Proyecto:
Reconocimiento de dígitos manuscritos.

Mario Muñoz Mesa
Pedro Ramos Suárez

19 de octubre de 2022

Índice

1. Descripción y Planteamiento.

El Dataset Optical Recognition of Handwritten Digits contiene, con objetivo de reducir dimensionalidad, una transformación de preprocesado en lo que originariamente eran bitmaps 32×32 de dígitos manuscritos de 0 a 9. En la transformación se divide el bitmap en bloques 4×4 (tendremos 64 bloques) y se asigna a cada uno el número de 1s en el bloque, a cada bloque le corresponderá un entero n con $0 \leq n \leq 16$. Ya se nos proporcionan conjuntos de training y test, donde en training se tienen los dígitos dibujados por 30 personas y en test los dibujados por otras 13 personas distintas. El conjunto de training tiene 3823 instancias y el de test 1797 instancias.

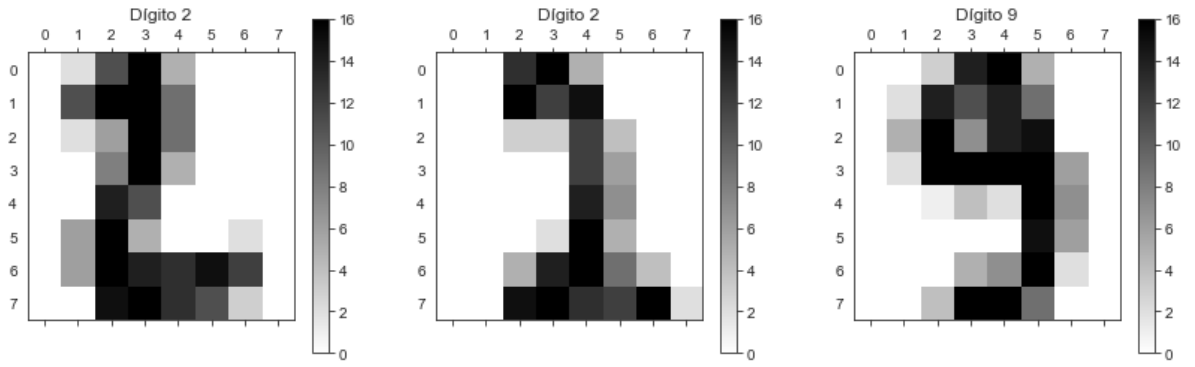


Figura 1: Representación visual de tres de nuestros vectores de características como matriz 8×8, etiquetados como 2, 2 y 9 respectivamente

Suponemos $(\Omega, \mathcal{A}, P)$ espacio probabilístico, donde Ω es el conjunto de posibles dígitos manuscritos 0 a 9 en bitmap 32×32 trazados por una persona cualquiera; $\mathcal{A}$ sigma-álgebra formada por todos los subconjuntos de Ω, y P distribución de probabilidad desconocida.

Sobre $(\Omega, \mathcal{A}, P)$ tenemos el vector aleatorio $\mathbf{x} = (x_1, \dots x_{64})$ donde cada variable aleatoria $x_i \colon \Omega \to \{0, \dots, 16\}$, $i \in \{1, \dots, 64\}$, mide la cantidad de 1s en el i-ésimo bloque 4×4 , y la variable aleatoria $y \colon \Omega \to \{0, \dots, 9\}$ que asigna el dígito manuscrito. Por lo que $\mathcal{X} = \mathbf{x}(\Omega) = \{0, \dots, 16\} \times \overset{64}{\cdots} \times \{0, \dots, 16\} = \{0, \dots, 16\}^{64}$ y $\mathcal{Y} = y(\Omega) = \{0, \dots, 9\}$
Buscamos encontrar aproximación de función $f \colon \mathcal{X} \to \mathcal{Y}$ que asigna el dígito correspondiente a cada matriz 8×8 transformada del bitmap original.

Suponemos $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)$ con $N = 3823$ muestra aleatoria i.i.d (independiente idénticamente distribuida). Lo que se nos ha proporcionado es una realización muestral o dataset $\mathcal{D} = (\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)$ (abuso de notación, ya no son vectores aleatorios).

Bajo estas condiciones podemos hacer uso de la Teoría de Vapnik-Chervonenkis.

2. Visualización.

Podemos visualizar nuestro conjunto de training mediante TSNE (t-Distributed Stochastic Neighbor Embedding)

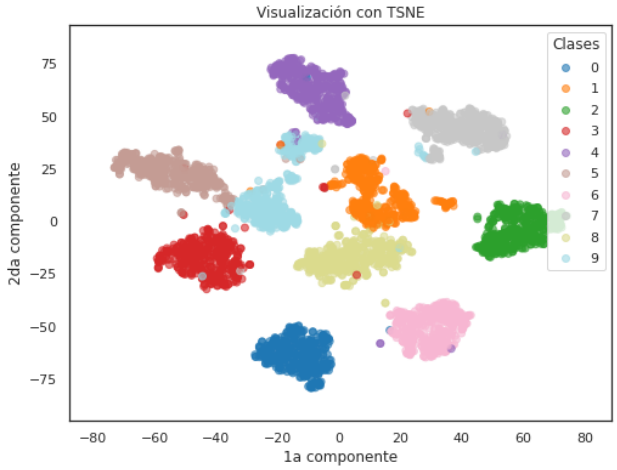


Figura 2: Visualización de los datos de training mediante TSNE.

donde observamos agrupamientos por clases, lo cual nos sugiere que obtener un buen clasificador en la muestra es posible.

3. Hipótesis finales que se usarán.

En el preprocesado:

(1) Se eliminan características varianza menor a 0.05 (también probaremos hipótesis eliminando solo las de varianza 0).

(2) En el caso del modelo lineal se realiza transformación polinomial de segundo orden.

(3) Se normalizan las características (media 0 y varianza 1).

La métrica a valorar es accuracy. Evaluaremos desempeño de: Regresión Logística multiclase (one vs all), Perceptron Multicapa, Random Forest y Máquina de Soporte de Vectores. Evaluaremos mediante 10-fold cross validation.

4. Generación de conjuntos training y test. Validación.

Hay un compromiso en la selección del tamaño de entrenamiento y test: el tamaño de la muestra de entrenamiento determina el número de instancias que tendremos para entrenar el modelo y por tanto para minimizar E_{in}, y el tamaño del conjunto de test condicionará la estimación, E_{test}, de E_{out} que obtengamos.

