Aprendizaje automático

Práctica 3: Ajuste de modelos lineales

Antonio Coín Castro

Curso 2019-20

Índice

Introducción

En esta práctica perseguimos ajustar el mejor modelo lineal en dos conjuntos de datos dados, para resolver un problema de clasificación y otro de regresión. En ambos casos seguiremos una estrategia común que nos llevará finalmente a elegir un modelo y estimar su error.

1. Analizaremos la bases de datos y entenderemos el contexto del problema a resolver.
2. Preprocesaremos los datos de forma adecuada para trabajar con ellos.
3. Elegiremos una clase de hipótesis (lineal) para resolver el problema.
4. Fijaremos algunos modelos concretos y seleccionaremos el mejor según algún criterio.
5. Estimaremos el error del modelo.

Trabajamos en su mayoría con las funciones del paquete `scikit-learn`, apoyándonos cuando sea necesario en otras librerías como `numpy`, `matplotlib` ó `pandas`. El código de la práctica se ha estructurado en tres *scripts* de `Python` debidamente comentados:

- En `p3_classification.py` se resuelve el problema de clasificación.
- En `p3_regression.py` se resuelve el problema de regresión.
- En `p3_visualization.py` se recogen todas las funciones de visualización de gráficas, tanto comunes como propias de cada problema.

La ejecución de los dos programas principales está preparada para que se muestren solo algunas gráficas además del procedimiento de ajuste del modelo (aquellas que consumen menos tiempo). Este comportamiento puede cambiarse mediante el parámetro `show` de la función principal en cada caso, eliminando todas las gráficas (valor 0) o mostrándolas todas (valor 2). Además, en las operaciones de cálculo intensivo que lo permitan se utiliza el parámetro `n_jobs = -1` para paralelizar el flujo de trabajo en tantas hebras como se pueda. Por pantalla se muestra información sobre el tiempo de ejecución de los ajustes de los modelos y del programa completo.

Para que los experimentos sean reproducibles se fija una semilla aleatoria al incio del programa. Todos los resultados y gráficas que se muestran a continuación se han obtenido con el valor 2020 para la semilla, y pueden reproducirse si se ejecuta el programa tal y como se proporciona.

Bases de datos y descripción del problema

Ambas bases de datos se han descargado del repositorio UCI.

Clasificación: *optical recognition of handwritten digits*

El primer conjunto de datos es un conjunto para clasificación. Se trata de un conjunto de 5620 dígitos manuscritos codificados en 64 atributos de tipo entero, al que contribuyeron 43 personas diferentes escribiendo dígitos. Cada dígito se escaneó con una resolución de 32×32, siendo cada píxel blanco o negro. Posteriormente, para cada bloque 4×4 no solapado se calculó el número de píxeles negros que hay en ese bloque, obteniendo finalmente 64 atributos que tienen un valor entero entre 0 y 16. La salida o variable de clase es un atributo categórico entre 0 y 9 que representa el dígito dibujado.

Por tanto, como disponemos de las etiquetas reales de cada instancia estamos ante un problema de aprendizaje supervisado; en particular, un problema de **clasificación multiclase**. En principio, consideramos como espacio muestral $\mathcal{X} = \{0, \dots, 16\}^{64}$ y como conjunto de etiquetas $\mathcal{Y} = \{0, \dots, 9\}$. Queremos aprender o estimar una función $f : \mathcal{X} \to \mathcal{Y}$ que asigne a cada muestra codificada como hemos dicho anteriormente su correspondiente dígito.

Podemos visualizar inicialmente la distribución de clases, tanto en el conjunto de entrenamiento como en el de *test*. Aunque ya nos lo decían en el informe de la base de datos, podemos observar en la Figura 1 que la distribución de clases es uniforme en ambos conjuntos; no hay problema de clases desbalanceadas.

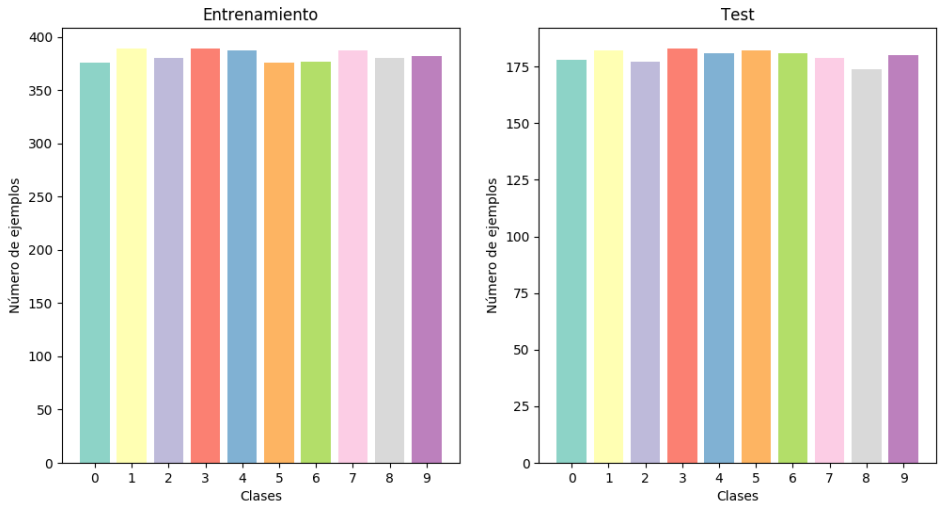


Figura 1: Distribución de clases en entrenamiento y *test* para el problema de clasificación.

Para intentar visualizar los datos, podemos emplear la técnica TSNE para visualizar datos de alta dimensión. Este algoritmo realiza una proyección en 2 dimensiones del conjunto de datos, minimizando una métrica estadística conocida como *divergencia de Kullback-Leibler*. En la Figura 2 podemos ver el resultado de su aplicación, donde se

observan 10 grupos claramente diferenciados correspondientes a las 10 clases de dígitos. En base a esto podemos pensar que los atributos contienen suficiente información para poder construir un buen clasificador, y por tanto esperamos unos resultados bastante buenos.

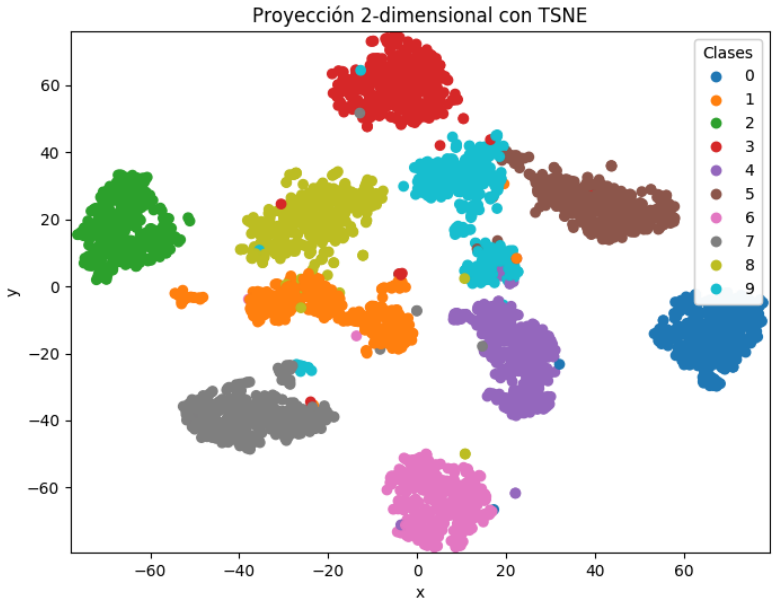


Figura 2: Proyección en 2 dimensiones del conjunto de entrenamiento con TSNE para el problema de clasificación.

