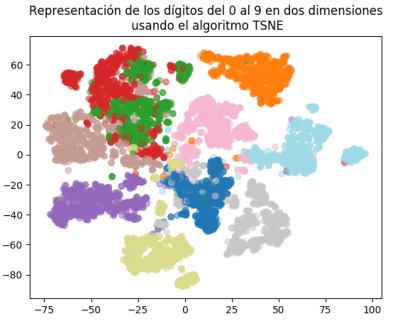
(a) PCA. Con preprocesamiento



(b) PCA. Sin preprocesamiento



(c) TSNE. Con preprocesamiento



(d) TSNE. Sin preprocesamiento

Figura 5: Proyecciones en 2D de las clases correspondientes a los dígitos del 0 al 9

Aunque existan algunas clases entremezcladas y algunas instancias dispersas, en las proyecciones TSNE se percibe que los datos correspondientes a la misma clase en general están cerca. En las proyecciones con PCA, se distinguen ligeramente algunas clases del resto, pero la mayoría están totalmente entremezcladas.

Comparando las proyecciones antes y después del preprocesamiento, no parece que exista una pérdida de información significativa. Las clases están mezcladas prácticamente en la misma medida.

No obstante, estas representaciones no deben ser nuestra única justificación para concluir que el preprocesamiento es adecuado. Hay que tener en cuenta que no estamos representando todas las clases en el mismo gráfico, luego hay parejas de clases que no estamos comparando entre sí. Es por ello y por la dificultad de interpretar los gráficos que basamos el grueso de nuestros argumentos de la sección anterior en resultados empíricos obtenidos por validación.

6. Modelo lineal: Regresión Logística

Como modelo lineal elegimos Regresión Logística por ser el modelo que mejor se adapta a clasificación multietiqueta de los que conocemos. Este modelo viene implemendado en la función `SGDClassifier` de *sklearn*, usando como función de pérdida la pérdida logarítmica. A la hora de predecir, este modelo estima la probabilidad de pertenencia de un dato a cada clase y le asigna la clase con mayor probabilidad (SoftMax).

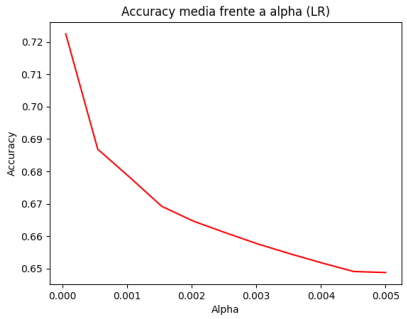
Para intentar compensar la menor potencia de este modelo, incorporamos algunas características polinómicas. Debido al elevado número de variables, no incorporamos los términos cruzados (provoca error de memoria aun con polinomios de grado 2), sólo los cuadrados, cubos y cuartas potencias de cada una de las características originales, multiplicando el número de variables por 4. Esto no sería viable de no haber realizado la reducción de dimensionalidad por downsampling (provoca error de memoria). Además, la documentación de esta función recomienda que las características presenten varianza 1 y media 0 para una convergencia más rápida del SGD cuando se utiliza el learning rate por defecto. Realizamos esto ajustando un StandardScaler (de *sklearn*) a los datos de training y aplicándolo para transformar tanto éstos como los de test.

Durante el entrenamiento del modelo recibimos warnings acerca de la convergencia, por lo que tuvimos que incrementar el número máximo de iteraciones (`max_iter`) de 1000 a 2000.

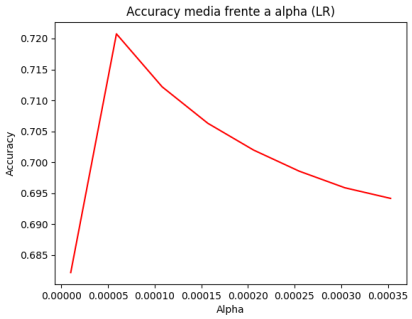
6.1. Estimación de hiperparámetros

El hiperparámetro que ajustaremos del modelo lineal es α (el coeficiente que penaliza en la regularización Ridge). En la implementación del modelo, el learning rate (seleccionando la opción `'optimal'`, que es la que ofrece por defecto la implementación) se calcula: $\eta = \frac{1}{\alpha(t+t_0)}$, con t_0 escogido por una heurística propuesta por Leon Bottou y t denotando la etapa.