En nuestro caso se nos proporciona ya una división en training y test, un 68 % de los datos para training y un 32 % de los datos para test. Cabe mencionar que los datos de test fueron generados por personas distintas a las de training; es por esto que mantendremos la división, esperamos así que E_{test} sea lo más representativo posible de E_{out}. En caso contrario (no mantener la división y mezclar para hacer división propia) cabe pensar que el test tiene cierta relación con training, en el sentido de que hay datos generados por misma persona, y las estimaciones en test pudieran ser optimistas.

Para validar sería interesante tener en validación las instancias de vectores de características generadas por autores diferentes a los que generaron el conjunto que estamos entrenando, sin embargo no encontramos información suficiente en la descripción del dataset para proceder así. Utilizaremos k-fold cross validation con $k = 10$ para la selección de modelos. Esta técnica es derivada de *Leave-one-out*, la cual nos proporciona una estrategia para abordar la problemática

o compromiso de elección del tamaño de conjunto de validación, K, esta es: $E_{out}(g) \approx E_{out}(g^-)$ cuando K pequeño y $E_{out}(g^-) \approx E_{val}(g^-)$ cuando K grande.

La técnica k-fold cross validation consiste en dividir la muestra de entrenamiento en k particiones, cada una de tamaño aproximado $\frac{N}{k}$, e ir iterando sobre ellas eligiendo cada vez una partición para actuar como conjunto de validación y entrenando sobre el resto de la muestra el modelo; una vez iteradas sobre todas las particiones se toma la media de errores obtenidos y ese es el error de validación cruzada E_{cv} que es un buen estimador de E_{out} cuando k es grande, cuanto menor sea k peor será la estimación de E_{out}. Se elige el modelo con menor E_{cv}

Se podría tomar $k = N$ (Leave One Out) pero esto implica un alto coste computacional; es por esto que se elige $k \ll N$, en nuestro caso $k = 10$ (es habitual $5 \leq k \leq 10$)

5. Detalles del preprocesado de datos.

Pasos realizados:

1. Mostramos visualemente la matriz de coeficientes de correlación lineal habiendo eliminado ya 1 característica que tenía varianza 0 (necesario para poder calcularla)

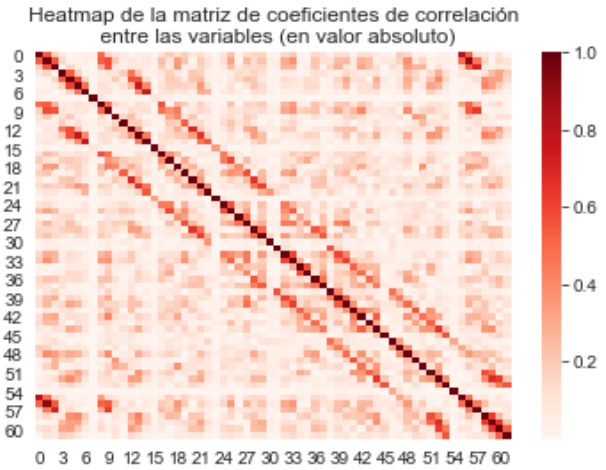


Figura 3: Matriz coeficientes de correlación lineal.

No observamos problema de altas correlaciones.

2. Podemos observar las varianzas de las características (todas tienen la misma medida) en training

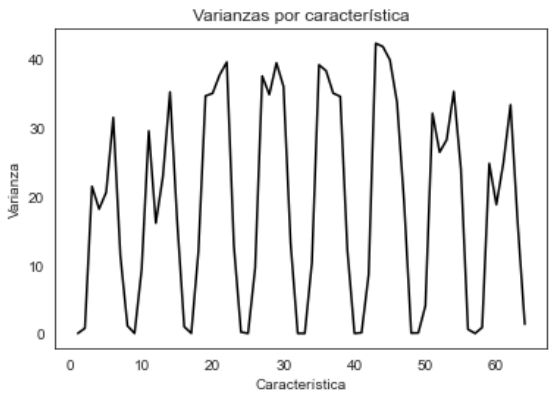


Figura 4: Varianzas por característica.

Vemos que hay características con una varianza muy baja. Es por esto que nos planteamos que varias de estas características tienen muy bajo poder predictivo. Nuestro caso es fácilmente interpretable y ya podemos intuir que probablemente esquinas o bordes de las imágenes no varíen apenas su valor.

Eliminaremos las características con varianza menor a 0.05, elegimos un valor prudente. Se eliminan por tanto las características 0, 8, 16, 24, 31, 32, 39, 47 y 56, todas con percentil 99: 0, excepto la 47 que tiene percentil 98: 0, (si nos fijamos todas expresables como $8n$ o $8n-1$ con $n \in \mathbb{N}$, que en la matriz 8×8 corresponden a los bordes laterales)

3. En el caso de modelo lineal: Realizamos transformación polinomial de segundo orden a cada característica para aumentar así la flexibilidad de la frontera de decisión de nuestro modelo.

4. Normalizamos las características (tendrán media 0 y varianza 1), no normalizar puede degradar el desempeño de Gradiente Descendente Estocástico (sensible a la escala de las características), normalización motivada tras la transformación polinómica. En los casos que no realizamos la transformación polinomial no consideramos que perdamos nada por realizarla aunque nuestras características ya tengan la misma escala.

6. Métrica de error a usar.

Visualizaremos el número de vectores de características de la realización muestral pertenecientes a cada clase, esto nos mostrará si hay clases desbalanceadas y qué métrica necesitamos utilizar.

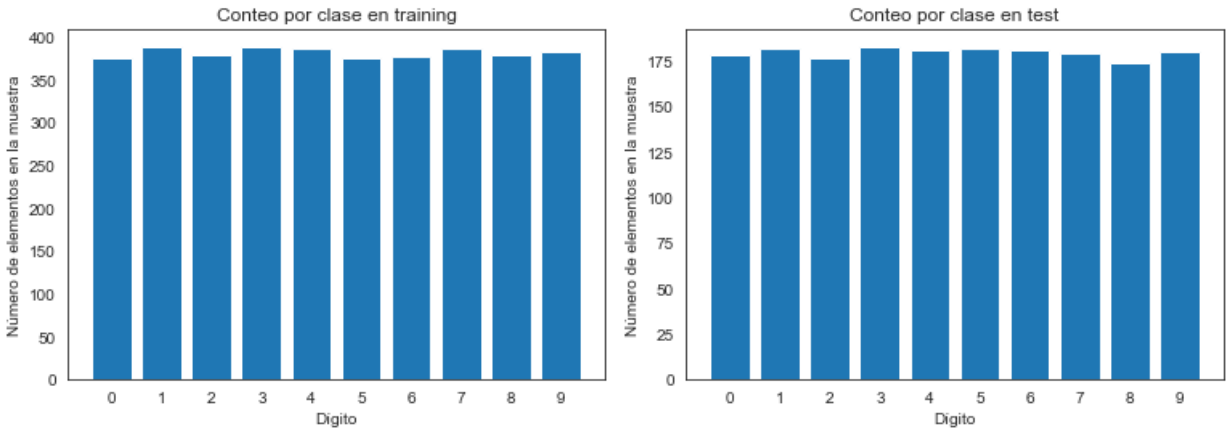


Figura 5: Número de elementos por etiqueta de dígito.

