UNIVERSIDAD DE GRANADA

Universidad de Granada

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

---

Aprendizaje Automático

Proyecto Final

Human Activity Recognition with Smartphones

Alejandro Pinel Martínez

Pablo Ruiz Mingorance

Junio 2021

Índice general

PARTE 1

Definición del problema y separación de datos

Nuestra base de datos es *Human Activity Recognition Using Smartphones*[1][2]. Esta base consiste en datos extraídos de los sensores de los smartphones de 30 voluntarios que los llevaban consigo mientras realizaban actividades de la vida cotidiana. El problema de clasificación consiste en, dado el vector de atributos de una muestra, asignarle la etiqueta correspondiente a la actividad que el sujeto estaba llevando a cabo en la ventana de tiempo correspondiente.

Concretamente, 30 voluntarios con edades comprendidas entre los 19 y los 48 años realizaron 6 tipos de actividades: andar, subir escaleras, bajar escaleras, sentarse, estar de pie y tumbarse. Estas acciones se realizaron mientras se capturaban las mediciones del acelerómetro y el giroscopio de un smartphone (Samsung Galaxy SII) que llevaban en la cintura. Los datos fueron recogidos en forma de aceleración lineal en 3 ejes y aceleración angular en 3 ejes con un ratio constante de 50Hz.

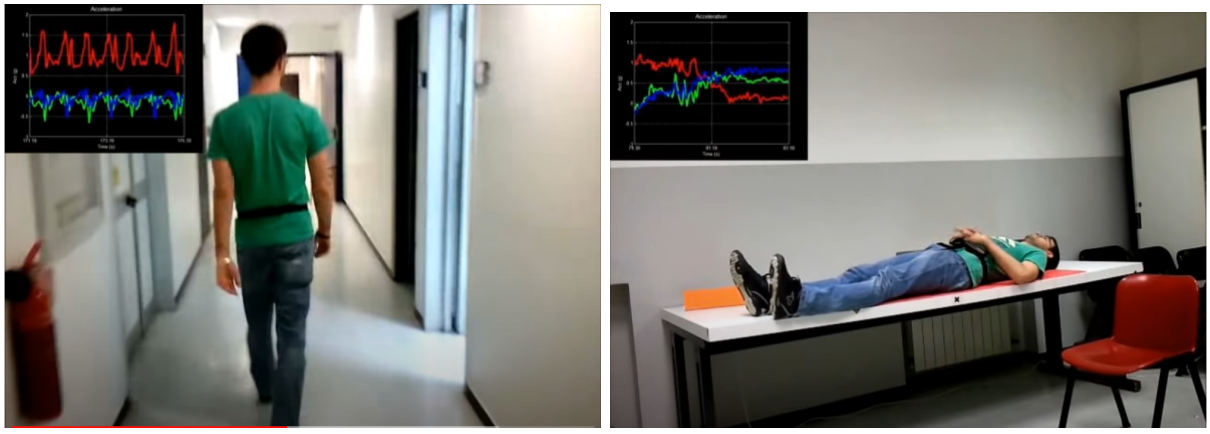

Figura 1.1: Experimentos llevados a cabo para conseguir los datos

Los datos han sido ya preprocesados por filtros de ruido y muestreadas en conjuntos de 2.56 segundos con un 50 % de superposición entre muestras adyacentes. En total, cada muestra tiene 128 lecturas que son las que se utilizan para extraer el vector de características.

El vector de características final se ha creado a base de diferentes datos correspondientes a esas 128 lecturas tomando datos como la media y la desviación típica de la aceleración lineal de cada uno de los ejes, los valores máximos y mínimos de las aceleraciones, etc, aplicadas sobre los datos de la ventana de tiempo de la muestra. En total, cada muestra tiene un vector de 561 componentes de tipo real y unas etiquetas que abarcan desde el valor 1 al 6, siguiendo el orden de las actividades anteriormente mencionado.

Los experimentos fueron grabados en vídeo[3] para su posterior etiquetado a mano.

Se nos proporciona 10299 instancias repartidas en un conjunto de train (70 % del total) y un conjunto de test (30 % del total). Aunque esto es aproximado, ya que estos conjuntos train y test, separan a los 30 voluntarios en 21 para train y 9 para test y cada individuo tiene un número de muestras distinto. Sin embargo, hemos optado por juntar los datos y los hemos separado nosotros.

1.1 Separación de Datos en conjuntos Train y Test

Nuestra base de datos tiene una particularidad muy importante: dado que las muestras son tomadas en ventanas de tiempo y que estas ventanas se solapan hasta en un 50 %. Eso quiere decir que los datos están altamente correlados entre sí y que no podemos separarlos de forma aleatoria, pues, si una muestra está en el conjunto de train y la muestra adyacente en el de test, habremos contaminado el conjunto de test (data snooping). Para solucionar esto, llevaremos a cabo el procedimiento aconsejado por los autores: separar los datos por individuos y no por muestras. Es decir, podemos separar el 30 % de los individuos, 9, como test y dejar a los 21 restantes como train.

Pero para que podamos considerar que la separación por individuos sea acertada, debemos asegurarnos que los números de muestras de cada individuo sean similares, o no se mantendrían los porcentajes deseados. Para ello, hemos calculado el número de muestras de cada individuo y lo hemos mostrado en este gráfico:

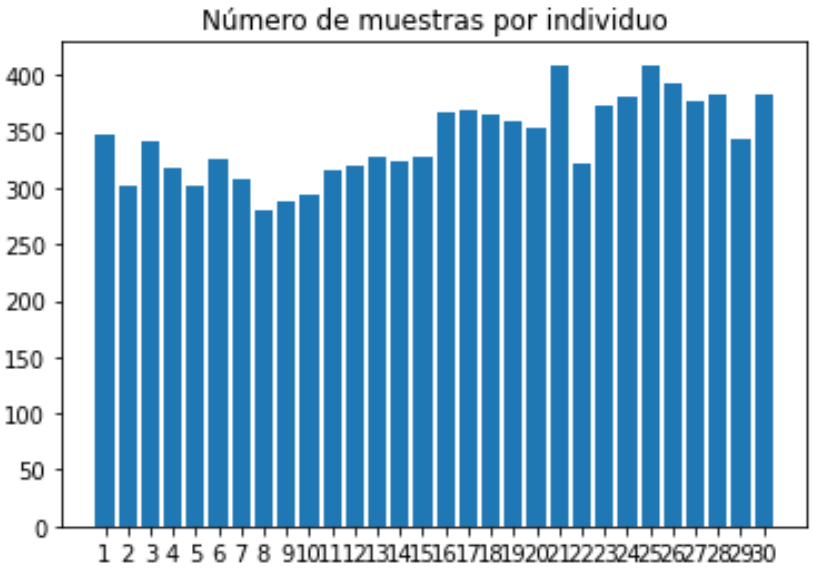

Figura 1.2: Distribución del número de muestras por individuos

Dado a que en general no hay demasiada variación entre la cantidad de datos que tenemos de cada persona, podemos proceder a separar los datos en train y test sin que nos queden en una proporción no deseada.

Primero, hemos unido los conjuntos de train y test separados por defecto en la base de datos y procederemos a separarlos nosotros. Al realizar la separación dejando al 70 % de las personas en train (21) y al resto en test (9) hemos obtenido 8032 datos en train y 2267 datos en test siendo esto un 77 % de los datos en train y un 23 % en test.

Esta varianza queda dentro del rango correcto de porcentaje de separación, pues se suele considerar correcto separar entre un 20 % y un 30 % de los datos. Por esto, consideraremos esta separación como adecuada.

Una vez dispuestos de los datos de train y test, se han de barajar de forma aleatoria.

Esta correlación entre muestras deberemos tenerla en cuenta también cuando realicemos cross validation, y hablaremos de ella en su apartado correspondiente.

Nota: todas las visualizaciones de los datos o estadísticas de ellos que se muestran a partir de ahora son exclusivamente del conjunto de entrenamiento. El conjunto de test no se ha tocado en ningún momento.