Regresión: *communities and crimes*

El segundo conjunto de datos con el que trabajamos es un conjunto para regresión. Esta vez se trata de 1994 instancias que recogen información de tipo muy variado sobre ciertas regiones de Estados Unidos, para intentar predecir a partir de ellas el número total de crímenes violentos por cada 100.000 habitantes en dicha región. Por ejemplo, se recoge información sobre el número de habitantes, su raza, la renta *per cápita* o el porcentaje de familias divorciadas, entre otros.

Cada ejemplo consta de 128 atributos, de donde los cinco primeros son no predictivos (*state*, *county*, *community*, *communityname* y *fold*), los 122 atributos son predictivos (con valores reales) y el último, de tipo real, es la variable que queremos predecir (`ViolentCrimesPerPop`). Se trata entonces de un problema de aprendizaje supervisado; en particular, un problema de **regresión**. Nos dicen que todos los datos están normalizados al intervalo $[0, 1]$, por lo que consideramos $\mathcal{X} = [0, 1]^{122}$, $\mathcal{Y} = [0, 1]$ y

queremos aprender una función $f : \mathcal{X} \to \mathcal{Y}$ que asigne a cada ejemplo la cantidad (normalizada) de crímenes violentos que se producen por cada 100.000 habitantes.

Para intentar visualizar los datos hemos seguido dos estrategias. Por un lado, se han seleccionado (mediante el criterio de importancia que otorga un `Random Forest`) las dos variables con mayor relevancia y poder predictivo, que resultan ser el tanto por uno de niños de 4 años o menos en familias biparentales (`PctYoungKids2Par`) y el tanto por uno de hombres que nunca han estado casados (`MalePctNevMarr`). Una representación de las mismas junto a la variable a predecir puede verse en la Figura 3, donde observamos que hay una correlación substancial (en el primer caso positiva y en el segundo negativa).

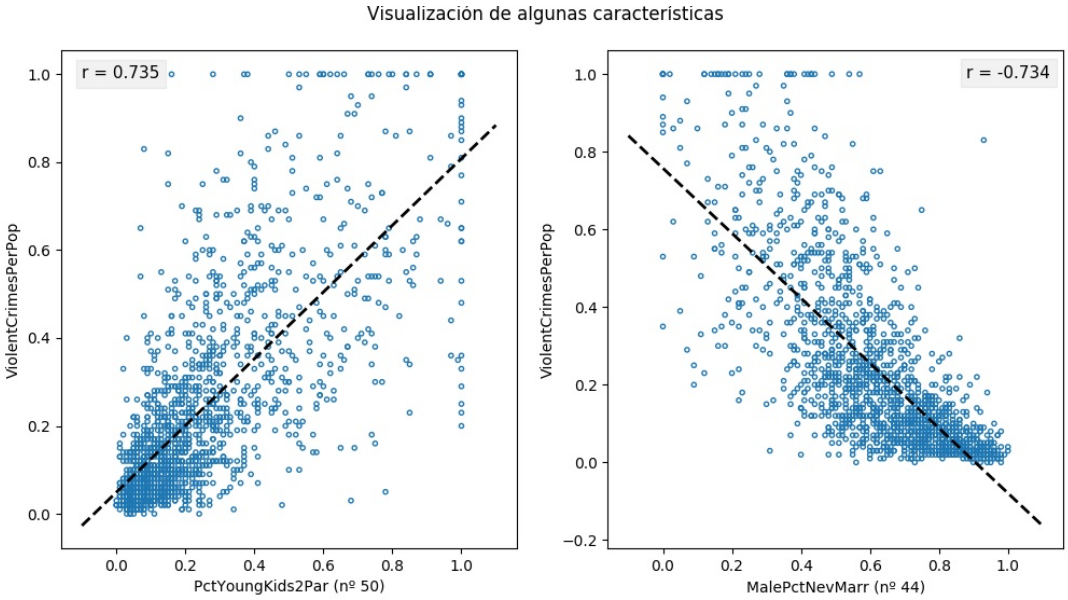


Figura 3: Representación de algunas características frente a la variable a predecir, junto al coeficiente de correlación lineal.

Por otro lado, podemos estudiar cómo es la distribución de la variable a predecir. Para ello realizamos un histograma de la misma en la Figura 4, donde vemos que en la mayoría de ejemplos el número de crímenes violentos se encuentra en la parte inferior del espectro, mientras que un número muy alto de crímenes es relativamente raro.

Selección de la clase de funciones

Como veremos en la sección Preprocesado de datos, emplearemos una técnica de selección de variables para reducir la dimensionalidad del problema y evitar los inconvenientes que un elevado número de variables conlleva. Así podremos seleccionar una clase de

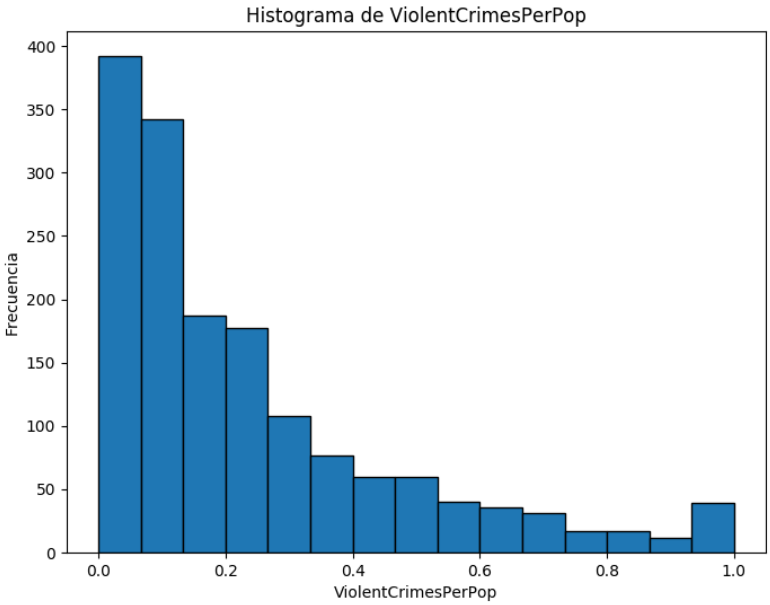


Figura 4: Histograma de la variable `ViolentCrimesPerPop`.

funciones de combinaciones cuadráticas de las observaciones sin elevar de forma exagerada la dimensión de cada punto. De esta manera aumentamos la flexibilidad del modelo, pues presuponemos que los datos no serán linealmente separables en el espacio original, pero podemos aspirar a que lo sean en el espacio transformado. Estamos pagando un precio al aumentar también la complejidad del conjunto de hipótesis, pero en ambos casos mi opinión es que los problemas a los que nos enfrentamos son lo suficientemente complejos como para que esté justificado el empleo de tranformaciones polinómicas. También corremos el riesgo de caer en el sobreajuste, pero añadiremos a nuestros modelos técnicas de regularización que ayuden a paliar este problema.

No considero que aumentar el grado de la transformación polinómica por encima de 2 vaya a resultar en un mejor ajuste, y además ya tendríamos un número excesivo de componentes que se traduciría en un mayor tiempo de ejecución y un riesgo más elevado de sobreajuste.