Como podemos ver las clases están balanceadas. Como no tenemos clases desbalanceadas utilizaremos la métrica *accuracy* que no es más que la proporción de predicciones de clase correctas.

7. Modelo lineal.

En el caso del modelo lineal realizaremos una transformación de segundo orden polinomial a los vectores de características, pues aumenta la flexibilidad de nuestra frontera de decisión. No utilizamos transformación polinómica de mayor orden pues cuanto mayor sea la longitud de los vectores de características mayores posibilidades de disminuir el error en la muestra pero mayores posiblidades de aumentar error en la población (sobreajuste); a parte de incrementar el coste computacional.

Por lo que, como hemos comentado, aplicaremos a $\mathcal{X}$ la transformación Φ_2, que genera combinaciones polinómicas de grado menor o igual a 2 de las características

$$\Phi_2(\mathbf{x}) = (1, x_1, \dots, x_{\hat{d}}, \underbrace{x_1x_2, \dots, x_1x_{\hat{d}}, x_2x_3, \dots, x_2x_{\hat{d}}, \dots x_{\hat{d}-1}x_{\hat{d}}}_{\text{combinaciones } x_ix_j \text{ con } i<j, \quad i,j\in\{1,\dots,\hat{d}\}}, x_1^2, \dots x_{\hat{d}}^2)^T$$

Nota: aquí $\hat{d} < d = 64$ pues no utilizaremos todas las características.

Regresión Logística multiclase (One-vs-Rest):

Nuestra primera elección es utilizar Regresión Logística multiclase pues en situaciones reales tiene más sentido pensar situaciones probabilísticas que en funciones deterministas. Además la función de error será fácilmente minimizable mediante Gradiente Descendente Estocástico. Asignaremos cada vector de características a la clase que se estime más probable.

Si

- Tomamos como función objetivo $f\colon \mathcal{X} \to [0,1]^{10}$ con $f(\mathbf{x}) = (P(y=0|\mathbf{x}), \dots, P(y=9|\mathbf{x}))^T$ la función que asigna a cada vector de características el vector de probabilidades de pertenecer a cada clase
- Denotamos con d a $2\hat{d} + \frac{(\hat{d}-1)\hat{d}}{2}$ y con x a $\Phi_2(\mathbf{x})$
- Denotamos con w_i, $i = 1, \dots, K$, con $K = 10$, el hiperplano que separa la clase i del resto

tenemos la clase de funciones

$$\mathcal{H} = \Big\{ h_{w_1,\dots w_K}\colon \mathbb{R}^{d+1} \to [0,1]^K \ : \\ h_{w_1,\dots w_K}(x)) = Softmax(\ (w_1^Tx, \dots, w_K^Tx))^T\ ),\ w_1, \dots w_K \in \mathbb{R}^{d+1} \Big\}$$

esto es

$$\mathcal{H} := \Big\{ h_{w_1,\dots w_K}\colon \mathbb{R}^{d+1} \to \mathbb{R}^K \ : \\ h_{w_1,\dots w_K}(x) = \frac{1}{\sum_{k=1}^K e^{w_k^Tx}}(e^{w_1^Tx}, \dots, e^{w_K^Tx})^T,\ w_1, \dots w_K \in \mathbb{R}^{d+1} \Big\}$$

La función de pérdida (máxima verosimilitud) es

$$E(w_1, \dots, w_K) := -\ln\left(L(Y|w_1, \dots, w_K)\right) = -\sum_{n=1}^N \sum_{k=1}^K y_{nk} \ln\left(\sigma(w_k^Tx_n)\right)$$

donde σ es la función logística (sigmoide), $\sigma\colon \mathbb{R} \to [0,1]$, $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$, y $y_{nk} = 1$ si $y_n = k$ y $y_{nk} = 0$ si $y_n \neq k$

Vemos que

$$\nabla_{w_j} E(w_1, \dots, w_K) = \sum_{n=1}^N (\sigma(w_j^Tx_n) - y_{nj})x_n$$

Utilizaremos SGD (Apéndice 11.1.) para minimizar. Una vez acabada la optimización, obtendremos los $w_1, \dots, w_K$ que minimizan E_{in}. Asignaremos entonces cada x a la clase más probable, es decir a la clase $j' \in \{1, \dots, K\}$ donde

$$j' = \arg max_{j\in\{1,\dots,K\}} \frac{e^{w_j^Tx}}{\sum_{k=1}^K e^{w_k^Tx}}$$

7.1. Parámetros usados y tipo de regularización elegida.

La regularización nos ayudará a evitar sobreajuste, (en el caso lineal también motivada por la transformación polinómica), con la regularización se aumentará ligeramente el sesgo para decrementar significativamente la varianza.

Modelo lineal:
Utilizaremos regularización Ridge, ahora tendremos error aumentado

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

donde $\lambda \geq 0$ es el parámetro de regularización. Este tipo de regularización penaliza coeficientes de $\mathbf{w}$ grandes. Otro tipo de regularización conocida es regularización Lasso (utiliza norma $\|\cdot\|_1$ en lugar de norma $\|\cdot\|_2$), ésta sin embargo tiende a hacer coeficientes de $\mathbf{w}$ a 0, es buena para selección de características. En nuestro caso consideramos más conveniente Ridge pues confiamos en haber eliminado ya características no informativas en preprocesado.

En Regresión Logística multiclase hemos utilizado la función `SGDClassifier`, ésta por defecto sigue el criterio one vs rest que ajusta cada clase respecto a las demás, y que es el que utilizaremos por su eficiencia e interpretabilidad. `SGDClassifier` utiliza tamaño de minibatch 1. Es interesante notar que la función de error junto con el término de regularización Ridge es convexa; pierde relevancia el uso de minibatches o punto de inicio al no tener óptimos locales. Se podría utilizar Gradiente Descendente sin problema. Aún así, dado que `sklearn` nos proporciona método para la versión estocástica, y que realmente ganamos eficiencia computacional al calcular gradiente en un solo punto, y el "ruido" que supone computar el gradiente en un solo punto termina promediándose en número alto de iteraciones; se decide utilizar la versión que nos facilita `sklearn`.