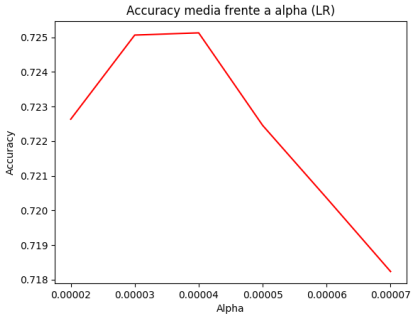
Para estimar el valor de α, que necesariamente tiene que ser distinto de cero para usar dicho learning rate, ajustamos el modelo con varios valores. En la primera gráfica, podemos observar que la accuracy del modelo disminuye al aumentar el valor de alpha, así que realizamos mediciones con valores menores. En la segunda, vimos que el pico se alcanzaba entre 0.00001 y 0.0001, por lo que volvimos a repetir el experimento ahora entre 0.00002 y 0.00007. Pudimos comprobar que el máximo se alcanza entre 0.00003 y 0.00004, así que volvimos a ajustar el modelo con valores de α comprendidos entre 0.000025 y 0.000045. En esta gráfica podemos observar que la diferencia de accuracy entre estos valores es bastante menor. Como se alcanza el mayor valor de accuracy en $\alpha = 0.00004$, éste es el valor que tomaremos para el ajuste.



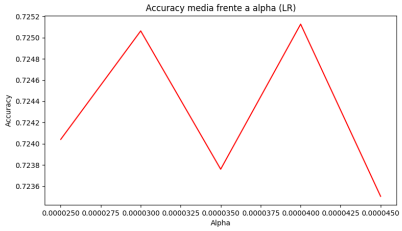
(a) Entre 0.00005 y 0.005, 11 mediciones



(b) Entre 0.00001 y 0.00035, 10 mediciones



(c) Entre 0.00002 y 0.00007, 6 mediciones



(d) Entre 0.000025 y 0.000045, 5 mediciones

Figura 6: Estimación del parámetro α

6.2. Función de pérdida y regularización

Dada una muestra de tamaño N, donde cada dato tiene d características, y un problema de clasificación multietiqueta con K etiquetas diferentes, la función de pérdida de este modelo es la pérdida logarítmica:

$$E(w_1, \ldots, w_k) = -\ln L(Y|w_1, \ldots, w_k) = -\sum_{n=0}^{N}\sum_{k=0}^{K} y_{nk} \ln \sigma(w_k^T x_n)$$

donde x_n el vector de características de la instancia n-ésima, w_k es la fila k-ésima de una matriz de pesos w de dimensión $K \times d$ e $y_{nk} = 1$ si la instancia n-ésima pertenece a la clase k y 0 en caso contrario.

La implementación de este modelo minimiza esta función de pérdida usando gradiente descendente estocástico.

Para disminuir la variabilidad del modelo, usamos regularización Ridge. Es adecuada porque todas las características son relevantes ya que tienen la misma naturaleza y durante el preprocesado hemos eliminado aquellas que podrían no aportar demasiada información.

Así, el error que minimizamos en regularización es $E_{aug} = E(w_1, \ldots, w_k) + \alpha\|w\|^2$, donde $\|.\|^2$ denota la norma de Frobenius.

Como hemos discutido en la sección de selección de hiperparámetros, usaremos $\alpha = 0.00004$.