Como hemos comentado vamos a realizar selección de variables, por lo que si el espacio original era d-dimensional, de cara a expresar la clase de funciones trataremos con un espacio $\mathcal{X}'$ de dimensión d' ($d' \leq d$) genérico, resultado de la eventual selección que hagamos. Consideramos entonces para cada $x = (x_1, \ldots, x_{d'}) \in \mathcal{X}'$ la transformación

$$\Phi_2(x) = (1, x_1, \ldots, x_{d'}, x_1^2, \ldots, x_{d'}^2, x_1x_2, x_1x_3, \ldots, x_1x_{d'}, x_2x_3, \ldots, x_{d-1}x_{d'}),$$

que incluye el término de *bias* al principio, resultando en que el espacio transformado $\mathcal{Z} = \Phi_2(\mathcal{X}')$ tiene dimensión

$$\tilde{d} = 1 + 2d' + \frac{d'(d'-1)}{2}.$$

Clasificación

Como estamos ante un problema de clasificación multiclase, debemos hacer una serie de ajustes para expresar la clase de hipótesis, empleando la técnica de clasificadores *one-versus-all*. En primer lugar, consideramos para cada $i \in \{0, \ldots, 9\}$ el clasificador binario en el espacio transformado $\mathcal{Z}$ dado por

$$h_i(z) = w^T z, \quad w \in \mathbb{R}^{\tilde{d}},$$

que salvo un reescalado por $||w||^{-1}$ mide la distancia con signo del punto z al hiperplano definido por w. Interpretaremos que una distancia positiva se corresponde a un etiquetado a favor de la i-ésima clase, y una distancia negativa a *cualquiera de las otras*. A la hora de predecir la clase de una instancia, digamos $c(z)$, elegiremos aquella clase para la cual la clasificación positiva sea más fuerte, o en caso de no haber ninguna, aquella que se quede más cerca de la frontera de clasificación. Esto se puede expresar como

$$c(z) = \arg\max_i \ h_i(z).$$

Por tanto, la clase de funciones en este caso quedaría

$$\mathcal{H}_C = \left\{ \arg\max_i w_{(i)}^T \Phi_2(x) : w_{(i)} \in \mathbb{R}^{\tilde{d}},\ i = 0, \ldots, 9 \right\}.$$

Notamos que estamos considerando un clasificador binario por cada clase, de forma que el número total de parámetros del modelo sería $10\tilde{d}$.

Regresión

En este caso la clase de funciones no requiere ningún tratamiento especial; se trata simplemente de las combinaciones lineales en el espacio transformado $\mathcal{Z}$:

$$\mathcal{H}_R = \left\{ w_0 + \sum_i w_i x_i + \sum_{i \leq j} w_{ij} x_i x_j : w_i, w_{ij} \in \mathbb{R} \right\} = \left\{ w^T \Phi_2(x) : w \in \mathbb{R}^{\tilde{d}} \right\}.$$

Conjuntos de entrenamiento, validación y *test*

La estrategia para la selección de modelos será usar *K-fold cross validation*, por lo que no es necesario considerar ningún conjunto de validación explícito.

Clasificación

En este caso, el conjunto de datos viene ya dividido en entrenamiento y *test*: disponemos de 3823 ejemplos de entrenamiento (sobre un 70 %) y 1797 ejemplos de *test* (sobre un 30 %). No veo ningún motivo para alterar esta división; de hecho, nos dicen que los dígitos manuscritos en ambos conjuntos fueron dibujados por personas distintas, por lo que es deseable que esta división se mantenga. De esta forma cuando evaluemos el modelo no solo intentará predecir ejemplos que no ha visto antes, sino que estos habrán sido dibujados por personas distintas a aquellos con los que fue entrenado.

Regresión

En este caso, debemos establecer nosotros la separación en entrenamiento y test. Debido a que no tenemos un número excesivo de ejemplos, se ha decidido hacer una división de 80 % para entrenamiento y 20 % para *test*, mediante una llamada a la función `train_test_split`.

Preprocesado de datos

Hemos visto que en ambos conjuntos los datos ya están normalizados en el sentido de que todas las columnas se mueven en el mismo rango. Sin embargo, para aumentar nuestras posibilidades de obtener un buen ajuste debemos realizar un preprocesado adicional de los datos. Es importante que cualquier transformación de los datos se haga primero únicamente en el conjunto de entrenamiento, y después a la hora de evaluar se haga la misma transformación (con los mismos parámetros) en el conjunto de *test*.

Para la fase de preprocesado usaremos los *pipelines* de `sklearn` [1], que nos permiten aplicar las mismas transformaciones en varias fases de forma cómoda. En ambos casos, consideramos cinco pasos en el preprocesado: tratamiento de valores perdidos, selección de variables, transformaciones polinómicas, umbral de varianza y estandarización.

Valores perdidos

Es importante tratar con los valores perdidos, pues en otro caso no podremos entrenar correctamente ningún modelo.

Clasificación

En el conjunto de dígitos no hay ningún valor perdido, por lo que este paso podemos saltarlo. Tampoco encontramos valores que podamos considerar inconsistentes.

Regresión

En esta ocasión nos encontramos que 22 columnas tienen más de un 80 % de valores perdidos. Decidimos eliminar directamente los predictores asociados de nuestro conjunto de datos, pues difícilmente vamos a poder establecer ninguna relación útil para predecir. Tras esto, nos queda un único valor perdido en uno de los ejemplos, al que decidimos atribuirle la mediana de su columna. Por un lado, decidimos rellenar el valor porque no tenemos ejemplos de sobra y la información que aportan el resto de sus atributos puede ser útil, y por otro decidimos usar la mediana como estrategia por su robustez ante posibles *outliers*.

Para atribuir los valores perdidos utilizamos la transformación `SimpleImputer`, especificando en sus parámetros que la estrategia de atribución es la mediana. Además de lo ya comentado, no encontramos valores inconsistentes o extraños en el conjunto.

[1] Documentación sobre `Pipeline` en `sklearn`.

Selección de características

Como ya comentamos, reducir la dimensionalidad de los datos es un paso muy importante, pues además de una reducción en el tiempo de entrenamiento de los modelos, podemos eliminar características poco relevantes y/o con alta correlación que haga que la calidad de los modelos aumente. También podemos combinar las características para obtener otras con mayor poder predictivo. En general, el fenómeno que hace que los modelos en *machine learning* empeoren o sean casi inviables de utilizar conforme aumenta la dimensión se conoce como *maldición de la dimensionalidad*.

En cualquier caso, se considera la opción de no realizar selección de características de ningún tipo, para constatar *a posteriori* que efectivamente obtenemos una mejoría en los resultados, además de en el tiempo de ejecución.

Clasificación

A la hora de realizar selección de variables se han considerado dos alternativas: análisis de componentes principales (PCA) ó análisis de varianza (ANOVA). La primera se trata de una técnica de reducción de dimensionalidad que considera combinaciones lineales de las variables, llamadas componentes principales, de forma que la primera recoge la mayor varianza según una cierta proyección de los datos, la segunda la segunda mayor, etc. La segunda alternativa, por su parte, basa su funcionamiento en el test estadístico conocido como *F-test*, que en este contexto se utiliza para estimar el grado de dependencia lineal de las variables dos a dos. Una vez hecho esto, se ordenan las variables de la más a la menos discriminativa.

En el caso de PCA establecemos que la varianza acumulada de las variables resultantes debe ser del 95 % de la original, obteniendo **29 variables** tras su aplicación. Para el test ANOVA imponemos que se seleccionen de entre todas las variables las 32 que mejor puntuación obtengan (la mitad).