PARTE 2

Codificación datos de entrada

Nuestros datos están formados por variables reales del rango [0, 1]. Tener sólo variables reales nos ahorra lidiar con diferentes clases de atributos y los problemas relacionados con los datos categóricos. Además, no hay datos perdidos, por lo que no será necesario tomar medidas a ese respecto.

Una variable importante es el grado de desbalanceo que tengan los datos, es decir, si el número de muestras de cada clase es similar. Para comprobar esto se ha realizado este gráfico con el número de muestras de cada clase del conjunto de entrenamiento:

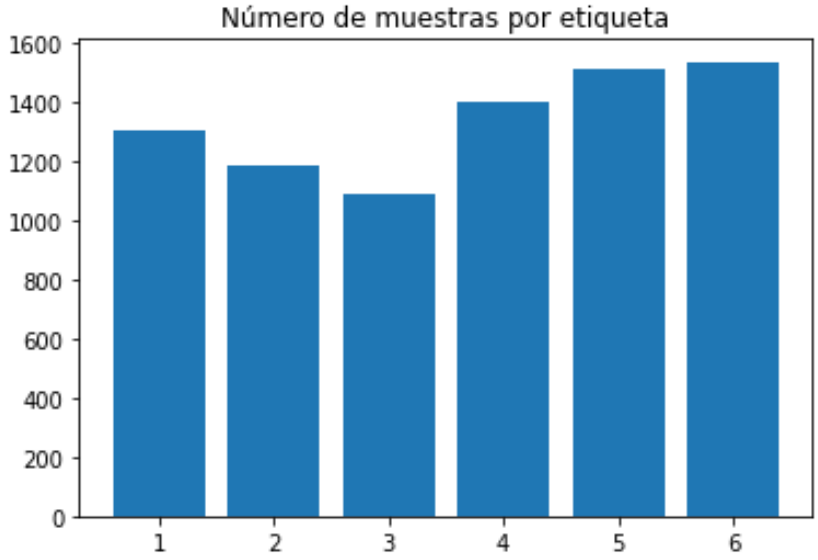

Figura 2.1: Distribución del número de muestras por clase

Podemos ver que existe un ligero desbalanceo, siendo la clase 3 la más escasa y la 6 la más numerosa. No es un caso de desbalanceo excesivo (como podría suceder en, por

ejemplo, un problema de segmentación de imágenes), pero lo tendremos en cuenta a la hora de tomar ciertas decisiones.

Más concretamente, para mitigar el efecto de este desbalanceo, hemos considerado que la mejor opción era elegir una función de pérdida que contase la precisión de forma balanceada entre todas las clases *'balanced_accuracy'*, como discutiremos en el apartado 5.

También, hemos generados gráficos que muestran ciertas informaciones sobre los atributos de los datos: concretamente los valores medios, mínimos, máximos y la desviación típica.

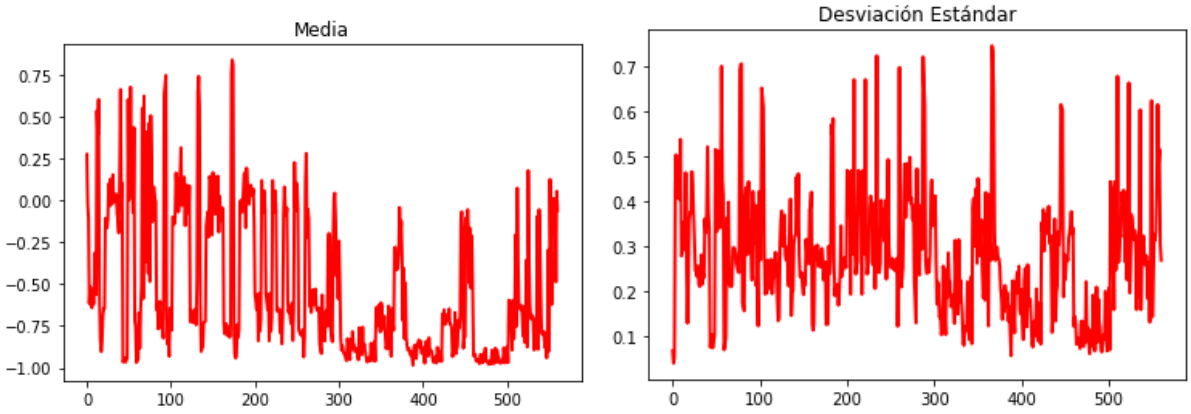

Figura 2.2: Media y desviación típica de los atributos

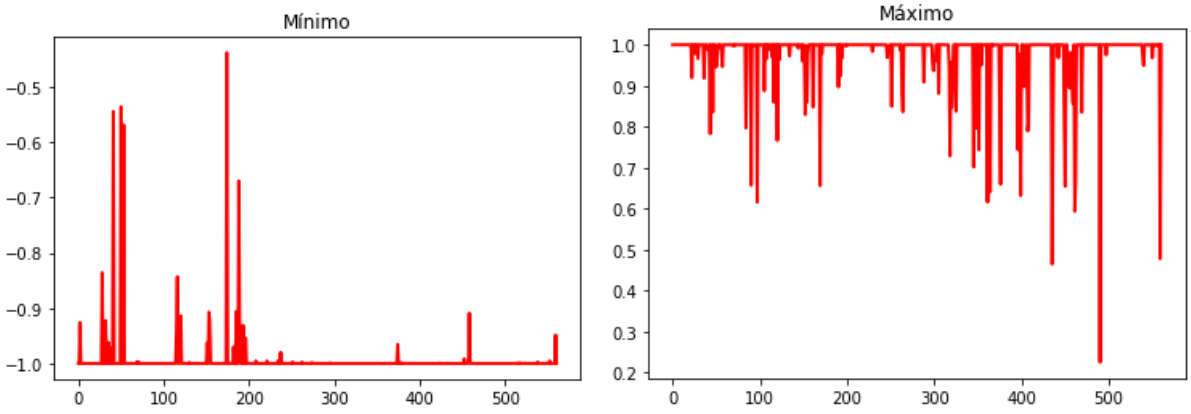

Figura 2.3: Mínimo y máximos de los atributos

En estos gráficos podemos ver cómo muchos mínimos están agrupados en -1 y los máximos están agrupados en 1, por lo que podemos entender que los sensores no tenían capacidad para detectar valores mayores o menores. Esta es una limitación física de la forma de extraer los datos.

También vemos que, aunque los datos están acotados entre -1 y 1, no están estandarizados, es decir, la media no es 0 y la desviación típica no es 1. Por ello, necesitarán un proceso de normalización que acometeremos más adelante.

2.1 Eliminación de outliers

Queremos reducir la escala de los datos, pero se nos presentan 2 alternativas, usar un método como **StandardScaler** (estandarización) pero eliminando posibles outliers, dado que está técnica se ve afectada por estos puntos, o usar una técnica que no se vea muy afectada por outliers como lo puede ser **RobustScaler** (similar a la normalización, pero en su lugar usa el rango intercuartil, y no se ve tan afectado un número reducido de outliers muy grandes). En nuestro caso, hemos escogido la primera opción por conocimiento de la técnica. Más adelante, en la sección de normalización se tratará con más detalle.

Dado que usaremos **StandardScaler** debemos de asegurarnos que no haya outliers o que haya la menor cantidad posible, sin que se pierda información relevante. Para ello hemos usado **LocalOutlierFactor**, que haciendo uso del algoritmo k-nearest neighbors, calcula la desviación de densidad local de un punto de los datos determinado con respecto a sus vecinos. Considera como valores atípicos las muestras que tienen una densidad sustancialmente menor que sus vecinos. Un punto será un outlier si está más lejos de sus k vecinos de lo que lo están en promedio sus k vecinos de sus correspondientes k vecinos.

El valor k de vecinos considerados normalmente se establece:

- Mayor que el mínimo número de muestras que un clúster tiene que contener, de modo que otras muestras pueden ser valores atípicos locales en relación con este clúster.
- Menor que el número máximo de muestras cercanas que pueden ser potencialmente valores atípicos locales.

Como en la practica no disponemos de esa información, hemos utilizado n_neighbors=20 (el valor k) que es un valor recomendado por Scikit-Learn porque suele dar buenos resultados.