El learning rate se ha elegido adaptativo, cuando no se esté mejorando en el criterio de tolerancia tras `n_iter_no_change` épocas, entonces actualizamos la tasa de aprendizaje mediante $learning\_rate = \frac{learning\_rate}{5}$, vamos disminuyendo conforme nos acercamos al mínimo para prevenir oscilación. Se ha elegido learning rate inicial algo elevado puesto que es adaptativo y tendremos una tolerancia `tol` a la que daremos hasta 5 épocas para que se halla disminuido en `tol` el error. Elegimos hasta `n_iter_no_change`=5 épocas de forma arbitraria, ahora bien, como no sabemos cómo de rápido descenderemos no sabemos qué valor de `tol` puede ser el adecuado para que tras 5 épocas sin mejora decremente el learning rate, por lo que lo consideraremos como hiperparámetro. El algoritmo parará si llegamos learning rate menor a 1e-6 o por iteraciones máximas. El número de iteraciones se ha elegido arbitrario pero que muestra convergencia, es decir, nuestro algoritmo para por learning rate menor a 1e-6 y no por iteraciones máximas.

Se adjunta código con comentarios del resto de parámetros:

```
{"model": [SGDClassifier(
           loss = 'log', # función de pérdida de regresión logística
           penalty = 'l2', # utilizaremos regularización l2 (reg. Ridge)
           # alpha  constante del término de regularización (probaremos distintos valores mediante 10-fold cross validation)
           fit_intercept = True, # añadimos sesgo o intercept pues nuestra matriz aún no tiene columna de 1s
           max_iter = 180, # Número máximo de iteraciones arbitrario
           #tol  Tolerancia para criterio de parada por tolerancia (parar si loss > best_loss - tol tras n_iter_no_change épocas seguidas)
           # el criterio valora si el error no mejora en tol el mejor error hasta el momento. En nuestro caso esta condición por tolerancia se usará para learning_rate adaptativo
           shuffle = True, # mezclamos después de cada época,
           n_jobs = -1, # máxima paralelización posible en ejecución
           random_state = SEED, # para tener reproducibilidad de los resultados
           learning_rate = 'adaptive', # si no se mejora resultado por criterio tolerancia (loss > best_loss - tol) tras n_iter_no_change épocas seguidas, entonces cambiamos learning rate por (learning rate)/5
                                       # queremos evitar oscilación
           eta0 = 0.05, # learning rate inicial arbitrario
```

```
15          early_stopping = False, # False pues no queremos reservar más datos
    para validación, consideramos nuestro dataset pequeño y que no conviene quitar
     más datos de training
16          n_iter_no_change = 5, # cada 5 épocas sin mejora en crit. tolerancia
    se realizará la adaptación del learning rate (learning_rate='adaptive')
17          class_weight = None, # se interpreta que todas las clases tienen peso
    1, se elige así pues nuestro problema tiene clases balanceadas
18          average = False, # no nos interesa obtener media de pesos
19          verbose = 0, # no nos interesan mensajes
20          warm_start = False, # no reutilizamos ninguna solución anterior
    durante la validación cruzada
21          l1_ratio = 0 # 0 corresponde a l2 penalty, no se usará pues solo se
    usa si learning_rate = 'elasticnet'
22          )], # máxima paralelización posible
23      "model__alpha": [0.000001, 0.0001, 0.001], # probamos valores de
    regularización (hiperparámetro)
24      "model__tol": [0.0001, 0.001, 0.01]} # probamos distintos valores de
    tolerancia (hiperparámetro)
25
26
```

Listing 1: Parámetros usados en SGDClassifier

8. Perceptron Multicapa.

Perceptron Multicapa es una generalización del modelo Perceptron que adquiere una gran flexibilidad pero también potencial sobreajuste, por lo que utilizaremos regularización. En este modelo, la clase de funciones a usar queda determinada por la red. En nuestro caso utilizaremos 3 capas, dos de ellas ocultas y la de salida. La clase de funciones tendrá la forma

$$\mathcal{H} = \{h_{\mathbf{w}} \colon \mathbb{R}^{d+1} \to [0,1]^{10} \ : \ Softmax\left(\mathbf{w}_3^T \theta(\mathbf{w}_2^T \theta(\mathbf{w}_1^T \mathbf{x}))\right)\}$$

donde θ es la función de activación. Ahora la función de pérdida es la función de pérdida de entropía cruzada

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

donde K es el número de etiquetas, Y es matriz binaria indicadora donde $y_{i,k} = 1$ si la instancia i tiene etiqueta k, y $\hat{P}$ es la matriz de probabilidades estimadas con $p_{i,k} = P(y_{i,k} = 1)$.

En Perceptron Multicapa la función a minimizar no es convexa, la minimización se hará mediante SGD y para el cálculo de gradiente se utilizará el algoritmo Backpropagation. Por la naturaleza de Backpropagation (vamos calculando gradientes desde capa final hasta inicial) tenemos el fenómeno conocido por *Vanishing Gradient Problem*, en nuestro caso no tenemos muchas capas aún así podemos elegir función de activación ReLU ($\theta(x) = \text{máx}\{0, x\}$) que ayudaría en esta problemática por ser non-saturating function a diferencia de tanh y σ que sí son saturating functions (ReLU a diferencia de tanh y σ no está acotada superiormente). Con ReLU podemos obtener buenos resultados al igual que con tanh y σ, y tendremos ganancia en tiempo de computación.

Elegiremos método de optimización Adam, tenemos los beneficios de tasa de aprendizaje adaptativa (tasas adaptativas individuales a cada parámetro) y momentum (ayuda a evitar situaciones oscilatorias). Desde scikit-learn vemos que es un método robusto, conocido por dar buenos resultados en datasets de miles de datos como el nuestro, es eficiente y da buenos resultados en validación. Además nos ahorramos tener que hacer tan buena "afinación" en los parámetros como en SGD para obtener buenos resultados.