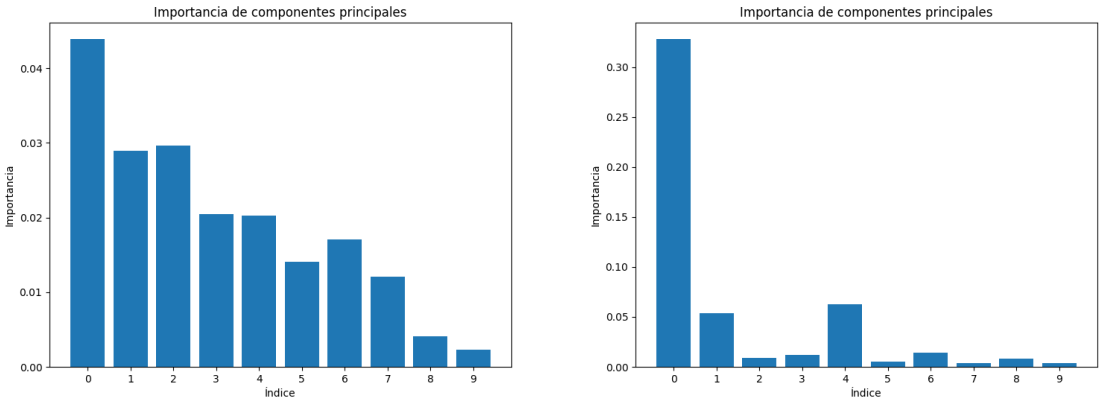
Se han elegido estas dos estrategias como representantes de los dos grandes campos en la selección de variables: PCA realiza una selección *no supervisada*, mientras que el análisis de varianza es un método *supervisado* de selección (es decir, utiliza la información de las etiquetas). Se estimó que era interesante realizar una comparación entre estos métodos para ver cuál de ellos tenía un mejor desempeño.

Regresión

En el caso de regresión, como los datos recogidos son de naturaleza muy dispar y no tan uniformes como en el caso de los dígitos, optamos por fijar directamente una aplicación de PCA con un valor más agresivo de reducción, imponiendo únicamente que se tenga que explicar el 80 % de la varianza del conjunto. Fijamos un valor más pequeño porque intuimos que puede haber muchas fuentes de variabilidad distinta en los datos, y que el porcentaje de ruido puede ser mayor.

Podemos ver en la Figura 5 cómo se refleja la transformación de componentes princi-

pales que realiza PCA en el criterio de importancia predictiva de las variables, medido de nuevo a través de un `Random Forest`. Observamos que efectivamente la primera componente principal es en ambos caso la que más relevancia tiene, y en el caso del problema de clasificación se aprecia mejor como en general la relevancia va disminuyendo conforme aumentamos el índice (pues la varianza explicada por cada variable va disminuyendo).



(a) Clasificación.

(b) Regresión.

Figura 5: Relevancia de componentes principales en clasificación y regresión.

Transformaciones polinómicas

Como ya comentamos cuando definimos la clase de funciones, emplearemos transformaciones polinómicas de segundo grado. Simplemente llamaremos a la función `PolynomialFeatures(2)`, que se encarga de realizar las transformaciones pertinentes.

Estandarización y umbral de varianza

En primer lugar, procederemos a eliminar las variables con varianza 0, pues son constantes, no aportan ninguna información y pueden provocar problemas en el ajuste. También eliminaremos aquellas variables con varianza muy pequeña, todo ello utilizando el transformador `VarianceThreshold`.

Por otro lado, un paso muy usual y que proporciona buenos resultados en la práctica es *estandarizar* las variables, pues hay algoritmos que implícitamente asumen que los datos están centrados en 0 y con varianza similar. A cada atributo le restamos la media de la columna y dividimos por la desviación típica, de forma que las variables resultantes quedan centradas en 0 y con varianza unidad. Esto permite que las columnas sean comparables, y es especialmente relevante cuando las variables no están medidas en las mismas unidades, como es el caso del problema de regresión.

Orden de las transformaciones

Como dijimos antes, encadenaremos todas estas transformaciones en un *pipeline*. Para el caso de clasificación tenemos dos opciones, siendo la primera de ellas la siguiente:

```
preproc = [
    ("selection", PCA(0.95)),
    ("standardize", StandardScaler()),
    ("poly", PolynomialFeatures(2)),
    ("var", VarianceThreshold(0.1)),
    ("standardize2", StandardScaler())]
```

Vemos que realizamos primero la selección de variables, después estandarizamos, seguimos con la transformación polinómica y finalmente acabamos con la eliminación de variables con varianza menor a 0.1 y otra estandarización (pues tras la transformación polinómica se han podido perturbar las variables). Decidimos hacer primero la selección para reducir la dimensión sobre el conjunto original, haciendo que por un lado no surjan demasiadas variables tras hacer las combinaciones cuadráticas, y por otro que estas combinaciones partan de una base que explique mejor los datos para poder encontrar relaciones entre variables más fácilmente.

La segunda opción es análoga, sustituyendo el primer elemento del cauce por la otra estrategia de selección. Además añadimos primero un nuevo transformador `VarianceThreshold`, pues en el *F-test* hay problemas si tenemos variables constantes:

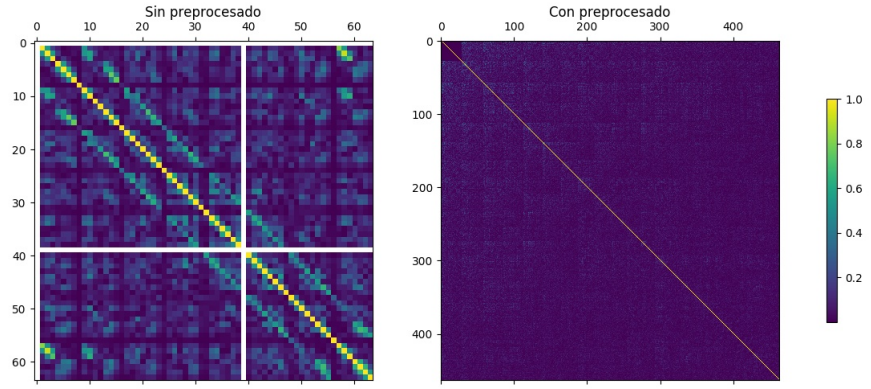
```
preproc = [
    ("var2", VarianceThreshold()),
    ("selection", SelectKBest(f_classif, k = 32)),
    ("standardize", StandardScaler()),
    ("poly", PolynomialFeatures(2)),
    ("var", VarianceThreshold(0.1)),
    ("standardize2", StandardScaler())]
```

Para regresión el cauce queda como en el correspondiente caso de clasificación, sustituyendo el parámetro de PCA por 0.8 y poniendo el umbral de la varianza a 0, pues en este caso muchas de las variables tienen varianza pequeña y perderíamos demasiadas.