7. Random Forest

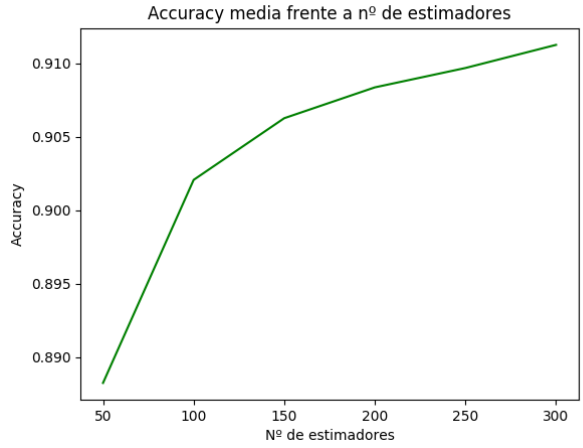
El modelo de random forest construye `n_estimators` árboles de decisión usando la totalidad de la muestra para la construcción de cada uno, pero sólo un subconjunto de las características para disminuir la correlación entre los árboles. Usamos la raíz cuadrada del número de características, que es un valor adecuado según lo estudiado en teoría y además, el valor que recomienda la implementación de *sklearn*. Tras esto, hace una media de dichos árboles para controlar el overfitting y reducir la variabilidad.

Elegimos este modelo por su capacidad para conseguir un bajo sesgo combinada con una baja variabilidad. Además, los árboles de decisión son muy adecuados para clasificación cuando el número de clases es elevado (basta asignar una clase a cada hoja), y en nuestro caso tenemos 46 clases.

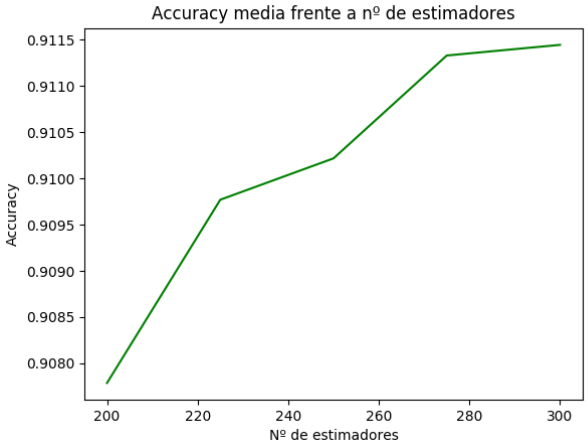
7.1. Estimación de hiperparámetros

De los múltiples hiperparámetros que podríamos ajustar (profundidad máxima de cada árbol, máximo número de nodos terminales, mínimo número de muestras en cada nodo, ...), sólo buscaremos un valor adecuado para el número de estimadores a tener en cuenta, manteniendo el resto por defecto.

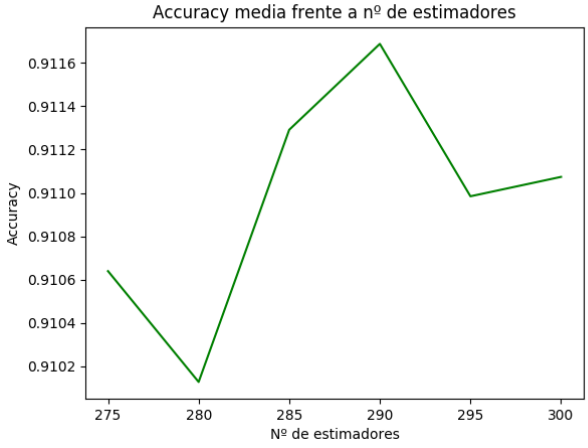
Para elegir un valor de `n_estimators`, representamos la accuracy media obtenida usando validación cruzada con tres subdivisiones (hacer la media de tres ejecuciones le da cierta estabilidad a los resultados) según diferentes valores de éste parámetro entre 50 y 300. Observamos que la accuracy es creciente con el múmero de estimadores, pero nos preguntamos si merece la pena ese crecimiento a costa del incremento del tiempo de ejecución que conlleva el uso de un mayor número de estimadores. Por eso, tomamos nuevas mediciones entre 200 y 300, donde vimos que este crecimiento parece saturar a partir de 275 árboles. Así, decidimos que el valor óptimo estaría dentro de este intervalo y repetimos ahí el experimento, obteniendo la máxima accuracy con 290 estimadores. Puesto que se hizo de 5 en 5, para obtener un valor más exacto, tomamos mediciones cada 2 estimadores entre 285 y 295. En la cuarta gráfica, observamos dos picos, en 287 y 293. Viendo la escala del eje, concluimos que la accuracy satura en este tramo y no compensa seguir aumentando el número de estimadores, por lo que elegimos 287 árboles (el primer pico) como valor para el hiperparámetro.