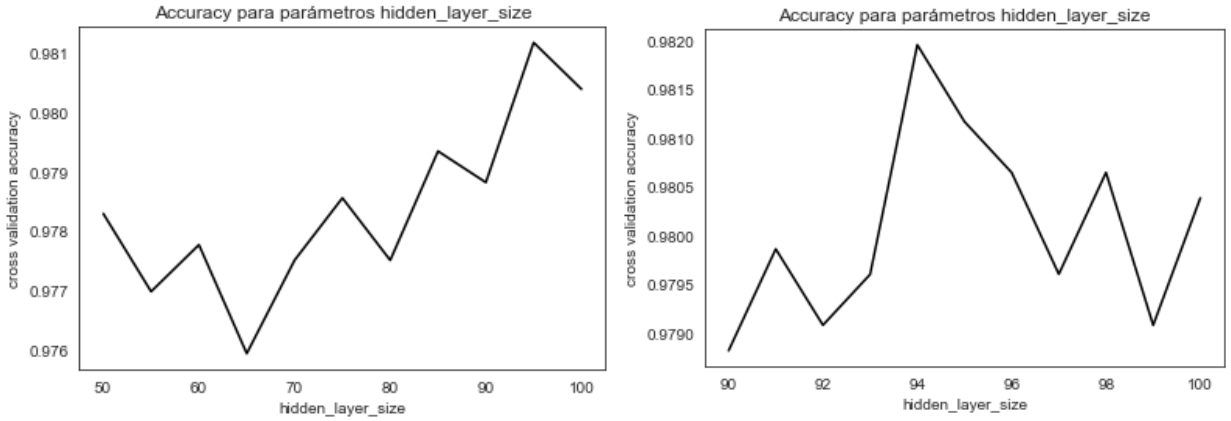
8.1. Parámetros usados y tipo de regularización elegida.

Utilizaremos regularización Ridge, ahora tendremos error aumentado

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

donde $\lambda \geq 0$ es el parámetro de regularización. Con este tipo de regularización penalizaremos coeficientes de $\mathbf{w}$ grandes. No utilizaremos Early Stopping pues consideramos que no tenemos conjunto de training de gran tamaño y reducirlo más podría degradar los resultados; además cualquier pequeña oscilación en error validación puede hacernos parar antes de tiempo.

Estimación de hipérparámetros:
Para estimar el número de neuronas en las capas ocultas hemos seguido la estrategia de hacer un "barrido" inicial con el mismo número de neuronas en las dos capas ocultas en el rango recomendado (entre 50 y 100) en pasos de 10 en 10, para después hacer un segundo barrido en pasos de 1 en 1 en la zona que muestre mejor accuracy medio en validación cruzada



(a) Barrido de 10 en 10 entre 50 y 100

(b) Barrido de 1 en 1 entre 90 y 100

Figura 6: Gráficas con accuracy medio por validación cruzada para distintos valores de número de neuronas en ambas capas

El mayor valor accuracy medio en validación cruzada lo encontramos con 94 neuronas para ambas capas ocultas, este será el valor elegido.

Como hemos comentado utilizaremos método Adam y función de activación ReLU. Una vez fijado el número de neuronas en las dos capas ocultas, tendremos los parámetros:

```
[{"model": [MLPClassifier(
      hidden_layer_sizes=(94,94), # hiperparámetro estimado por validación cruzada
      activation='relu', # elegimos función activación relu por ventajas ante tanh y sigmoide
      solver='adam', # elegimos método adam, robusto y eficiente, no requiere afinar tanto parámetros como sgd
      #alpha parámetro reg. l2 a estimar
      batch_size=128, # elegimos tamaño de batch arbitrario, pensamos que es un tamaño razonable para el tamaño de nuestro dataset aunque no podemos saber cómo se comportará si no es experimentalmente
      # es candidato a hiperparámetro a estimar por cv, nosotros fijamos uno por no hacer cv con mucha carga
      learning_rate='adaptive', # solo usado cuando solver='sgd', no es nuestro caso
      learning_rate_init=0.001, # elegimos learning rate inicial recomendado (pág. 2 https://arxiv.org/pdf/1412.6980.pdf) (buenos resultados por defecto)
      power_t=0.5, # solo usado cuando solver='sgd', no es nuestro caso
      max_iter=250,  # número máximo de épocas arbitrario pero que muestra convergencia
```

```
        shuffle=True,  # mezclamos después de cada época
        random_state=SEED, # para tener reproducibilidad de los resultados
        tol=0.001, # tolerancia de parada, si no mejoramos una milésima tras n_iter_no_change épocas entonces paramos, pensamos que es un valor razonable
        verbose=False, # no queremos mensajes
        warm_start=False, # no reutilizamos ninguna solución anterior durante la validación cruzada
        momentum=0.9, # solo usado cuando solver='sgd', no es nuestro caso
        nesterovs_momentum=True, # solo usado cuando solver='sgd', no es nuestro caso
        early_stopping=False, # pues consideramos que tenemos un bajo tamaño de entrenamiento
        validation_fraction=0.1, # no nos importa valor pues no usamos early_stopping
        beta_1=0.9, # elegimos tasa exponential decay para primer momento recomendada (pág. 2 https://arxiv.org/pdf/1412.6980.pdf) (buenos resultados por defecto)
        beta_2=0.999, # elegimos tasa exponential decay para segundo momento recomendada (pág. 2 https://arxiv.org/pdf/1412.6980.pdf) (buenos resultados por defecto)
        epsilon=1e-08, # valor numerical stability para adam recomendado (pág. 2 https://arxiv.org/pdf/1412.6980.pdf) (buenos resultados por defecto)
        n_iter_no_change=10, # damos hasta 10 épocas para mejorar error en 0.001, si no se mejora pararemos
        max_fun=15000)], # solo usado cuando solver='lbfgs', no es nuestro caso
        "model__alpha": [0.00001,0.0001,0.001]} # parámetro de reg. l2 para cross validation


```

Listing 2: Parámetros usados en MLPClassifier

9. Máquina de Soporte de Vectores.

Elegimos SVM porque contempla una posibilidad que no contempla ninguno de los otros modelos, que es buscar solución con "margen amplio", por lo que la dimensión d_{VC} de nuestra solución será menor y podemos esperar más capacidad de generalización.
SVM persigue buscar un hiperplano frontera de clasificación óptimo, óptimo en el sentido de maximizar su distancia a los puntos frontera de cada clase (vectores soporte) (buscamos "pasillo" lo más ancho posible), éste se resuelve como problema programación cuadrática con restricciones. Cuando los datos no son separables en espacio original pero al transformarlos sí, entonces programación cuadrática con restricciones ofrece solución pero una eficiencia inabordable. En este caso resolvemos el problema dual utilizando el "truco del kernel" (nos facilita el cálculo del producto escalar de transformación de vectores). El problema hasta ahora se conoce como Hard-Margin SVM.