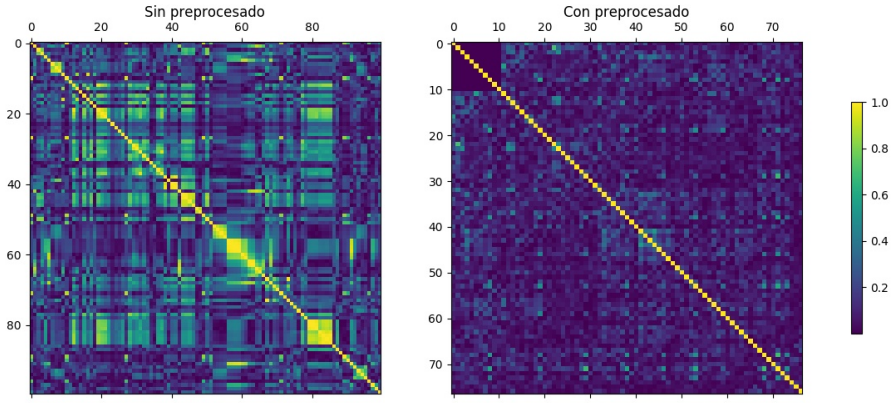
Mostramos finalmente en la Figura 6 un ejemplo de cómo evoluciona la matriz de correlación (en valor absoluto) en ambos problemas. Podemos apreciar que inicialmente hay muchas variables con fuerte correlación (especialmente en el caso de regresión), pero que tras el preprocesado muchas de ellas resultan casi incorreladas, haciendo que disminuya el número de variables potencialmente redundantes.

Métricas de error

Fijamos a continuación las métricas que usaremos para la selección y evaluación de los modelos.



(a) Clasificación.



(b) Regresión.

Figura 6: Matriz de correlaciones absolutas en ambos problemas, antes y después del preprocesado. Las líneas blancas indican variables constantes.

Clasificación

En el caso del problema de clasificación no hay muchas dudas; la métrica por excelencia es el *error de clasificación*, una medida simple pero efectiva del error cometido. Si denotamos los ejemplos de entrada a un clasificador h por (x_n, y_n), podemos expresar el error como la fracción total de elementos mal clasificados:

$$E_{class}(h) = \frac{1}{N}\sum_{i=1}^{N}[\![h(x_n) \neq y_n]\!].$$

Sabemos que esta medida de error se encuentra en el intervalo $[0, 1]$, siendo 0 lo mejor posible y 1 lo peor. Se trata de una medida de fácil interpretación; podemos expresarla en porcentaje o invertirla, de forma que $1 - E_{in}$ es lo que se conoce como el *accuracy* del modelo. Presentaremos los resultados utilizando esta última descripción ya que parece más ilustrativa.

Regresión

En este caso la gama de opciones es más amplia, por lo que decidimos realizar una elección doble. Emplearemos como métrica principal el *error cuadrático medio*, y como métrica secundaria el *coeficiente de determinación* R^2, ambas de uso muy extendido.

El error cuadrático medio (MSE) cuantifica la diferencia entre las predicciones y las etiquetas reales. Se trata de un error que penaliza con mayor severidad los *outliers* en comparación, por ejemplo, al error absoluto medio. Puede expresarse como

$$\text{MSE}(h) = \frac{1}{N}\sum_{n=1}^{N}(y_n - h(x_n))^2.$$

Además, aunque la métrica de error sea el MSE, para mostrar los resultados emplearemos su raíz cuadrada (abreviada RMSE), simplemente porque de esa forma el error estará en unidades de la variable a predecir y facilitará su interpretacón. Aunque esta es una de las métricas de error más usadas, tiene como desventaja que no está acotada superiormente y no tenemos un valor de referencia; solo podemos usarla en general para comparar los ajustes dentro de un mismo conjunto de datos, y no de manera absoluta.

Para tener una medida absoluta de error consideramos también el coeficiente de determinación R^2. Este coeficiente indica la bondad del ajuste, tomando su máximo valor en 1 (ajuste perfecto), y pudiendo tomar también valores negativos (a pesar de su nombre). Su definición es la siguiente:

$$R^2(h) = 1 - \frac{\sum_{n=1}^{N}(y_n - h(x_n))^2}{\sum_{n=1}^{N}(y_n - \bar{y})^2}, \quad \text{con } \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_n.$$

Regularización

El uso de la regularización es esencial para limitar la complejidad del modelo y el *overfitting*, cosa que nos permitirá obtener una mejor capacidad de generalización. En principio se consideran dos posibilidades de regularización:

- **Regularización L2 (Ridge)**: se añade una penalización a la función de pérdida que es cuadrática en los pesos,
$$L_{reg}(w) = L(w) + \lambda||w||_2^2.$$
- **Regularización L1 (Lasso)**: en este caso la penalización es lineal en los pesos, considerando la suma del valor absoluto de los mismos,
$$L_{reg}(w) = L(w) + \lambda\sum_i |w_i|.$$

  En ambos casos el valor de $\lambda > 0$ es un hiperparámetro del modelo, que controla la intensidad de la regularización (a mayor valor, más pequeños serán los pesos). Encontrar un valor adecuado es una tarea complicada, pues si es demasiado pequeño seguiremos teniendo sobreajuste, pero si es demasiado grande podríamos caer en el fenómeno opuesto: tener *underfitting* porque el modelo sea poco flexible y no consiga ajustar bien los datos de entrenamiento.

Clasificación

Elegimos la regularización de tipo L2 frente a L1 porque la primera penaliza con más fuerza los valores grandes, y funciona mejor cuando se cree que todas las variables son relevantes para la predicción. Además, la penalización que se introduce en ese caso es una función derivable de los pesos, que hace que su tratamiento computacional sea más eficiente.

Regresión

Para el caso de regresión seguimos el mismo razonamiento que para clasificación; seleccionamos la pérdida L2. Sin embargo, en este caso también pensamos que merece la pena comparar con la pérdida L1, pues si bien ya se ha realizado una selección de características, el origen diverso de los datos en este conjunto hace que podamos pensar que todavía existan algunas variables que no influyan demasiado en el resultado. La regresión L1 realiza una especie de selección automática de características (obtiene modelos *dispersos* en los que muchos de los pesos van a 0), proporcionando una segunda capa de selección que esperamos mejore la calidad del ajuste. En cualquier caso, consideraremos modelos con ambos tipos de regularización y dejaremos que compitan entre ellos para seleccionar el mejor.

Técnicas y selección de modelos

Pasamos a realizar la selección de modelos para el ajuste. Como ya dijimos, vamos a usar la técnica de *K-fold cross validation* para elegir el mejor modelo. Esta técnica se basa en dividir el conjunto de entrenamiento en K conjuntos de igual tamaño, y va iterando sobre ellos de forma que en cada paso entrena los modelos en los datos pertenecientes a $K-1$ de los conjuntos, y los evalúa en el conjunto restante. Finalmente se realiza la media del error a lo largo de todos los mini-conjuntos de validación y se escoge el modelo con menor error. Este error se conoce como *error de cross-validation*, denotado E_{cv}, y sabemos que es un buen estimador del error fuera de la muestra.

Es importante destacar que debemos considerar un modelo como un estimador junto a un conjunto de parámetros fijo; es decir, pondremos a competir una serie de parejas (estimador, hiperparámetros) de forma que tras la selección obtengamos **el mejor modelo con los mejores parámetros**. Para ello utilizamos la función `GridSearchCV`, la cual puede recibir un *pipeline* como estimador y una lista de diccionarios que represente el espacio de parámetros. En cada uno de los diccionarios podemos fijar el estimador a usar y los parámetros con los que queremos probar. Para evitar el fenómeno de *data snooping* que podría contaminar los conjuntos de validación, todo el cauce de preprocesado y selección de modelos se realiza de principio a fin: fijamos al principio las divisiones en K *folds* y las utilizamos en todo el proceso. Para el caso de clasificación, estas divisiones serán en particular instancias de `StratifiedKFold`, que respeta la distribución de clases en las divisiones.