"In practice, such informations are generally not available, and taking n_neighbors=20 appears to work well in general."[5]

Además la función nos permite elegir que distancia usará para calcular a los vecinos (euclídea, manhattan...), nosotros usamos la distancia manhattan pues en distintas fuentes[6] hemos encontrado estudios que avalan que en problemas de alta dimensionalidad funciona mejor que la euclídea.

Al utilizar esta técnica con nuestros datos, hemos eliminado sólo 15 outliers, el **0.19** %. Con lo cual no se ha hecho un gran cambio y nos aseguramos de que esos valores más dispersos no afecten a la normalización

PARTE 3

Selección de subconjunto de atributos

En nuestro problema, tenemos 561 atributos referentes a valores obtenidos de los sensores sobre una ventana de tiempo. Podemos especular que muchos de estos atributos tendrán una alta correlación entre sí: por ejemplo, es probable las aceleraciones medias sean muy similares entre sí. Esto nos da la idea de que sería buena idea reducir la dimensionalidad de los datos, pues con menos dimensiones se podría almacenar la misma información.

Reducir la dimensionalidad nos reporta los siguientes beneficios:

- Reduce la dimensión Vapnik–Chervonenkis de los modelos lineales (**recordamos que en los modelos lineales,** $d_{VC} = d + 1$). Por lo que es más probable que generalice bien y sobreajuste menos.
- El entrenamiento será más rápido de llevar a cabo, pues hay menos parámetros que ajustar.

Pero por supuesto, tenemos que tener en cuenta que si reducimos demasiado la dimensionalidad, perderemos información relevante para el problema.

Para esta tarea, hemos utilizado el análisis de componentes principales o **PCA** por sus siglas en inglés. Esta es una técnica que busca la mejor proyección de los datos en términos de mínimos cuadrados en un espacio con dimensión menor[4].

Sklearn tiene una implementación de PCA y será la que usaremos. Este algoritmo permite seleccionar la dimensión objetivo o buscar la dimensión que mantenga un porcentaje de la varianza de los datos. Sin embargo, nosotros hemos optado por calcular el valor óptimo de la dimensión objetivo.

Para esto, se ha realizado un experimento, ajustando un modelo sencillo (regresión logística sin regularización) utilizando cross-validation a los datos después de reducirles la dimensionalidad a distintos valores. Con esto, se ha generado el siguiente gráfico en

donde se puede ver como varían los resultados según la dimensión (desde la dimensión 5 a la 561).

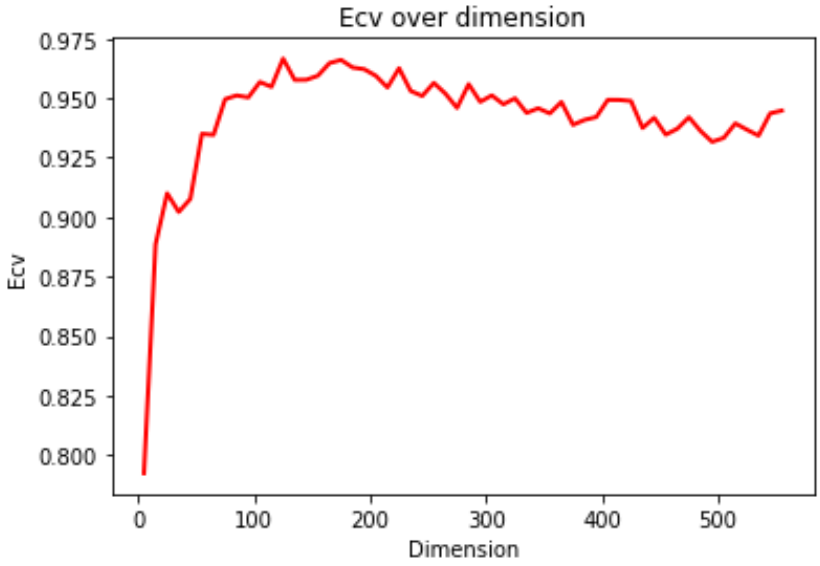

Figura 3.1: Estudio de la dimensionalidad óptima

Se puede observar que la reducción de dimensionalidad nos trae mejoras en el error de validación. De hecho, los mejores resultados se encuentran entre las dimensiones 90 y 250, estando por encima de 92.5 %. Si reducimos más, la accuracy comienza a decrecer hasta por debajo de 80 % si reducimos hasta dimensión a 5. Este experimento se ha llevado a cabo sólo con un modelo, pero tomamos como cierto que la gráfica es generalizable a otros modelos que afronten el mismo problema.

Vemos que podríamos optar por un valor de 40 y la accuracy seguiría por encima de 90. Sin embargo, para no arriesgarnos a perder demasiada información, optaremos por un valor más conservador: 160, que está dentro del intervalo con mejores resultados. Nuestros modelos se entrenarán con muestras de **160 dimensiones**. Así conservamos aproximadamente un 99 % de la varianza original pero con un porcentaje de los atributos mucho menor. Sin embargo, como no hay razones teóricas que confirmen que el modelo reducido generalizará mejor, probaremos también todos los modelos con los datos con la dimensionalidad original.

Tenemos que notar que el ajuste del **PCA** se lleva a cabo sólo con los datos de entrenamiento y luego ese modelo se usa para transformar tanto los de train como los de test. De otro modo, si ajustáramos también con los datos de test estaríamos cayendo en *data snooping*.

PARTE 4

Normalización

Los modelos de aprendizaje suelen funcionar mejor si los vectores de características están normalizados[8]. El objetivo de la normalización es cambiar los valores de las columnas numéricas en el conjunto de datos a una escala común, sin distorsionar las diferencias en los rangos de valores. Es muy necesario cuando las características tienen diferentes rangos entre sí, para que los algoritmos funcionen mejor. Para lograrlo, utilizaremos el método de normalización más usado[7]: la estandarización, es decir, conseguir una distribución con media 0 y desviación típica 1. Para ello, restamos a cada característica la media y la dividimos por la desviación típica.

En este paso hay que tener cuidado, pues es muy fácil caer en Data Snooping si utilizamos todos los datos para normalizarlos. Para evitar esto, calcularemos la media y la desviación típica de **únicamente los datos de entrenamiento** (después del proceso de eliminación de outliers y la reducción de dimensionalidad) y utilizaremos estos factores tanto para normalizar los datos de entrenamiento como los de test.

Si ahora observamos las estadísticas de nuestros datos podemos ver cómo han cambiado:

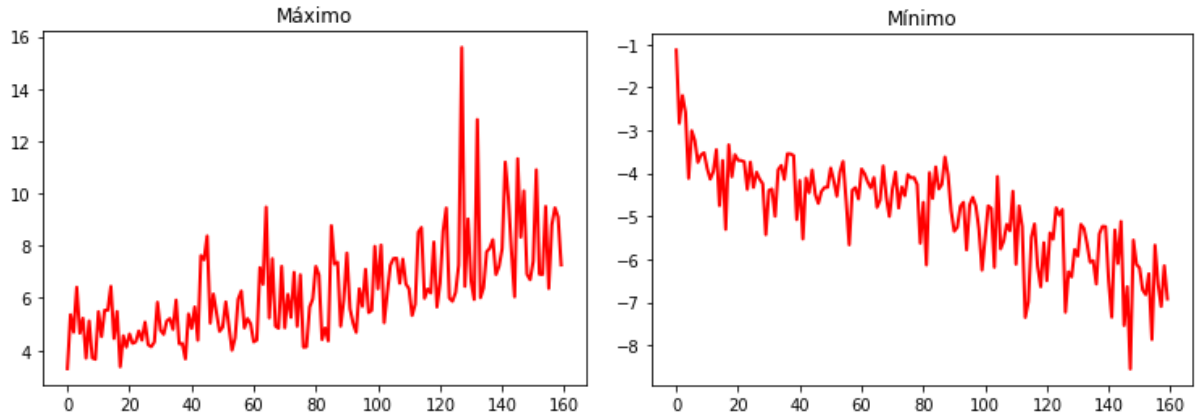

Figura 4.1: Media y desviación típica de los atributos

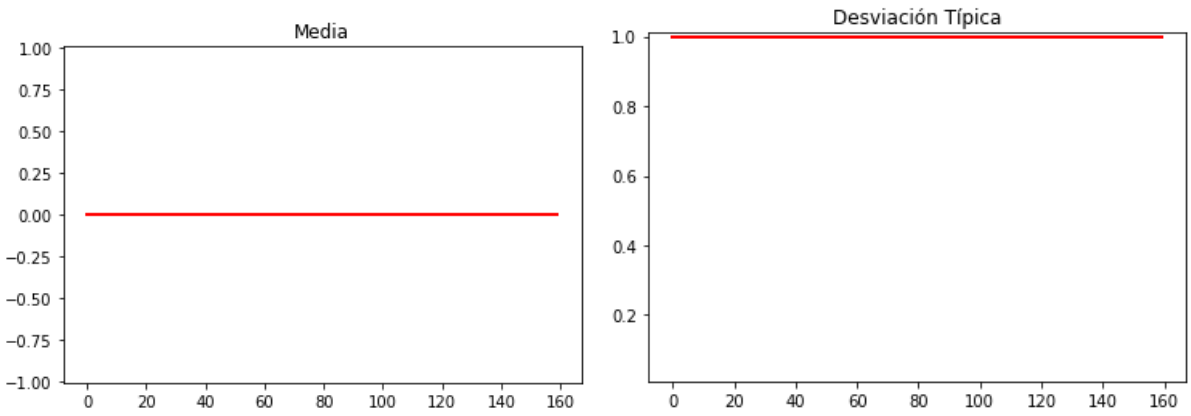

Figura 4.2: Mínimo y máximos de los atributos

Ahora la media es 0 y la desviación típica es 1, por lo que estamos ante unos datos estandarizados.

PARTE 5

Función de pérdida

Como estamos en un problema de clasificación, descartamos las funciones de pérdida de regresión (como el error cuadrático medio) y nos centraremos en las funciones de pérdida propias de clasificación. La función de pérdida más habitual en clasificación es **accuracy**, es decir, la proporción de muestras bien clasificadas. Es una métrica muy fácil de entender y que funciona con problemas multiclase como el nuestro. Además, su valor puede interpretarse como el porcentaje de acierto, siendo muy fácil de interpretar por humanos.

Sin embargo, hemos decidido utilizar una versión distinta de la accuracy, la balanced_accuracy, dado que en problemas donde no hay el mismo número de datos por clase puede dar lugar a malos resultados. Un caso extremo sería tener 100 datos, 97 de la clase 1 y 3 de la clase 2. Si nuestra hipótesis final hubiese aprendido a solo clasificar bien a la clase 1, la métrica accuracy nos daría un valor de 0.97 que aparentemente es bueno, pero para todo dato que pertenezca a la clase 2, lo clasificará mal. En cambio si usamos balanced_accuracy, que realiza el accuracy por clases y hace la media de estos, el resultado sería $\frac{0{,}97(clase1)+0{,}00(clase2)}{2} = 0{,}485$.

Como comprobamos que nuestros datos están ligeramente desbalanceados, hemos optado por utilizar balanced_accuracy.

PARTE 6

Criterio de elección de los modelos utilizados

Para este proyecto debemos probar como mínimo un modelo lineal y 2 modelos no lineales. Hemos escogido 3 modelos además de uno lineal. El conjunto seleccionado ha sido: **Regresión Logística**, **SVM**, **Perceptron Multicapa** y **Random Forest**.

Probar modelos lineales como primera opción es una buena idea, pues, cuanto más compleja sea la clase de funciones utilizada, mayor será su dimensión Vapnik–Chervonenkis y, por tanto, es menos probable que el modelo generalice bien fuera de la muestra. Los modelos lineales tienen una dimensión VC de *dimension + 1*, por ello, comenzar con modelos lineales es un buen punto de partida antes de probar modelos más complejos.

6.1 Modelo Lineal - Regresión Logística

Al tratarse de un problema de clasificación disponemos de dos posibles modelos lineales como principales candidatos: **Regresión Logística**, **Perceptron**.

- **Perceptron:** El perceptron es un modelo que necesita que los datos sean linealmente separables. Es muy improbable que nuestros datos lo sean, por que lo que hemos decidido descartarlo frente a una opción más robusta.
- **Regresión logística:** Es un modelo basado en la regresión lineal que utiliza una función δ (sigmoide, tanh, o similar) que aproxima la salida a una función de salto de forma suave y derivable. Es un modelo que hemos utilizado en prácticas anteriores y que hemos comprobado empíricamente que da bueno resultados en problemas similares a este.

Por estas razones, utilizaremos **regresión logística**. Como la regresión logística es un modelo de clasificación binaria, es necesario modificarlo para poder usarlo con múltiples clases. Para ello, se utilizará la estrategia **one against all**. Esta consiste en generar un conjunto de **w** por cada clase que haya y entrenarlo de forma separada para cada clase. Es decir, cada conjunto de w_i se entrenará tomando las muestras con la etiqueta i como casos positivos (1) y al resto como casos negativos (-1). Así, tendremos un conjunto de w_i capaces de separar una clase de las demás. Una vez hecho esto, podemos escoger el valor más alto de todos los $\delta(w_i^T x)$ y seleccionarlo como etiqueta predicha.

6.2 SVM

Support Vector Machines (SVMs) son un conjunto de métodos de aprendizaje supervisado utilizado para clasificación y regresión.

Las ventajas que nos ofrece SVM son:

- Eficaz en espacios de alta dimensión.
- Sigue siendo eficaz en los casos en que el número de dimensiones es mayor que el número de datos.
- Sólo utiliza un subconjunto de puntos de entrenamiento en la función de decisión (llamados support vectors) por lo que es eficiente en memoria también.
- Versatilidad, permitiendo especificar distintas funciones Kernel.

En cambio la principal desventaja es:

- Si el número de características es mucho mayor que el número de datos, para evitar el sobreajuste es crucial un buen uso de las funciones Kernel y de la regularización.

Hemos escogido SVM porque, en nuestro caso, no sufrimos de dicha desventaja ya que tenemos bastantes más datos y de muy alta dimensionalidad, por lo que comprendemos que SVM dará buenos resultados.

Nosotros usaremos la función SVC de Scikit-Learn, que implementa tanto "one-versus-one" como "one-vs-all" en el caso de múltiples clases. En nuestro caso usaremos "one-vs-all" como venimos haciendo hasta ahora.

SVC permite usar la regularización L2 a la cual, podremos ajustar la intensidad a través de un parámetro. Utilizaremos el kernel polinómico, porque nos permite utilizar el truco del kernel para tratar con transformaciones de la dimensionalidad que sean muy altas o casi infinitas y además, probaremos a usar distintos grados en el polinomio para darle distinto grado complejidad al modelo.

6.3 Perceptron Multicapa

Los perceptrones multicapa o redes neuronales son un modelo que ha tenido un crecimiento inmenso en la última década debido a su potencia. Esta complejidad, sin embargo, provoca que sea un modelo que tiende a sobreajustar y es necesario aplicarle regularización.

Las principales ventajas del perceptron multicapa son:

- Es capaz de aprender datos que sean no linealmente separables.
- Es un modelo muy versátil, que cambiando su arquitectura alteramos de forma drástica su comportamiento.

Pero esta potencia tiene sus desventajas:

- Es un modelo sensible a la inicialización aleatoria de los pesos.
- La elección de la arquitectura y el número de neuronas es un proceso manual (hay que considerarlos como más hiperparámetros).
- Es un modelo sensible a la escala de los atributos (nosotros los hemos estandarizado, por lo que superamos esta desventaja).
- Sufre de mucho sobreajuste.

Utilizaremos una arquitectura de 3 capas (dos capas ocultas y la capa de salida). La salida vendrá dada por una activación *softmax* que devolverá las probabilidades de que la muestra pertenezca a cada clase. La predicción será aquella clase con más probabilidad.