Nosotros usaremos la versión "suave", Soft-Margin SVM, pues pretendemos evitar incurrir en sobreajuste por ruido al introducir un margen de violabilidad del pasillo para cada punto. Finalmente la función a minimizar es

$$\min_{\mathbf{c},\mathbf{w}} \lambda \mathbf{w}^T\mathbf{w} + \underbrace{\frac{1}{N}\sum_{n=1}^{N} \max\{1 - y_n(\mathbf{w}^T\mathbf{x}_n + b), 0\}}_{E_{SVM}(b,\mathbf{w})} \quad \lambda = 1/2CN$$

Elegimos kernel RBF pues en principio no tiene por qué darnos peor resultado que el kernel polinomial, y nos facilita el ajuste al tener solo un parámetro λ para "afinar".

9.1. Estimación de hiperparámetros.

Tendremos dos hipérparámetros a estimar γ y C. Para valores C grandes tendremos baja violabilidad del pasillo (pasillo estrecho), y con C pequeño más libertad de violabilidad. Por otra parte tenemos el parámetro γ asociado a nuestro kernel que determinará cómo de lejos puede cada punto influir en la determinación de la frontera de decisión; actúa como la inversa del radio de influencia (valores bajos: lejos, valores altos: cerca).

Desde la documentación de scikit-learn se nos indica que un grid de búsqueda en escala logarítmica entre 10^{-3} y 10^3 suele ser suficiente para estimar C y γ. La estrategia que hemos seguido ha sido tomar 7 parámetros para C entre 10^{-3} y 10^3 siguiendo escala logarítmica, y para cada uno hacer una búsqueda dicotómica de γ (binaria, desplazándonos hacia los valores que dan más accuracy). El parámetro C ganador fue 10

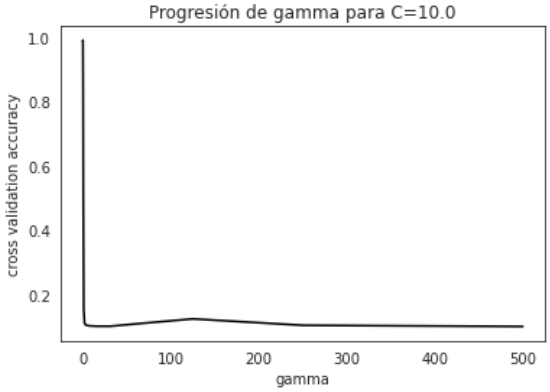


Figura 7: Progresión de γ para $C = 10$ (de derecha a izquierda)

y el ganador γ junto a ese C fue $\gamma = 0.02007346725463867$
Adjuntamos código con el resto de parámetros

```
{"model": [SVC(C=w_p_c, # hiperparámetro estimado c, w_p_c=10
      kernel='rbf', # kernel Radial Basis Function
      degree=3, # solo para kernel poly, no es nuestro caso
      gamma=w_p_gam, # hiperparámetro estimado gamma, w_p_gam
  =0.02007346725463867
      coef0=0.0, # solo para kernel poly, no es nuestro caso
      shrinking=True, # suponemos número alto de iteraciones, nos podrá ayudar a
   reducir tiempo de cómputo
      probability=False, # consideramos que no es necesario y nos aumentaría
  coste computacional
      tol=0.001, # tolerancia para criterio parada, hasta milésima
      cache_size=200, # puede mejorar tiempo ejecución para problemas de muchos
  datos, en nuestro caso consideramos que son pocos datos y que con 200MB sería
  suficiente
      class_weight=None, # suponemos todas las clases con peso 1 pues tenemos
  clases balanceadas
      verbose=False, # no queremos mensajes
      max_iter=-1, # no establecemos criterio de parada por iteraciones
      decision_function_shape='ovr', # devolvemos función decisión one vs rest
      break_ties=False, # no consideramos casos de empates, y ahorraremos
  considerablemente en coste computacional
      random_state=SEED)], # para tener reproducibilidad de resultados
      }


```

Listing 3: Parámetros usados en SVC

10. Random Forest.

El modelo Random Forest consiste en crear una serie de árboles, utilizando *bootstrap* y fijando el número de características de cada árbol a la raíz del número total de características, seleccionadas de manera aleatoria.

La función de pérdida que intentamos minimizar es:

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

donde $|T|$ es el número de nodos terminales, α es el parámetro de complejidad utilizado para la poda de costo mínimo-complejidad, N_m es el número de muestras que hay en el nodo m, y $Q_m(T)$ es la medida de la impureza de los nodo m.

Como medida de la impureza Q_m utilizaremos el criterio de impurezas Gini Index, que viene dada por:

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

donde $\hat{p}_{mk}$ es la proporción de la clase k en el nodo m, y viene dada por:

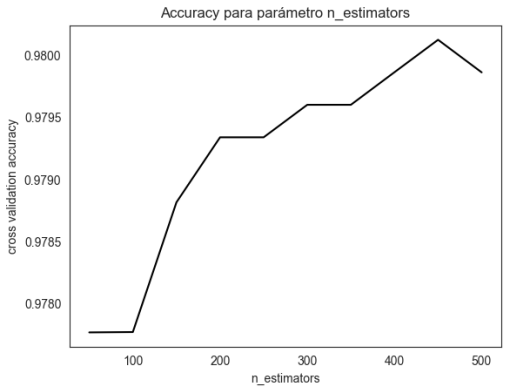
$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} [y_i = k]$$

donde R_m es la región que representa el nodo m.
Utilizamos el criterio Gini Index en lugar de entropy porque los resultados son similares y nos ofrece mejor eficiencia al no tener que computar función logarítmica.