Una vez hemos encontrado el mejor modelo con los mejores parámetros, **se vuelve a entrenar sobre todo el conjunto de entrenamiento**, para obtener un mejor rendimiento. Este es el comportamiento por defecto de la función `GridSearchCV`.

Clasificación

Comentamos ahora los modelos que pre-seleccionamos en el problema de clasificación para que compitan entre ellos. La métrica usada para decidir el mejor modelo será, de forma natural, el *accuracy* medio en los conjuntos de validación. Fijamos el número máximo de iteraciones en 500 para todos los modelos.

Regresión logística

En primer lugar consideramos un modelo de regresión logística, implementado en el objeto `LogisticRegression`, que sabemos que obtiene en la práctica muy buenos resultados en problemas de clasificación multiclase. Como ya comentamos, fijamos el tipo de regularización a L2, y la estrategia de clasificación multiclase a *one-vs-all*, que es la que usaremos en todos los modelos. El parámetro de regularización, cuyo inverso es lo que en el código se alude como `C`, lo dejaremos a elección del proceso de selección, considerando unos valores posibles de 10^{-4}, 1 y 10^4 (no se consideran más valores para no aumentar en exceso el tiempo de ejecución).

```
{"clf": [LogisticRegression(multi_class = 'ovr',
                            penalty = 'l2',
                            max_iter = max_iter)],
 "clf__C": np.logspace(-4, 4, 3)}
```

En este caso, la técnica de optimización es la que viene por defecto, que se conoce como LBFGS. Se trata de un algoritmo iterativo similar al método de Newton para optimización, pero que utiliza una aproximación de la inversa de la matriz Hessiana. Se ha elegido porque tiene un tiempo de ejecución asumible y los resultados suelen ser buenos. La función a optimizar es la *pérdida logarítmica*:

$$L_{log}(w) = \frac{1}{N}\sum_{n=1}^{N} \log(1 + e^{-y_n w^T x_n}),$$

a la que se añade el término de penalización L2. Debemos tener en cuenta que aunque el algoritmo optimice esta función para proporcionar un resultado, la métrica de error que nosotros estamos usando es el *accuracy* y no el error logarítmico.

Regresión lineal

Consideramos también un modelos de regresión lineal, como ya hicimos en las prácticas anteriores. Utilizamos un objeto `RidgeClassifier`, que fija implícitamente la regularización L2. En este caso, la constante de regularización se llama `alpha`, y de nuevo dejamos que varíe en el mismo rango que en regresión logística.

```
{"clf": [RidgeClassifier(random_state = SEED,
                         max_iter = max_iter)],
 "clf__alpha": np.logspace(-4, 4, 3)}
```

En este caso se pretende minimizar el error cuadrático de regresión (añadiendo el correspondiente término de regularización):

$$L_{lin}(w) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w^T x_n)^2.$$

Para ello se utiliza la técnica de la Pseudoinversa basada en la descomposición SVD de la matriz de datos, obteniendo una solución en forma cerrada y sin seguir un procedimiento iterativo. Se ha elegido esta técnica en lugar de SGD porque el tiempo de ejecución es más reducido y las soluciones obtenidas son suficientemente buenas.

Perceptrón

Por último, comparamos con un modelo perceptrón, llamado `Perceptron()` en el código. Establecemos la penalización L2, con el parámetro `alpha` en las mismas condiciones que en regresión lineal.

```
{"clf": [Perceptron(penalty = 'l2',
                    random_state = SEED,
                    max_iter = max_iter)],
 "clf__alpha": np.logspace(-4, 4, 3)}
```

La técnica usada es el algoritmo de optimización iterativa SGD, con un *learning rate* constante de 1 y la pérdida asociada al perceptrón (más un término de regularización):

$$L_{pla}(w) = \frac{1}{N}\sum_{n=1}^{N}\text{máx}(0, -y_n w^T x_n).$$

Esta es la única técnica que proporciona la API de `sklearn` para este modelo, pero sabemos que es equivalente (salvo la condición de parada) al algoritmo perceptrón clásico.

Regresión

En el problema de regresión, la métrica que utilizaremos para comparar los modelos y elegir el mejor será nuestra métrica principal: el MSE. Sin embargo, como la implementación subyacente de `GridSearchCV` busca siempre maximizar la métrica que se le proporciona, le pasamos el parámetro `neg_mean_squared_error` (el MSE negado). Fijamos el número máximo de iteraciones en 2000.

Ahora solo tenemos un tipo de modelo a elegir: la regresión lineal. Lo que haremos será diferenciar tres modelos, según la técnica de regularización y de optimización. Todos ellos tratan de minimizar el error cuadrático medio. Como se ha observado que los tiempos de ejecución en este caso son menores, se permite seleccionar hasta 10 parámetros equiespaciados en escala logarítmica entre 10^{-4} y 10^4 para la constante de regularización.

```
{"reg": [SGDRegressor(penalty = 'l2',
                      max_iter = max_iter,
                      random_state = SEED)],
 "reg__alpha": np.logspace(-4, 4, 10)},
{"reg": [Ridge(max_iter = max_iter)],
 "reg__alpha": np.logspace(-4, 4, 10)},
{"reg": [Lasso(random_state = SEED,
               max_iter = max_iter)],
 "reg__alpha": np.logspace(-4, 4, 10)}
```

Así, tenemos un primero modelo con pérdida L2 y técnica SGD de minimización de la función de pérdida. Después consideramos otro modelo con el mismo tipo de regularización L2, pero esta vez usando la técnica de la Pseudoinversa que comentábamos antes. Por último, consideramos un modelo con pérdida L1 y técnica de optimización basada en descenso de coordenadas, cuya eficiencia es conocida y hace que sea usado en la práctica.

Análisis de resultados y estimación del error

Veamos ahora cuál ha sido el resultado de la selección de modelos y el desempeño del modelo final. Como ya comentamos, el proceso de selección tal y como fue descrito en la sección anterior se materializa con una llamada a la función `GridSearchCV`, donde usaremos $K = 5$. Se proporciona una medida del tiempo de ejecución de los procesos de ajustes de modelos en mi máquina (`Intel i5 7200U (4) @ 3.1GHz`).

Clasificación

Para el problema de clasificación, estudiamos el desempeño diferenciando las dos posibles estrategias de selección que teníamos. Utilizando PCA, el modelo triunfador ha sido el de **regresión logística**, con un valor de 1.0 para la constante de regularización, obteniendo un accuracy en *cross-validation* de $98.718\,\%$ y usando finalmente 464 variables para el ajuste. Tras reentrenar el modelo y evaluarlo, obtenemos un *accuracy* del $100\,\%$ en entrenamiento y un nada desdeñable $98.831\,\%$ en el conjunto de *test*.

Por otro lado, utilizando la estrategia de selección basada en el test ANOVA, obtenemos de nuevo que el mejor modelo es el de regresión logística, con la misma constante de regularización y usando 560 variables. Esta vez conseguimos un *accuracy* de $98.561\,\%$ en *cross-validation*, de $100\,\%$ en entrenamiento, y de $97.551\,\%$ en *test*.

Por último, mencionamos que se ha probado a ejecutar el programa sin realizar ningún tipo de selección de variables. El resultado es el que ya podemos adivinar: la regresión logística se impone de nuevo, esta vez con una constante de regularización de 10^{-4} y 1935 variables. Los resultados son un $98.692\,\%$ de *accuracy* en *cross-validation*, un $100\,\%$ en entrenamiento y un $98.052\,\%$ en *test*, pero el tiempo de ejecución aumenta de forma considerable.