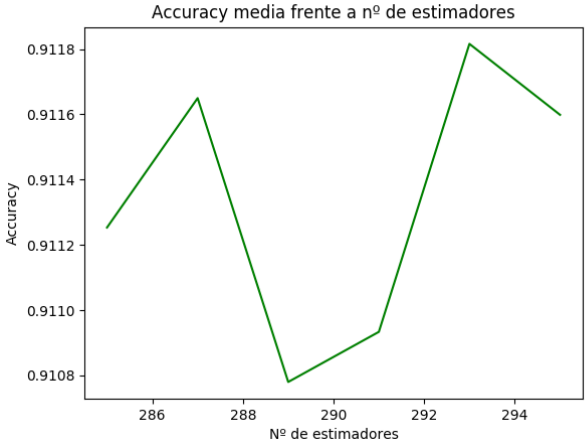
(a) Entre 50 y 300, de 50 en 50



(b) Entre 200 y 300, de 25 en 25



(c) Entre 275 y 300, de 5 en 5

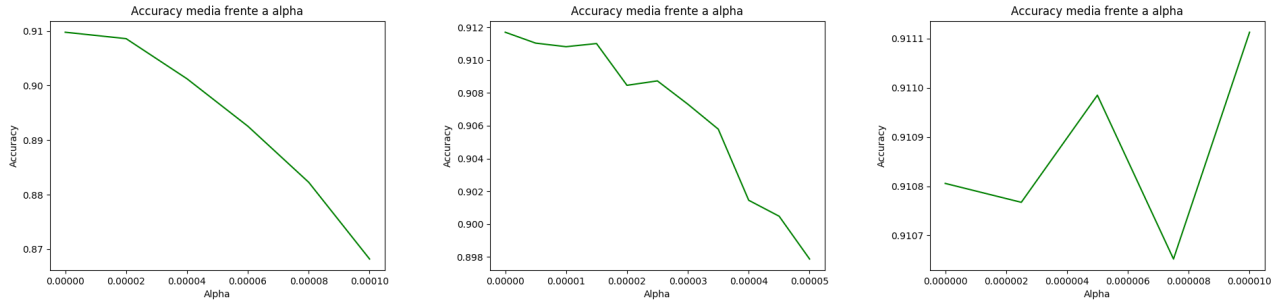


(d) Entre 285 y 295, de 2 en 2

Figura 7: Accuracy media de tres validaciones para distintos valores del hiperparámetro `n_estimators`

A continuación medimos la accuracy sobre el conjunto de train, obteniendo 1. Esto nos hace pensar que el modelo sobreajusta e intentamos regularizar para reducir la complejidad (el número de nodos) de cada árbol penalizando con el parámetro α.

Originalmente el valor por defecto de α es 0, hemos probado a incrementarlo y medir la accuracy media de tres subdivisiones de validación cruzada para cada uno de los diferentes valores de α probados. Claramente, la accuracy decrece al aumentar alpha. Observando la escala del eje en la última gráfica, concluimos que no mejoramos nada aumentando éste, por lo que mantenemos el valor inicial de α, 0.



(a) Entre 0 y 0.0001, 6 mediciones (b) Entre 0 y 0.00005, 11 mediciones (c) Entre 0 y 0.00001, 6 mediciones

Figura 8: Accuracy media de tres validaciones para distintos valores del hiperparámetro α

7.2. Función de pérdida y regularización

La función de pérdida que intenta minimizar cada uno de los árboles de decisión que promedia random forest es:

$$R(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|$$

donde $|T|$ es el número de nodos terminales del árbol T, N_m es el número de instancias que caen en el nodo terminal m, α el coeficiente que penaliza la complejidad del árbol (para regularizar) y $Q_m(T)$ la medida de impureza del nodo terminal m. Como medida de impureza, tomamos Gini Index, aunque la entropía cruzada proporciona resultados similares.