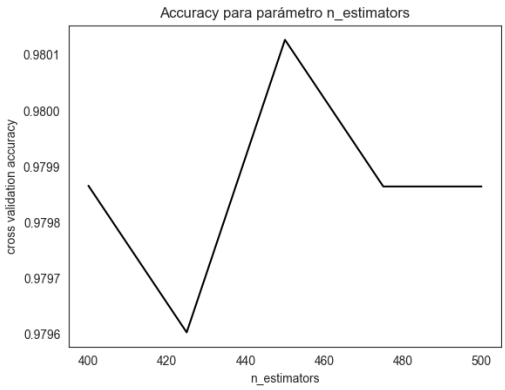
El número de árboles será hiperparámetro y para estimarlo utilizaremos $\alpha = 0$

10.1. Estimación de hipérparámetros:

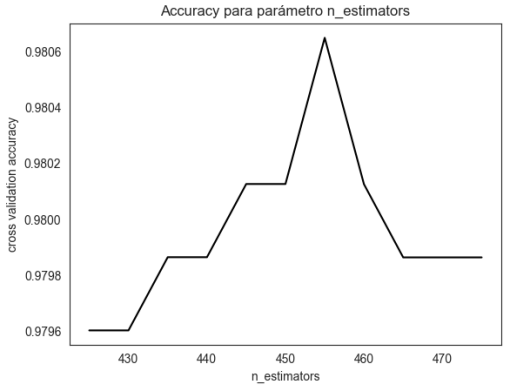
Para estimar el número de árboles de decisión utilizados, utilizamos de nuevo la estrategia de hacer un "barrido" inicial con el número de árboles entre 50 y 500 en pasos de 50 en 50, para luego hacer más barridos reduciendo el tamaño del intervalo y de los pasos hasta encontrar un buen valor.



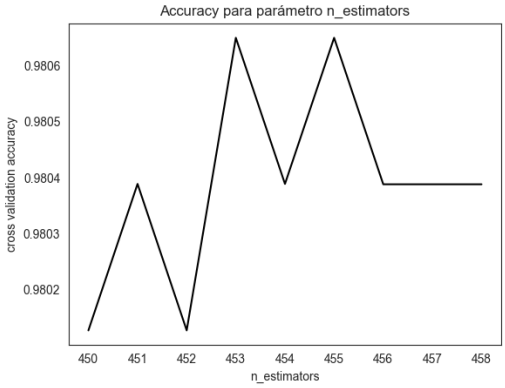
(a) Barrido de 50 en 50 entre 50 y 500



(b) Barrido de 25 en 25 entre 400 y 500



(c) Barrido de 5 en 5 entre 425 y 475



(d) Barrido de 1 en 1 entre 450 y 458

Figura 8: Gráficas con accuracy medio por validación cruzada para distintas cantidades de árboles de decisión.

El mayor valor de accuracy lo obtenemos para 453 y 455 árboles de decisión, por lo que usaremos 453 árboles para que el tiempo de ejecución sea ligeramente inferior al tener menos árboles.

Utilizando el criterio Gini Index como hemos comentado previamente, y 453 árboles de decisión, los parámetros finales para el modelo son:

```
{"model": [RandomForestClassifier(
      n_estimators=453,    # usamos el número de estimadores calculados
      criterion='gini',    # usamos gini porque es mas rapido y nos da resultados
   similares
      max_depth=None,      # para que explore la profundidad maxima del arbol
      min_samples_split=2,  # número mínimo de muestras para dividir un nodo,
  con 2 divide siempre que no sean hojas
      min_samples_leaf=1,      # una hoja se considera cuando solo es una muestra
      min_weight_fraction_leaf=0.0,   # para que todas las hojas tenga el mismo
  peso
      max_features='auto',     # utiliza la raiz cuadrada del número de caracterí
  sticas
      max_leaf_nodes=None,     # para que utilice todos los arboles
      min_impurity_decrease=0.0,  # para que expanda todos los nodos
      min_impurity_split=None,     # para que expanda todos los nodos
      bootstrap=True,      # como hemos visto en teoría, mejoramos los resultados
   usando bootstrap
      oob_score=True,      # solo queremos que use los datos de la muestra
      n_jobs=-1,      # para que utilice todos los cores del procesador
      random_state=SEED,  # para tener reproducibilidad de los resultados
      verbose=0,  # no queremos mensajes
      warm_start=False,    # para que cree el arbol de cero
```

```
        class_weight=None,  # para que todas las clases tengan el mismo peso
        #ccp_alpha grado de penalización por complejidad. Cuanto más alto más
    podado. Parámetro para regularizar y evitar sobreajuste. Será hiperparámetro a
     estimar
        max_samples=None)], # para que utilize todos los datos
        "model__ccp_alpha": [0.0, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5]} #hiperparámetro a
     estimar

```

Listing 4: Parámetros usados en RandomForestClassifier

11. Selección del mejor modelo.

Para la selección del mejor modelo utilizamos 10-fold cross validation como ya comentamos. Elegiremos como modelo ganador aquel con menor E_{cv}. El modelo ganador se entrenará finalmente sobre toda la muestra. Para hacer esto utilizamos `GridSearchCV`, función que nos permite entrenar el modelo ganador en toda la muestra mediante `refit=True`, y nos facilita variables con los resultados obtenidos. La métrica considerada para la elección del mejor modelo es el accuracy (tenemos clases balanceadas).

Las funciones de los modelos con parámetros ganadores son

- `SGDClassifier(eta0=0.05, l1_ratio=0, learning_rate=ádaptive', loss='log', max_iter=180, tol=0.0001, alpha=0.0001)`
- `MLPClassifier(alpha=1e-05, batch_size=128, hidden_layer_sizes=(94,94), learning_rate=ádaptive', max_iter=250, tol=0.001)`
- `RandomForestClassifier(ccp_alpha=0.0, n_estimators=453)`
- `SVC(C=10, gamma=0.02007346725463867)`

Nos referiremos a estos modelos, con los parámetros ya comentados, como $M_{RLClass}$, $M_{MLPClass}$ y $M_{RandForestClass}$ y $M_{SVMClass}$ respectivamente. Veamos los accuracies medios obtenidos con 10-fold cross validation

Modelo	Accuracy medio 10-fold cross validation
$M_{SVMClass}$	0.98979878
$M_{SGDClass}$	0.98325906
$M_{MLPClass}$	0.98195563
$M_{RandForestClass}$	0.98065014

Cuadro 1: Accuracies medios por 10-fold cross validation de cada modelo.

Elegimos el modelo con menor E_{cv}, en este caso $M_{SVMClass}$ (mayor accuracy). Una vez nuestro modelo ganador es entrenado sobre toda la muestra de entrenamiento (en 10-fold cross validation no utilizábamos N instancias, siempre reservábamos una de las 10 particiones para validar), utilizaremos nuestro conjunto de test que guardamos desde el principio para estimar el error de generalización, estimaremos E_{out} mediante E_{test}. Procediendo así, y teniendo en cuenta que estamos valorando mediante la métrica accuracy, estimamos error, de muestro mejor modelo $M_{SVMClass}$, fuera de la muestra: $1 - 0.97495826 = 0.02504174$. Podemos observar que es un error algo más elevado que el obtenido por validación cruzada $1 - 0.98979878 = 0.01020122$, aquí es donde juega su papel como medida de calidad el haber mantenido en test las imágenes de autores diferentes a los implicados en training (podríamos decir que al no ser los mismos autores no hemos "modelado

la forma de escritura propia de los autores", hay un "comportamiento" en test no modelado; la idea es que sea así por los motivos ya explicados en el punto 4).