Vemos un resumen de los resultados obtenidos, expresándolos ahora en base al error y no al *accuracy*:

	E_{cv}	E_{in}	E_{test}	Tiempo
PCA	1.282	0.000	1.169	10.962
ANOVA	1.439	0.000	2.449	20.625
Sin selección	1.308	0.000	1.948	69.435

Tabla 1: Tabla de resultados para el problema de clasificación. Los errores se expresan en porcentaje, y el tiempo en segundos.

Vemos que la versión sin selección de variables obtiene resultados ligeramente mejores en *test* que la estrategia de selección ANOVA; sin embargo, el número exesivo de variables y el alto tiempo de ejecución hacen que descartemos esta opción por completo. Vemos que con la estrategia de selección con PCA obtenemos los mejores resultados, por lo que fijamos este modelo como nuestra hipótesis definitiva g.

Presentamos ahora una serie de gráficos para ilustrar el proceso de entrenamiento y los resultados. En primer lugar, vemos en la Figura 7 la matriz de confusión del clasificador en el conjunto de *test*, que muestra la correspondencia entre las etiquetas predichas y las reales para cada clase; en particular, en la diagonal se muestran los aciertos, y en el resto de posiciones los fallos. Como podemos observar, los errores se producen en apenas unos pocos ejemplos, y como anécdota podemos comentar que los dígitos que más se confunden son el 1 y el 8 (4 instancias mal clasificadas en total).

Podemos intentar visualizar el ajuste en dos dimensiones, empleando para ello las dos primeras componentes principales. Proyectamos el conjunto de *test* (tras normalizarlo) sobre sus primeras dos componentes, y lo etiquetamos según las etiquetas predichas por nuestro clasificador, obteniendo una gráfica como la de la Figura 8. Vemos cómo hay 10 grupos más o menos diferenciados, si bien se observa algo de solapamiento, pero concluimos que hemos tenido éxito en nuestra tarea de separar las clases.

También podemos volver a proyectar sobre las dos primeras componentes principales, esta vez coloreando los puntos según sus etiquetas reales, y hacer lo propio con tres clasificadores elegidos arbitrariamente. El resultado será una gráfica como la de la Figura 9, en la que podemos distinguir la proyección de las fronteras de clasificación y ver que más o menos separan los datos en dimensión 2 de acuerdo a sus verdaderas etiqutas.

Finalmente, podemos mostrar en la 10 una gráfica de la *curva de aprendizaje* de nuestro modelo. Esta curva consiste en ir entrenando el modelo con porciones incrementales de nuestro conjunto de entrenamiento, reservando en cada caso un subconjunto de validación para ir evaluándolo en paralelo.

Podemos observar esta curva de aprendizaje en la primera gráfica, donde vemos que el *accuracy* en entrenamiento comienza en 0, y va disminuyendo ligeramente conforme

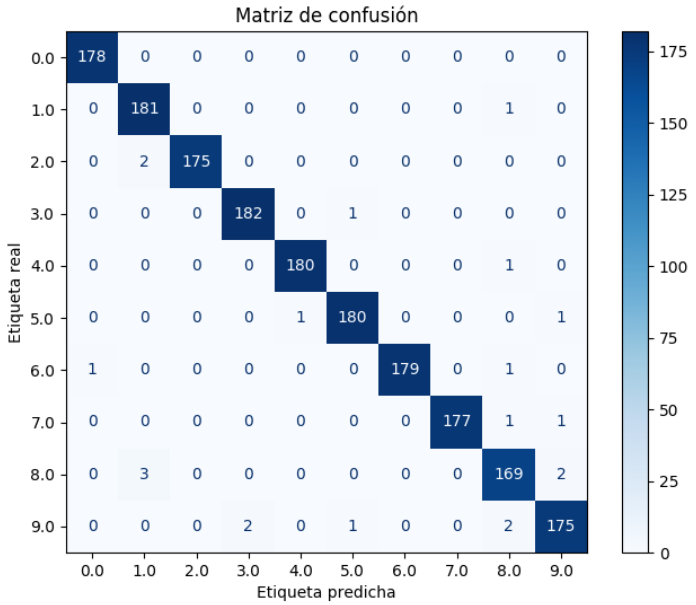


Figura 7: Matriz de confusión para el mejor clasificador en el conjunto de *test*.

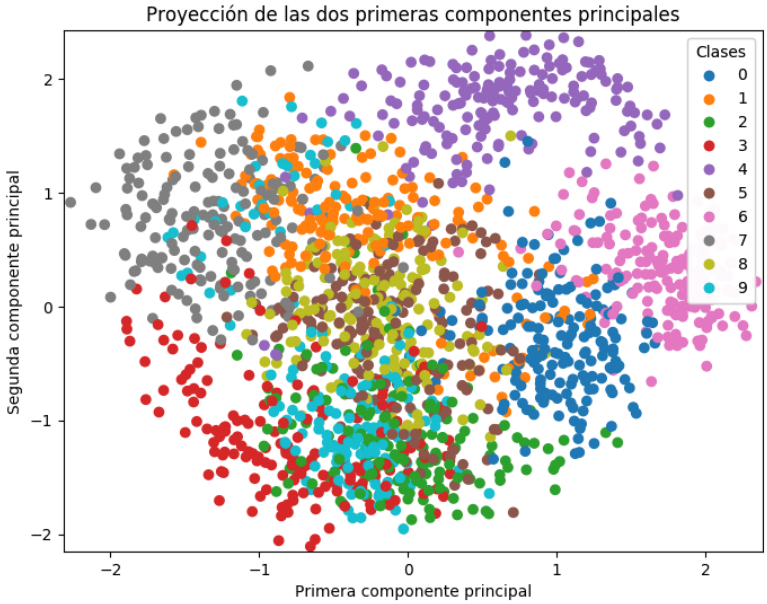


Figura 8: Proyección de las dos primeras componentes principales con etiquetas predichas.

aumentamos el número de ejemplos. Por su parte el *accuracy* de validación va avanzando de manera inversa: va mejorando conforme aumentamos el número de ejemplos. Este es el comportamiento esperado, y de hecho podemos concluir que nuestro modelo aún no ha llegado al punto de saturación en el aprendizaje y se beneficiaría de disponer de maś

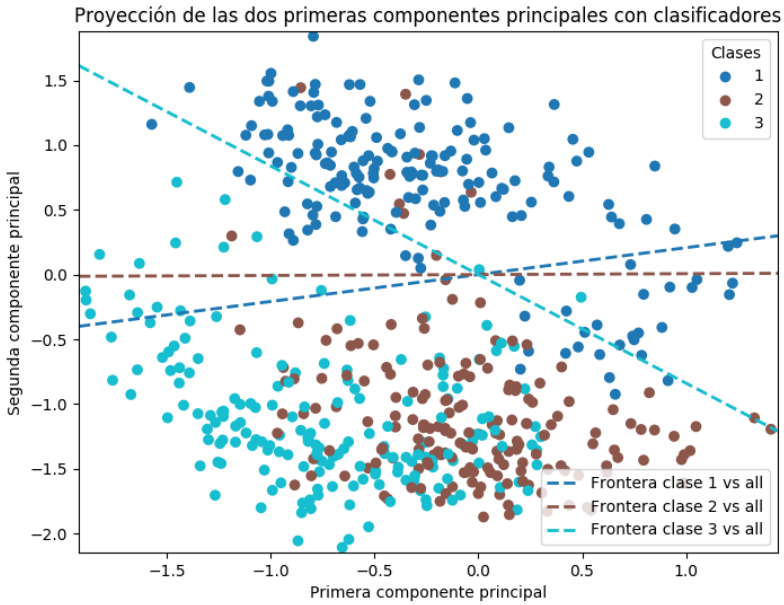


Figura 9: Proyección de las dos primeras componentes principales y de las fronteras de clasificación para las clases 1, 2 y 3, con etiquetas reales.

ejemplos para entrenar, pues todavía no ha alcanzado el punto en el que el *accuracy* en validación comienza a disminuir. Notamos que la escala está aumentada para que veamos bien las diferencias, pero en realidad si viésemos el *accuracy* en escala $[0, 1]$ apreciaríamos que las dos curvas están muy pegadas.