La medida de impureza Gini Index es:

$$Q_m(T) = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

donde $\hat{p}_{mk}$ es la proporción de la clase k en el nodo m.

Como hemos visto en el apartado anterior, aumentar el valor de α no se consiguen mejores resultados, por lo que en este caso, a pesar de que el modelo parece sobreajustado, regularizar de esta forma no es la mejor opción. Es por esto que tomamos $\alpha = 0$, quedando la función de pérdida:

$$R(T) = \sum_{m=1}^{|T|} N_m Q_m(T)$$

8. Multilayer Perceptron (MLP)

El modelo de perceptrón multicapa usado consta de tres capas: la capa de entrada, dos ocultas y una de salida. Como función de activación en cada neurona hemos considerado tanh y como algoritmo para ajustar los pesos usamos Adam, como nos recomendaron en teoría y como recomienda la documentación de *sklearn* para datasets grandes como es el nuestro.

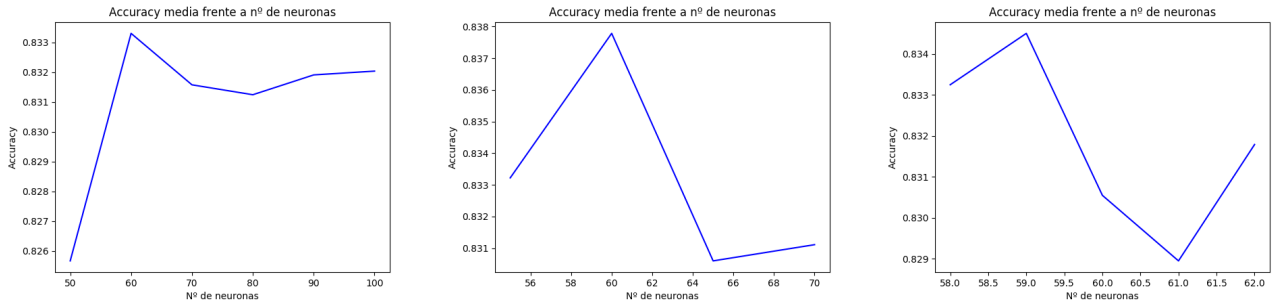
Al igual que random forest, el perceptrón multicapa tiene suficiente complejidad para obtener un bajo sesgo y facilidad para clasificación no binaria. Para afrontar el problema del sobreajuste, podemos utilizar early stopping

como regularización, ya que disponemos de un número elevado de muestras de entrenamiento y podemos permitirnos sacrificar algunas con el fin de combatir el sobreajuste. Por tanto, consideramos este modelo adecuado para el problema.

Durante el entrenamiento del modelo recibimos warnings acerca de la convergencia, por lo que tuvimos que incrementar el número máximo de iteraciones (`max_iter`) a 800.

8.1. Estimación de hiperparámetros

Estimamos el número de neuronas, `N_NEUR`, que tiene cada una de las capas ocultas. Como nos recomendaban valores entre 50 y 100 hicimos las mediciones de la accuracy media usando validación cruzada con dos subdivisiones del modelo con `N_NEUR` en dicho rango. Observamos que el máximo se alcanza en torno a 60 neuronas por capa, luego repetimos las mediciones ahora en el intervalo $[55, 70]$. A pesar de que la diferencia entre la accuracy es leve, seguimos buscando un valor para `N_NEUR` en torno a 60, que es donde conseguimos la accuracy más alta. Al medir la accuracy en el intervalo $[58, 62]$, obtenemos el máximo en 59. Por tanto, cada capa oculta del MLP usado para el ajuste tendrá 59 neuronas.