Aunque nuestro problema parece una tarea sencilla, hemos escogido las redes neuronales porque es un modelo capaz de ajustar datos no linealmente separables (como nuestro caso) y su principal desventaja, el sobreajuste, lo podemos mitigar con el uso de regularización.

6.4 Random Forest

Random Forest es un modelo que utiliza un conjunto de árboles de decisión entrenados con distintos subconjuntos de los datos. Las principales ventajas del uso de Random Forest son:

- Puede ajustar problemas no linealmente separables.
- Es un modelo muy robusto al sobreajuste, debido al consenso entre árboles (es improbable que todos los árboles estén sesgados en el mismo sentido).
- Funciona bien en casos con alta dimensionalidad.

Sin embargo, tenemos que tener en cuenta sus desventajas:

- Es sensible a datos extremos y pequeños cambios en los datos.
- Es computacionalmente costoso de entrenar.

En nuestro caso, tenemos datos con una muy alta dimensionalidad que no son linealmente separables, por lo que hemos escogido Random Forest porque ha tenido buenos resultados en problemas similares. Su principal ventaja, su robustez frente al sobreajuste será muy interesante de comparar frente a los perceptrones multicapa, que sufren de mucho sobreajuste.

PARTE 7

Regularización

La regularización[9] se encarga de priorizar los modelos más sencillos dentro de una clase de funciones determinada. La regularización añade una penalización a la función de pérdida según la complejidad del modelo, resultando en la siguiente fórmula:

$$E_{in}(w) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, x_i w^T) + \alpha R(w)$$

Esto hace que hipótesis más complejas sean penalizadas más que aquellas más simples (y por tanto, generales).

En esta práctica comprobaremos empíricamente si la regularización puede ayudar en nuestro problema y el tipo de regularización óptima. Se probarán: el modelo sin regularización, con regularización Lasso y con regularización Ridge. La diferencia entre estos dos tipos de regularización es el siguiente:

- **Lasso** (l1): Esta regularización penaliza la suma del valor absoluto de los coeficientes de regresión. Tiene la capacidad de forzar a que los coeficientes de las características tiendan a cero. Dado que una característica con coeficiente de regresión cero no influye en el modelo, Lasso consigue excluir los predictores menos relevantes. El grado de penalización está controlado por el hiperparámetro α. Cuando $\alpha = 0$ el resultado es equivalente a no aplicar regularización.
- **Ridge** (l2): su penalización es la suma **del cuadrado de los valores absolutos** de los pesos. Esta opción penaliza los valores muy grandes y busca que todos los pesos sean pequeños, pero sin que estos lleguen a 0. Al igual que Lasso, está controlado por el hiperparámetro α y cuando $\alpha = 0$ el resultado es equivalente no aplicar regularización. Es una regularización más útil cuando todas las entradas influyen en la decisión.

Estos dos tipos de regularización se probarán en regresión logística. En SVM y el perceptron multicapa se utilizará la regularización Ridge.

Además, en el perceptron multicapa utilizaremos *early stopping*, que es otra medida de regularización, pues separa una parte de los datos como validación y durante el entrenamiento, si comprueba que en varias iteraciones no se ha mejorado el error de validación, para el entrenamiento (así evita seguir sobreajustando).

Otros métodos de regularización posibles para las redes neuronales son la inclusión de capas de **BatchNormalization** y **Dropout** que dan muy buenos resultados. Sin embargo, se ha optado por no incluirlas por aumentar innecesariamente la complejidad de la arquitectura propuesta cuando la regularización L2 parece ser suficiente.

PARTE 8

Algoritmo de aprendizaje usado

8.1 SGD

Para el ajuste de Regresión Logística y SVM se utilizará **Stochastic Gradient Descent**. Este algoritmo da mejores resultados que el descenso del gradiente convencional debido a que actualiza los pesos por cada muestra recorriéndolas de forma aleatoria, por lo que evita fácilmente los mínimos locales.

En el perceptron multicapa, la implementación en sklearn utiliza minibatches, por lo que utiliza un pequeño conjunto de muestras por cada actualización de los pesos en vez de actualizar en cada muestra como SGD.

8.2 Cross Validation

Para validar todos los modelos que utilizaremos, usamos el método K-Fold Cross-Validation, que es un proceso iterativo para evaluar nuestro modelo de aprendizaje. Consiste en dividir los datos de forma aleatoria en k grupos de aproximadamente el mismo tamaño, k-1 grupos se emplean para entrenar el modelo y uno de los grupos se emplea como validación. Este proceso se repite k veces utilizando un grupo distinto como validación en cada iteración. El proceso genera k estimaciones del error cuyo promedio se emplea como estimación final. El número de iteraciones necesarias viene determinado por el valor k escogido. Por lo general, se recomienda un k entre 5 y 10.

La aplicación de una técnica de validación nos permite tener una medida de seguridad para no sobreajustar nuestro modelo.

Sin embargo, como ya hablamos en la sección 1.1 de separación de los datos en train y test, nuestro conjunto tiene un gran problema: que las muestras adyacentes están altamente correladas. Si separamos los folds de manera aleatoria habrá muestras en entrenamiento y validación que sean adyacentes y, por tanto, el conjunto de validación

estará contaminado. Debido a este problema, debemos separar los folds por individuos en vez de por muestras para asegurarnos de que el fold de validación nunca está contaminado.

Debido a esto, no hemos podido hacer uso de las funciones implementadas en sklearn como **GridSearchCV** o **cross_validation**, sino que hemos tenido que crear nuestra propia versión que separa por individuos en vez de muestras. La gran desventaja de este enfoque es que, al haber individuos con diferente número de muestras, los folds no tienen el mismo tamaño. Aún así, los tamaños son lo suficientemente similares como para hacer un buen uso de cross validation.

Usaremos esa función para obtener los mejores parámetros para cada modelo que utilicemos y nos quedaremos con aquel que produzca un menor error con todos los datos de validación. Usaremos un cv=5 ya que pensamos que con cv=10 el conjunto de validación queda demasiado reducido.

Nota: En un principio no caímos en este detalle y usamos GridSearch, el resultado fue que en los distintos modelos, el error de validación rondaba el 0.98-0.99. Eso nos hizo replantearnos si había algo que estuviésemos haciendo mal. Después caímos en que no estábamos separando por individuos en el GridSearchCV y que estábamos haciendo una selección aleatoria de valores que podían contener datos de una persona presente tanto en validación como en el entrenamiento, lo cual causaría una contaminación del conjunto de validación. Al corregir esto con la implementación de unas funciones propias, los errores de validación reflejan mejor la bondad del modelo porque el conjunto de validación no está contaminado.

8.3 Selección de parámetros

8.3.1 Regresión Logística

Para la regresión logística se utilizará la clase **SGDClassifier** de sklearn, que permite utilizar regresión logística. Además, esta clase se encarga de llevar a cabo *SGD* y *one against all* y permite un fácil entrenamiento con las funciones de grid search y cross validation.

Los parámetros importantes para **SGDClassifier** son los siguientes:

- **loss**: Este parámetro indica el tipo de función de perdida que estamos usando. Utilizaremos *'log'* para la función de pérdida de la regresión logística.
- **learning_rate**: determina si el learning rate es dinámico o constante. Entre las opciones, las relevantes son *'constant'*, que mantiene el valor de η igual a η_0 durante todo el entrenamiento y *'adaptive'*, que disminuye el η cada vez que el modelo no es capaz de mejorar los resultados en una iteración. En la práctica 1 ya se habló

de cómo lo ideal sería utilizar un η dinámico, por lo que esta será nuestra opción predilecta.

- **eta0**: valor de η inicial.
- **penalty**: tipo de regularización. Utilizaremos *'l1'* para regularización Lasso y *'l2'* para regularización Ridge.
- **alpha**: constante que multiplica el valor de la regularización en la función de pérdida. Para probar el modelo sin regularización, incluiremos un posible valor 0.
- **max_iter**: máximo de iteraciones antes de parar. En todas las ejecuciones utilizaremos 100000 iteraciones.
- **shuffle**: controla si los datos han de ser barajados en cada epoch. Para usar bien SGD, mantendremos este parámetro siempre a *'True'*.
- **fit_intercept**: controla si se añade una fila de 1 a los datos de entrenamiento que actúe como bias, al igual que en el caso anterior, lo tendremos a *'True'*.
- **n_jobs**: indica el número de núcleos de CPU que se pueden usar para aprovechar el paralelismo que permite la estrategia *one against all* en problemas multiclase. Para aprovechar al máximo la capacidad de cómputo, tendremos este valor al máximo posible del sistema (indicado con un -1).