Mostramos:

- La matriz de confusión del modelo ganador:

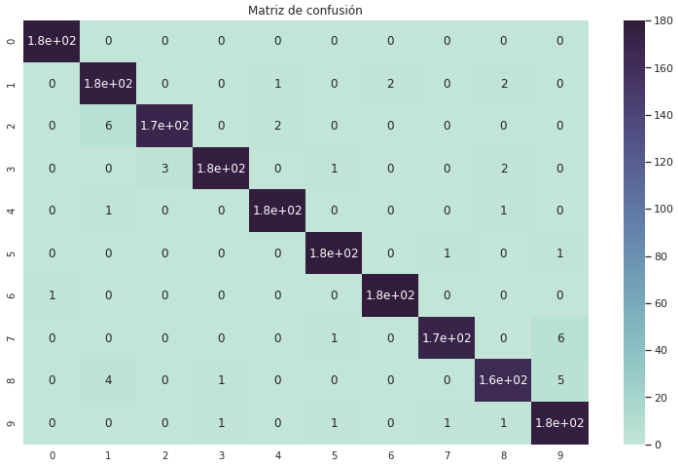


Figura 9: Matriz de confusión de $M_{SVMClass}$ para test.

La matriz de confusión nos muestra las etiquetas predichas (eje x) y las verdaderas (eje y). Observamos que la mayoría de datos se encuentran en la diagonal, lo cual nos asegura que nuestro modelo realiza una buena clasificación.

- Y la curva de aprendizaje del modelo ganador

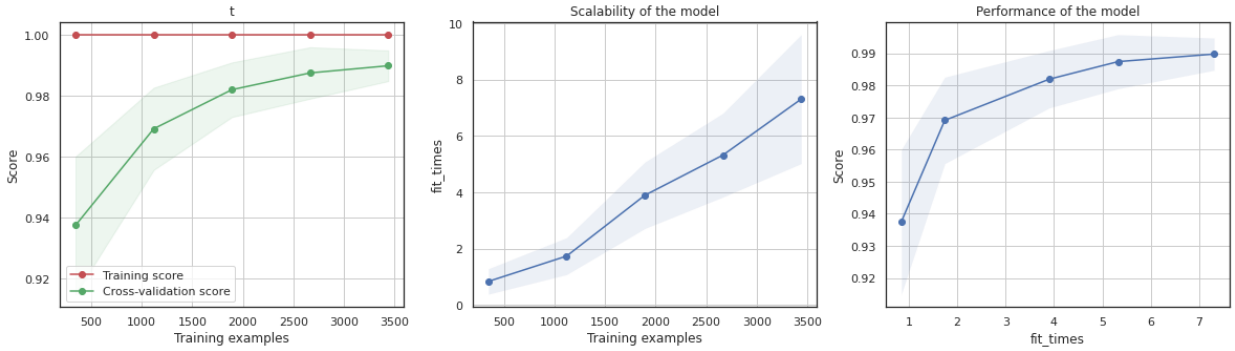


Figura 10: Curva aprendizaje, escalabilidad y desempeño del modelo ganador $M_{SVMClass}$. (código para la generación de la gráfica tomado de https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html)

en la que observamos que: (1) nuestro modelo consigue constantemente clasificar muy bien el conjunto de entrenamiento, (2) el accuracy en validación cruzada asciende en forma logarítmica conforme aumenta el conjunto de entrenamiento, cuanto mayor sea la muestra de entrenamiento más se aprende. Por otra parte, en la escalabilidad del modelo, vemos que cuantas más muestras de entrenamiento más tiempo en ajustar, asciende de forma lineal por lo cual consideramos que no tiene una muy mala escalabilidad. En el desempeño del modelo podemos observar que cuanto más tiempo de ajuste mayor accuracy pero las mejoras cada vez son menores.

Además, podemos dar una cota de E_{out} basada en E_{test} mediante la desigualdad de Hoeffding. Ahora ya tenemos 1 función hipótesis ganadora fijada $\mathcal{H} = |\{g\}| = 1$, por lo que, tomando $\delta = 0.05$ y valorando métrica accuracy

$$E_{out}(g) \leq \underbrace{0.02504174}_{E_{test}(g)} + \sqrt{\frac{1}{2 \cdot 1797} \log\left(\frac{2}{0.05}\right)} \quad \textit{con probabilidad } \geq 1 - 0.05$$

esto es, tenemos que $E_{out}(g) \leq 0.04615475961$ con probabilidad ≥ 0.95.

Podemos comprobar que se tomó la función hipótesis con menor error estimado fuera de la muestra viendo el desempeño de los otros modelos en test

Modelo	Accuracy en test	Accuracy en entrenamiento
$M_{SVMClass}$	0.97495826	1
$M_{SGDClass}$	0.97273233	1
$M_{MLPClass}$	0.96382860	1
$M_{RandForestClass}$	0.97161937	1
Aleatorio	0.10294936004451864	0.09285901124771122

Cuadro 2: Accuracies en entrenamiento y test de los modelos una vez entrenados sobre toda la muestra de entrenamiento. Añadimos accuracy de modelo aleatorio, se aprecia diferencia sustancial.

Comprobamos que $M_{SVMClass}$ es el que mejor accuracy tiene en nuestra estimación fuera de la muestra (en test).

Por último cabe mencionar que se realizó todo este mismo procedimiento eliminando solo las características con varianza 0 (había 1) y, ajustando debidamente los hiperparámetros de la misma forma, los accuracies medios por 10-fold cross validation obtenidos para modelo SVM, Reg. Logística, MLP y Random Forest, fueron 0.98718234, 0.98064263, 0.98116824 y 0.97960234 respectivamente; todos inferiores a los obtenidos eliminando características con varianza menor a 0.05. Para los accuracies en test ocurre lo mismo: 0.97106288, 0.96772398, 0.96828047 y 0.97440178 respectivamente, todos inferiores a 0.97495826. Asegurándonos así que nuestro preprocesamiento no perjudicó a los resultados.

12. Referencias y enlaces.

- Learning From Data by Yaser S. Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin
- https://scikit-learn.org/stable/
- https://www.cs.toronto.edu/~jlucas/teaching/csc411/lectures/lec10_handout.pdf
- https://arxiv.org/pdf/1412.6980.pdf