Las dos gráficas restantes nos dan una idea de la escalabilidad y el rendimiento del modelo, midiendo el tiempo de entrenamiento frente al número de ejemplos, y el *accuracy* frente al tiempo de entrenamiento, respectivamente. En el primer caso podemos ver cómo el tiempo de ejecución crece de manera aproximadamente lineal con el número de ejemplos, y en el segundo cómo el crecimiento del *accuracy* respecto al tiempo de entrenamiento presenta un comportamiento más o menos logarítmico.

Estimación del error

Finalmente pasamos a estimar el error del modelo. Como ya comentamos al principio de la sección Técnicas y selección de modelos, el error de *cross-validation* es un buen estimador del error fuera de la muestra, y en general podremos esperar que $E_{out} \leq E_{cv}$ (aunque no está garantizado). Sin embargo, la mejor medida que podemos hacer en estas condiciones del error de generalización viene a partir de E_{test}, pues el conjunto de *test* no se ha usado en ninguna de las fases del entrenamiento y la selección de modelos, y por tanto nos proporciona una buena estimación de E_{out}. En este caso particular, como estamos utilizando como métrica el error de clasificación, es de aplicación la *cota*

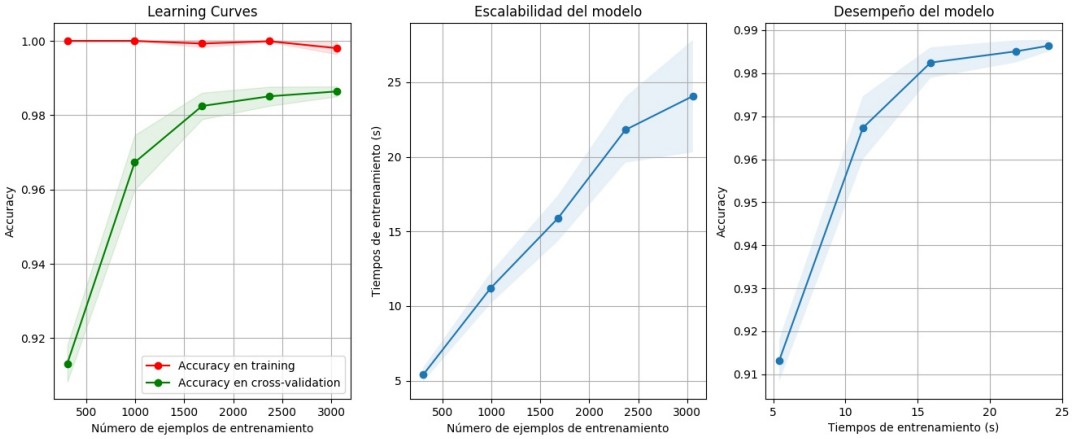


Figura 10: Curvas de aprendizaje, escalabilidad y rendimiento para el clasificador escogido.

de Hoeffding que conocemos para el error en *test*, a saber:

$$E_{out}(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N_{test}} \log \frac{2}{\delta}}, \quad \text{con probabilidad } \geq 1 - \delta.$$

Por ejemplo, fijando $\delta = 0.05$ podemos garantizar con un 95 % de confianza que

$$E_{out} \leq 1.169 + \sqrt{\frac{1}{2 \cdot 1797} \log \frac{2}{0.05}} \approx 1.20.$$

Es decir, si una empresa nos hubiera encargado realizar este ajuste, le diriamos que el modelo proporcionado tiene un error del 1.2 % con un 95 % de confianza.

Regresión

Reproducimos ahora los resultados obtenidos para el problema de regresión. En este caso solo tenemos una estrategia de preprocesado, para la cual el modelo ganador ha sido **regresión lineal con penalización L1**, obteniendo un error RMSE de *cross-validation* de 0.139, de 0.131 en entrenamiento y de 0.128 en el conjunto de *test*. La mejor constante de regularización encontrada es `alpha` $\approx$ 0.00078, utilizando finalmente 77 variables en el ajuste. Los resultados obtenidos sin utilizar ningún tipo de selección de variables son similares (ligeramente peores en el conjunto de *test*), pero ni siquiera los consideramos por usar un número demasiado elevado de variables (más de 5000). Por tanto, elegimos como hipótesis definitiva g el modelo de regresión Lasso obtenido, cuyos resultados quedan recogidos en la siguiente tabla:

Vamos a ilustrar también en esta ocasión el proceso de entrenamiento y los resultado con algunas gráficas. En primer lugarm vemos en la Figura {fig:residues} una doble gráfica. La primera es una representación de los *residuos* del ajuste, definidos como las diferencias entre los valores predichos y los valores reales:

$$R(y, h(x_n)) = y - h(x_n)$$

	$RMSE_{cv}$	$RMSE_{in}$	$RMSE_{test}$	R^2_{in}	R^2_{test}	Tiempo
PCA	0.139	0.131	0.128	0.684	0.702	2.599

Tabla 2: Tabla de resultados para el problema de regresión. El RMSE está en unidades de la variable a predecir, el R^2 en esas unidades al cuadrado, y el tiempo en segundo.

Lo ideal es que estos valores estén cercanos a 0, y que no presenten ninguna distribución que haga pensar que tienen cierta información subyacente que no hemos podido capturar en el ajuste. Por otro lado, la segunda gráfica muestra los valores predichos frente a los valores reales, junto con dos rectas: la recta identidad que sería un ajuste perfecto, y la recta que mejor aproxima la nube de puntos.

Conclusiones y justificación

- Recapitular modelos elegidos
- • interpretar los errores.

Clasificador no lineal para comparar (no se realiza preprocesado, explicar por qué, limitar profundidad)

```
--- Clasificador no lineal (RandomForest) ---
Número de árboles: 200
Profundidad máxima: 32
Número de variables usadas: 64
Accuracy en training: 100.000%
Accuracy en test: 97.440%
Tiempo: 1.341s

Clasificador dummy para comparar

--- Clasificador aleatorio ---
Número de variables usadas: 64
Accuracy en training: 10.097%
Accuracy en test: 9.683%
Tiempo: 0.002s
```

Regresión

- núemro de variables pequeño aun tras aumento

```
--- Regresor no lineal (RandomForest) ---
Número de árboles: 200
Número de variables usadas: 100
```

```
RMSE en training: 0.062
R2 en training: 0.929
RMSE en test: 0.134
R2 en test: 0.673
Tiempo: 9.659s

--- Regresor aleatorio ---
Número de variables usadas: 100
RMSE en training: 0.233
R2 en training: 0.000
RMSE en test: 0.234
R2 en test: -0.001
Tiempo: 0.000s
```