(a) Entre 50 y 100, de 10 en 10 (b) Entre 55 y 70, de 5 en 5 (c) Entre 58 y 62, de 1 en 1

Figura 9: Accuracy media de dos validaciones para distintos valores del hiperparámetro `N_NEUR`

8.2. Función de pérdida y regularización

En la documentación nos indican que la función de pérdida usada en la implementación de `MLPClassifier` es la pérdida logarítmica, que para clasificación multietiqueta, con un conjunto de K etiquetas, es:

$$L_{\log}(Y, P) = -\log Pr(Y|P) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_{ik} \log p_{ik}$$

donde Y es uma matriz codificada en binario con las verdaderas etiquetas de la muestra, es decir, $y_{ik} = 1$ si el dato x_i tiene etiqueta k (y 0 en otro caso) y P es una matriz de estimaciones probabilísticas que depende de los pesos.

Para regularizar usamos Early Stopping (`early_stopping = True`). Esto separa un 10 % de los datos de ajuste para validación y termina el entrenamiento cuando la accuracy de este conjunto no mejora en al menos 10^{-4} por 10 iteraciones consecutivas.
La accuracy obtenida con y sin regularización es:

	Sin early stopping	**Con early stopping**
Train Accuracy	0.9940217391304348	0.941128516624041
Validation Accuracy	0.8642583120204603	0.8879156010230179

Como podemos observar, regularizando el modelo conseguimos un mejor desempeño en el conjunto de valización, claramente reduciendo el sobreajuste.

9. Comparación de los modelos

9.1. Validación

Una vez ajustados los hiperparámetros de los modelos, procedemos a validarlos. Para ello entrenamos con el 80 % de los datos de train y validamos con el 20 %, como hemos comentado anteriormente. Insistimos en que disponemos de suficientes datos de validación para que los resultados sean significativos, por lo que evitamos realizar validación cruzada, que es computacionalmente muy costosa. En la siguiente tabla mostramos la accuracy obtenida con los diferentes modelos en el conjunto de entrenamiento y de validación:

	Train	**Validación**
Random Forest	1	0.9189258312020461
Multilayer Perceptron	0.9329923273657289	0.8831202046035805
Regresión Logística	0.7436700767263427	0.7192455242966752

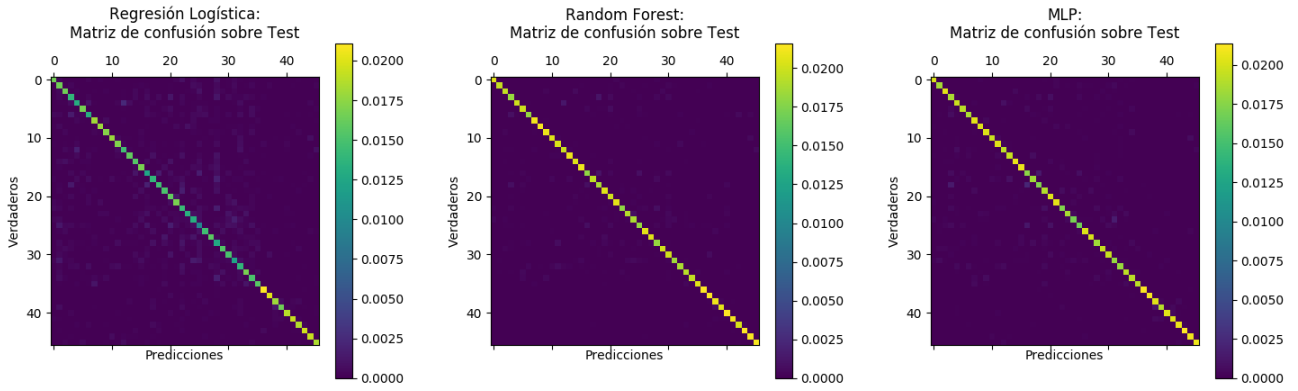
A la vista de los resultados obtenidos, el modelo con mejor desempeño es el Random Forest, seguido por el MLP y el modelo de Regresión Logística se queda bastante atrás.