Para escoger los parámetros óptimos se realizará pruebas con cada uno de los siguientes parámetros y valores:

- **eta0** = [0.1, 0.01, 0.001]
- **penalty** = ['l1', 'l2']
- **alpha** = [0, 0.001, 0.0001, 0.00001]

Estos valores han sido escogidos por ser una muestra de los valores más habituales para estos parámetros. Por supuesto, un rango más amplio de valores permitiría un mejor ajuste del modelo, pero debemos encontrar un equilibrio entre el número de valores a probar y el tiempo que disponemos para ejecutar las pruebas.

8.3.2 SVM

Para SVM hemos utilizado la clase **SVC** de sklearn. Los parámetros más relevantes son:

- **C**: Este parámetro sirve para ajustar la inversa de la intensidad de regularización es decir es $\mathbf{1}/\alpha$. Concretamente usa la regularización L2. Usaremos distintos valores durante Cross-Validation.

- **kernel**: Especifica el tipo de kernel que se va a utilizar en el algoritmo. Debe ser uno de los siguientes [**'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'**]. Hemos escogido el tipo polinomial ('poly') como ya se ha discutido.
- **degree**: Es un parámetro que sirve para establecer el grado del polynomial kernel ('poly'). Para los demás kernels no repercute. Usaremos distintos valores durante Cross-Validation.
- **gamma**: El coeficiente del Kernel para los tipos 'rbf', 'poly', 'signoid'. Usamos la opción más habitual, 'scale' **gamma=1 / (n_features * X.var())**.
- **coef0**: Es el valor del término independiente en la función kernel. Lo dejamos a 0.
- **shrinking**: Si se debe usar la heurística de reducción. Lo hemos dejado a True para reducir el tiempo de entrenamiento.
- **probability**: Si se habilitan las estimaciones de probabilidad. Estos se calculan utilizando un costoso five-fold cross-validation. En nuestro caso lo dejaremos a False para no incrementar el tiempo.
- **tol**: Es el valor de tolerancia (umbral) que, si se cruza, detiene las iteraciones del solver. Usaremos 1e-2 debido al elevado tiempo de entrenamiento.
- **cache_size**: Especifica el tamaño de la caché del núcleo en MB. Usaremos distintos valores durante Cross-Validation.
- **class_weight**: Usaremos el modo **"balanced"** que utiliza los valores de 'y' para ajustar automáticamente los pesos inversamente proporcionales a las frecuencias de clase en los datos de entrada como **n_samples / (n_classes * np.bincount(y))**.
- **max_iter**: Es el número de iteraciones límite del solver. Usamos el valor 100000 al igual que el resto de algoritmos.
- **decision_function_shape**: Utilizamos este parámetro para que la función de decisión siga la estrategia "one-vs-all".

Al igual que en el caso anterior, se realizará un grid search con los siguientes valores:

- **C**: [0.1, 1, 10, 100, 1000, 10000, 100000].
- **kernel**: ['poly'].
- **degree**:[1, 2, 3, 4, 5].
- **cache_size**: [200, 400].

Hemos usado valores del parámetro C lo suficientemente distanciados entre si para abarcar las posibilidades de una regularización intensa, intermedia y suave. Para el Kernel hemos escogido el kernel polinómico, nos permite utilizar el truco del kernel para tratar con transformaciones de la dimensionalidad que sean muy altas o casi infinitas. Permitiendo al polinómico usar desde el grado 1 al 5, ya que no se recomienda más grado por posibilidad de overfitting[10]. El parámetro cache lo modificamos porque Scikit-Learn nos dice que puede ser útil de cara a reducir tiempos pero nosotros no hemos notado demasiada mejoría.

8.3.3 Perceptron Multicapa

Para el perceptron multicapa utilizaremos la implementación de sklearn *MLPClassifier*. Esta implementación utiliza regularización L2 o Ridge. Los hiperparámetros más relevantes son los siguientes:

- **solver:** algoritmo de aprendizaje. Utilizaremos *'sgd'* para indicar Stochastic Gradient Descent.
- **hidden_layer_sizes:** Arquitectura y número de parámetros de la red. Se tratará como un parámetro más y se darán valores entre 50 y 100 para elegirse en la grid search.
- **activation:** función de activación de las capas ocultas. Se probarán las dos funciones más habituales: *'tanh'* y *'relu'*.
- **learning_rate:** igual que en RL, se utilizará *'adaptive'* por las razones ya comentadas.
- **learning_rate_init:** valor de η inicial.
- **early_stopping:** Al utilizarlo, separará un 10 % de los datos proporcionados y los utilizará como validación cada época. Si el error de validación no mejora en varias épocas seguidas, se termina el entrenamiento para evitar sobreajuste. Dado que es una forma de regularización extra, vamos a establecerla a *'True'*.
- **alpha:** intensidad de la regularización L2.
- **shuffle:** controla si los datos han de ser barajados en cada epoch. Para usar bien SGD, mantendremos este parámetro siempre a *'True'*.
- **max_iter:** máximo de iteraciones antes de parar. En todas las ejecuciones utilizaremos 100000 iteraciones.

Al igual que en los casos anteriores, se realizará un grid search con los siguientes valores:

- **activation** = ['tanh', 'relu']

- **hidden_layer_sizes:** [[50, 50], [100, 50], [100, 100]]
- **learning_rate_init** = [0.1, 0.01, 0.001]
- **alpha** = [0, 1, 0.1, 0.01, 0.001, 0.0001]

En la función de activación se han probado *'tanh'* y *'relu'*. La tangente hiperbólica se ha escogido por ser la función que se ha usado durante las practicas anteriores y ha dado buenos resultados. Relu se ha escogido por ser una de las funciones más usadas en las redes neuronales (por ejemplo, durante la asignatura de Visión por Computador se trabaja con ella).

8.3.4 Random Forest

Para el random forest hemos utilizado la clase **RandomForestClassifier** de sklearn. Sus principales parámetros son:

- **n_estimators:** El número de árboles a generar.
- **criterion:** el criterio a utilizar: *'entropy'* o *'gini'*. En teoría vimos que no hay mucha diferencia de resultados entre ellos, por lo que escogemos el más utilizado, *'gini'*.
- **max_depth:** La profundidad máxima de cada árbol. Probaremos distintos valores en el grid.
- **min_samples_split:** el número mínimo de muestras para separar. Dado que vamos a modificar *max_depth*, podemos dejar este parámetro a simplemente 2.
- **min_samples_leaf:** el número mínimo de muestras para un nodo hoja. Dejaremos este parámetro a 1 (con una muestra, ya es una hoja), pues este valor es más importante en regresión que en clasificación, como es nuestro caso.
- **max_features:** máximo número de atributos a considerar por cada split. En teoría vimos que un buen valor es la raiz cuadrada del número de atributos, por lo que escogeremos la opción *'sqrt'*.
- **min_impurity_decrease:** Valor mínimo de decrecimiento de la impureza requerida para crear un split. Utilizaremos un valor de 0, para permitir que se realicen todos los splits aunque reduzcan muy poco la impureza.
- **bootstrap:** La utilización o no de bootstrap. En nuestro modelo lo mantendremos siempre a *'True'* para que se utilicen subconjuntos distintos de los datos.
- **class_weight:** Parámetro que permite asignar pesos a las distintas clases. Al igual que en el caso de SVM, utilizaremos *'balanced'* por los mismos motivos.

Al igual que en los casos anteriores, se realizará un grid search con los siguientes valores:

- **n_estimators** = [10, 100, 500, 1000]
- **max_depth** [None, 25, 50]

El valor de *None* en la profundidad máxima representa que los árboles no tengan límite, por lo que se pueden expandir hasta nodos hojas. Hemos probado con distintos números de árboles, siendo conscientes de que a mayor número de árboles, mayor será el tiempo de entrenamiento.