9.2. Desempeño sobre test

Vamos a ver ahora el desempeño de estos modelos sobre el conjunto de test. Esta información no es relevante para tomar decisiones, ya que en un problema real carecemos de un subconjunto de test para realizar estas pruebas, por lo que todas las decisiones deben tomarse sobre la información obtenida a partir de los datos del subconjunto de entrenamiento y de validación.
Para sacar estos resultados, entrenamos con todo el conjunto de datos de entrenamiento iniciales (78200 instancias, 1700 por clase) uniendo nuestro anterior conjunto de train con el de validación. Calculamos la accuracy sobre las 13800 instancias (300 por clase) del conjunto de test. Al entrenar con más datos, es esperable que los resultados obtenidos sean mejores.
La accuracy obtenida por cada modelo es:

	Train	**Test**
Random Forest	1	0.923695652173913
Multilayer Perceptron	0.9485549872122763	0.8973913043478261
Regresión Logística	0.74423273657289	0.735072463768116

Las matrices de confusión de los modelos obtenidas sobre el conjunto de test son:



(a) Matriz de confusión Regresión Logística (b) Matriz de confusión Random Forest (c) Matriz de confusión Multilayer Perceptron

Figura 10: Matrices de confusión de los modelos

Las matrices de confusión de los modelos no lineales son casi diagonales, mientras que la del modelo de regresión logística presenta algunas impurezas.

Los resultados de los modelos sobre el conjunto de test confirman nuestras conclusiones sobre su bondad que deducimos de los resultados en validación.

10. Conclusiones y justificación

10.1. Valoración de los resultados

Ante los resultados obtenidos por validación, seleccionamos Random Forest como el mejor modelo y estimamos una accuracy de generalización de 0.9189258312020461 (obtenido por validación).

Tras comparar con los scores obtenidos sobre el conjunto de Test, observamos que efectivamente Random Forest generaliza mejor. La accuracy sobre test obtenida es ligeramente mayor a la estimada, como en teoría debe ocurrir, ya que el modelo con el que hemos validado estaba entrenado con menos datos y presenta una estimación pesimista de la accuracy fuera de la muestra.

De igual modo ocurre con los otros modelos, la validación es una estimación pesimista de su capacidad de generalización. Al menos basándonos en los datos de test.

10.2. Comparación con modelos diseñados por terceros

En el artículo Deep learning based large scale handwritten Devanagari character recognition consiguen una accuracy sobre el conjunto de test del 98.47 % usando Deep Convolutional Neural Network con Dropout para mejorar dicha accuracy. Según la página Devanagari Handwritten Character Dataset Data Set, es la accuracy más alta obtenida sobre el conjunto de test para este dataset. También hemos encontrado otros trabajos sobre este dataset como https://github.com/PrathamNawal/Devanagari-Character-Recognition, que utilizan técnicas similares y logran aproximadamente el mismo resultado.

Esto nos lleva a pensar que las Redes Neuronales Convolucionales están muy por encima para este tipo de problemas, como ya se nos comentó en clase de teoría. Luego los modelos considerados no son los más adecuados para este problema.

Aun así, dentro de los modelos que hemos estudiado, consideramos que nuestros resultados son más que aceptables. Hemos encontrado un trabajo (https://github.com/rishianand54/devanagari-character-recognition-system) en el que ajusta diversos modelos, que incluyen Random Forest y KNN, y al final logra una accuracy máxima sobre test del 92 %, precisamente en Random Forest.

Este otro trabajo (utiliza el mismo dataset, pero sólo considera las 36 clases correspondientes a los caracteres, desechando los dígitos): https://github.com/PriSawant7/ML-Devanagari-Character-Recognition compara diversos modelos, que incluyen KNN, Random Forest, MLP y CNN. Aunque las accuracy que obtiene son bastante más bajas, sospechamos que se debe a que no realiza preprocesamiento (ni siquiera elimina el marco).

Por tanto, aunque nuestros modelos sean superados por las CNN, consideramos que hemos logrado sacarles un buen desempeño mediante la elección de hiperparámetros y el preprocesamiento.