PARTE 9

Experimentación y elección de la mejor hipótesis

Para escoger la mejor hipótesis, primero se elegirán los hiperparámetros indicados de cada modelo. Para ello, se realizará una **Grid Search** utilizando **5-fold Cross Validation**.

Se probarán 2 versiones de cada modelo, una con los datos originales (561 atributos) y otra versión utilizando una dimensionalidad reducida a 160 mediante PCA como se explicó en el apartado 3 (en ambos se normalizan los datos). Nuestra **mejor hipótesis será aquel modelo con mejor** E_{CV} de entre todos los modelos probados.

A continuación, se muestran los mejores parámetros para cada modelo (incluyendo las dos versiones, con dimensión 160 y 561):

Regresión Logística			
	eta0	penalty	alpha
RL - 160	0.001	l2	1e-05
RL - 561	0.001	l2	1e-05

SVM			
	C	degree	cache_size
SVM - 160	100000	5	400
SVM - 561	100000	5	400

Perceptron Multicapa				
	activation	hidden_layer_sizes	learning_rate_init	alpha
MLP - 160	relu	[100, 100]	0.001	0.0001
MLP - 561	relu	[100, 100]	0.001	0.0001

Random Forest		
	n_estimators	max_depth
RF - 160	1000	50
RF - 561	1000	50

Y a continuación, exponemos los resultados de Ecv, Ein y tiempo en segundos de entrenamiento de los modelos con sus parámetros óptimos:

Resultados			
	E_{cv}	E_{in}	Tiempo
RL - 160	0.9702	0.9966	2.26
RL - 561	0.9665	0.9973	1.75
SVM - 160	0.9689	0.9948	1.88
SVM - 561	0.9595	1.0	1.60
MLP - 160	0.9607	1.0	5.51
MLP - 561	0.9573	1.0	5.52
RF - 160	0.9057	1.0	25.07
RF - 561	0.9472	1.0	43.03

Cuadro 9.1: Resultados de los modelos

Por último, escogemos el modelo que mejor resultado ha dado en cross validation y lo entrenamos con todos los datos de entrenamiento. Así, aprovechará todos los datos que tenemos y podremos probar su desempeño utilizando los datos de test, que no habíamos tocado hasta ahora.

En nuestro caso, hemos visto que el modelo con mejor error ha sido la **Regresión Logística** con la dimensionalidad reducida.

Mejor Hipótesis		
	E_{test}	E_{in}
RL - 160	0.8763	0.9919

Cuadro 9.2: Resultado de la mejor hipótesis

PARTE 10

Valoración de los resultados

10.1 Selección de modelos

Lo primero que podemos concluir al ver los resultados de la tabla 9.1 es que nuestros datos conforman un problema bastante sencillo de aprendizaje. Todos los modelos son capaces de conseguir más de un 90 % de accuracy en validación. De hecho, los tres mejores modelos (RL, SVM y MLP) han tenido resultados de accuracy muy similares entre ellos.

También podemos ver que la reducción de la dimensionalidad ha resultado ser una decisión acertada. Casi todos los modelos han tenido mejores resultados en su versión reducida frente a la versión con todos los atributos.

El overfitting vemos que ha sido un factor muy importante en nuestro entrenamiento. Los modelos capaces de ajustar problemas no lineales han sobreajustado al máximo, llegando a un error $E_{in} = 1{,}0$. De hecho, es sorprendente ver que los parámetros de regularización han sido escogidos muy bajos, lo cual quiere decir que los modelos obtenían mejores resultados en validación con poca regularización.

La gran conclusión que podemos extraer de estos resultados es que, en este problema, la **navaja de Ockam** es clave. El modelo más sencillo, la regresión logística ha obtenido mejores resultados que otros modelos más complejos. La razón es que este modelo sencillo ha sido capaz de reducir mucho el error de varianza debido a su simplicidad, a costa de aumentar ligeramente el error de sesgo con respecto a los otros modelos. Nuestro problema requería una clase de funciones sencilla y al utilizar clases de funciones más complejas sólo provocábamos más sobreajuste.

10.2 Resultados finales

La mejor hipótesis ha resultado ser **regresión logística**. Podemos ver que sufría de un poco de overfitting, pues su error de entrenamiento y de cross validation era muy optimista con respecto a su error obtenido en test 9.2.

Todos los resultados finales los hemos expresado en función de la balanced_accuracy, pero para visualizar los resultados finales, también vamos a utilizar una **matriz de confusión**. Estas matrices son una forma simple de saber en qué clases falla más cada modelo: cada fila representa los valores de las etiquetas reales y las columnas los valores predichos. Cada casilla contendrá el número de veces que, siendo la etiqueta real la que indique la fila, se ha predicho el valor de la columna. Por lo tanto, en la diagonal se podrá observar aquellas etiquetas correctamente clasificadas.

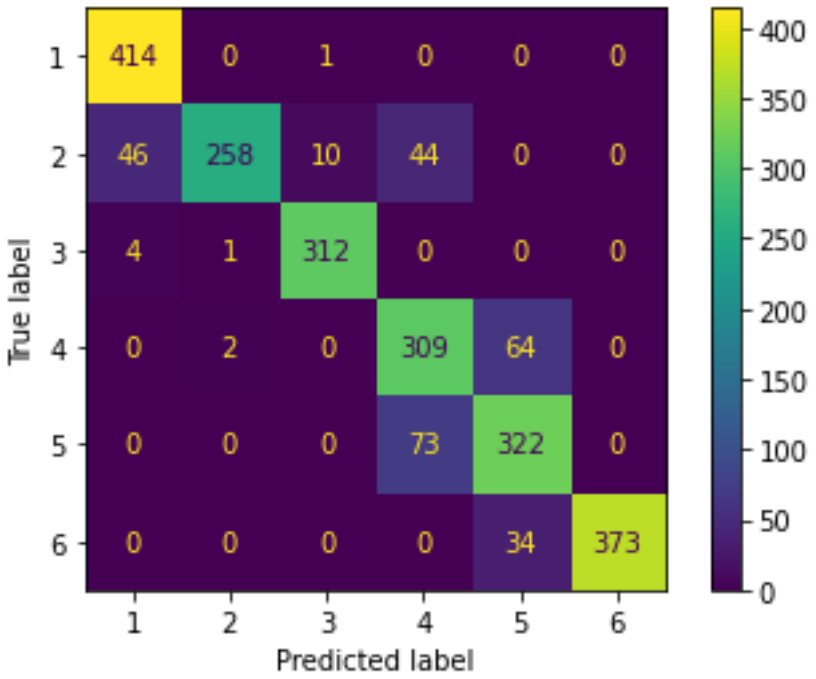

Figura 10.1: Matriz de confusión

Aquí podemos observar el resultado obtenido en la matriz de confusión. Recordemos que significaba cada clase:

- Clase 1 → **ANDAR**.
- Clase 2 → **SUBIR ESCALERAS**.
- Clase 3 → **BAJAR ESCALERAS**.
- Clase 4 → **SENTARSE**.

- Clase 5 → **ESTAR DE PIE**.
- Clase 6 → **TUMBARSE**.

Podemos ver que nuestro modelo es capaz de clasificar la mayoría de situaciones, pero se confunde con algunos casos que pueden ser similares.

- Podemos ver que el modelo es capaz de distinguir los casos de Andar (1) y Bajar escaleras(3) sin casi ningún error. Podemos entender que estas actividades tienen un patrón muy definido.
- Sin embargo, podemos ver cómo el hecho de Subir Escaleras (2) no es tan claro como el de bajar, pues es muchas veces confundido con el hecho de andar o sentarse. Podemos considerar que la razón de esta confusión es que cuando se sube escaleras también se está andando.
- Las actividades de Sentarse (4) y Estar de pie (5) son confundidas con frecuencia. Este es un caso claro de que tal vez la información proporcionada por los sensores no es suficiente para distinguir una clase de otra, pues la aceleración en ambos casos sería cercana a 0 y la posición captada por el giroscopio tampoco variaría mucho, pues la cintura de los voluntarios estaría más o menos en la misma posición.
- Por último, la acción de tumbarse se clasifica casi siempre correctamente, pero hay veces que se confunde con estar de pie. Podemos barajar como hipótesis de esta confusión de nuevo que la aceleración en ambas es 0.

Tras ver estos resultados, consideramos que el ajuste ha sido un éxito, y que se ha logrado el objetivo de obtener un modelo que soluciona correctamente este problema.

PARTE 11

Error de generalización

Recordemos que el objetivo de cualquier problema de aprendizaje es conseguir minimizar el error fuera de la muestra o E_{out}, es decir, el error que el modelo producirá al utilizarse con datos nuevos, no vistos durante el entrenamiento. Por supuesto, el error E_{out} no puede ser medido directamente y debe ser aproximado.

Una opción es calcular la cota a partir del error de entrenamiento y la dimensión Vapnik-Chervonenkis del modelo utilizado. Sin embargo, esta cota tendrá mucho margen y sólo será posible calcularla en aquellos modelos que tengan una dimensión Vapnik-Chervonenkis finita.

$$E_{out}(h) \geq E_{in}(h) - \sqrt{\frac{8}{N} log(\frac{4((2N)^{d_{VC}} + 1)}{\delta})}$$

La segunda opción, es utilizar directamente el error de validación obtenido en cross validation. Debido a que los datos no han sido vistos en el entrenamiento, su cota sobre el error está ajustada sólo por el número de hipótesis que se han probado. Mediante el error de la hipótesis entrenada con los datos reducidos de cross validation podemos acotar su error fuera de la muestra, que acota a su vez al error fuera de la muestra de la hipótesis entrenada con todos los datos de entrenamiento:

$$E_{out}(g_{m*}) \geq E_{out}(g^-_{m*}) \geq E_{val}(g^-_{m*}) - \sqrt{\frac{1}{2N} log(\frac{2M}{\delta})}$$

Como vemos, el margen de la cota es mucho menor, pero todavía depende demasiado de todas las pruebas que se han realizado. En nuestro caso, al escoger la mejor hipótesis hemos calculado su error con un conjunto de test que no ha sido utilizado en ningún momento del proceso (se mantiene totalmente insesgado a diferencia de los anteriores). Debido a que es un conjunto que no ha influido en ningún momento para escoger la hipótesis podemos utilizar la siguiente cota:

$$E_{out}(h) \geq E_{test}(h) - \sqrt{\frac{1}{2N} log(\frac{2}{\delta})}$$

Al ser esta la cota más ajustada, es la que usaremos para aproximar E_{out} con nuestra mejor hipótesis. Tomando el valor de E_{test} y con un intervalo de confianza del 95 % $1 - \delta = 0{,}95; \delta = 0{,}05$

$$E_{out}(h) \geq 0{,}8763 - \sqrt{\frac{1}{2 \cdot 2267} log(\frac{2}{0{,}05})} = 0{,}8478$$

Existe una última opción que podríamos realizar si el modelo fuera destinado a producción: tomar todos los datos (train y test) y reentrenar nuestro modelo con todos. Podemos esperar que su error fuera de la muestra fuera menor o igual que el error fuera de la muestra calculado anteriormente, pero perderíamos la opción de conocer su error fuera de la muestra.

Durante todo el procedimiento de selección de la mejor hipótesis se ha utilizado cross validation teniendo cuidado de mantener los folds insesgados. Por esto, consideramos que los resultados obtenidos han seguido un criterio adecuado y que hemos obtenido la mejor solución de entre las planteadas. Además, debido a separar el conjunto de test, consideramos que el ajuste realizado puede generalizarse para casos fuera de la muestra.

Bibliografía

[1] D. Anguita, A. Ghio, L. Oneto, X. Parra, J. L. Reyes-Ortiz, *UCI Machine Learning Repository: Human Activity Recognition Using Smartphones Data Set*, Archive.ics.uci.edu, 2012. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones. [Accessed: 10- Jun- 2021]. 1

[2] D. Anguita, A. Ghio, L. Oneto, X. Parra and J. Reyes-Ortiz, *A Public Domain Dataset for Human Activity Recognition Using Smartphones*, European Symposium on Artificial Neural Networks, vol. 21, 2013. 1

[3] J. Reyes Ortiz, *Activity Recognition Experiment Using Smartphone Sensors*, Youtube.com, 2012. [Online]. Available: https://www.youtube.com/watch?v=XOEN9W05_4A. [Accessed: 10- Jun- 2021]. 1

[4] Na8, *Comprende Principal Component Analysis*, Aprendemachinelearning.com, 2018. [Online]. Available: https://www.aprendemachinelearning.com/comprende-principal-component-analysis/. [Accessed: 18- May- 2021]. 3

[5] *Outlier detection with Local Outlier Factor*, Scikit-learn.org. [Online]. Available: https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html?highlight=localoutlierfactor. [Accessed: 10- Jun- 2021]. 2.1

[6] C. Aggarwal, A. Hinneburg and D. Keim, *On the Surprising Behavior of Distance Metrics in High Dimensional Space*, 2002. Available: https://bib.dbvis.de/uploadedFiles/155.pdf. [Accessed 11 June 2021]. 2.1

[7] J. Hale, *Scale, Standardize, or Normalize with Scikit-Learn*, Medium, 2019. [Online]. Available: https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02. [Accessed: 20- May- 2021]. 4

[8] S. Lakshmanan, *How, When, and Why Should You Normalize / Standardize / Rescale Your Data?*, Towards AI — The Best of Tech, Science, and Engineering, 2019. [Online]. Available: https://towardsai.net/p/data-science/how-when-and-why-should-you-normalize-standardize-rescale-your-data-3f083def38ff. [Accessed: 18- May- 2021]. 4

[9] J. Rodrigo, *Selección de predictores, regularización ridge, lasso, elasticnet y reducción de dimensionalidad*, Cienciadedatos.net, 2016. [Online]. Available: https://www.cienciadedatos.net/documentos/31_seleccion_de_predictores_subset_selection_ridge_lasso_dimension_reduction. [Accessed: 18- May- 2021]. 7

[10] J. Rodrigo, *Máquinas de Vector Soporte (Support Vector Machines, SVMs)*, Cienciadedatos.net, 2017. [Online]. Available: https://www.cienciadedatos.net/documentos/34_maquinas_de_vector_soporte_-support_vector_machines. [Accessed: 12- Jun- 2021]. 8.3.2

Anexo: Instrucciones de ejecución

Para ejecutar nuestro código es necesario seguir los siguientes pasos:

1. Descomprimir la carpeta **Codigo** que está entregada junto a esta memoria.
2. Descargarse los datos de **Human Activity Recognition Using Smartphones** de esta página de UCI:

   https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

   Dentro de la página, seleccionar la opción **Data Folder** y la opción **UCI HAR Dataset.zip**.
3. Descomprimir los datos descargados en el directorio **Codigo/Datos/**.
4. Por último, ejecutar el archivo **Codigo/proyectofinal.py**

La estructura final de directorios debe ser esta:

```
- Codigo/
    - Datos/
        - test/
            - subject_test.txt
            - X_test.txt
            - y_test.txt
            - ...
        - train/
            - subject_test.txt
            - X_train.txt
            - y_train.txt
            - ...
    - proyectofinal.py
```

En el código, primero se realizará un análisis de los datos iniciales, imprimiendo algunas de las gráficas del primer y segundo apartado.

Después, se realizará el preprocesado descrito en la memoria y se mostrarán las gráficas del resultado.

Se entrenará la mejor hipótesis (descrita en la memoria) con los datos de train y se probará con los de test. Se mostrarán gráficas de los resultados.

El proceso de selección de hiperparámetros y selección de la mejor hipótesis está comentado, debido a que es un proceso que tarda mucho. Si se quiere realizar, se puede descomentar la llamada a la función **SelectBestModel**.

Si se quiere ejecutar alguno de los otros experimentos, basta con descomentar la llamada a su respectiva función en el main.