Práctica entregable de Análisis de datos y machine learning con R (caret)

Los trabajos tendrán que entregarse en **grupos de 3 alumnos**.

Fecha de entrega: 10 Diciembre 2024 23:55

Aprendizaje Computacional Curso 2024-2025

Version 1.0

1 Introducción

En la asignatura de *Aprendizaje Computacional* de Cuarto Curso del grado en Ingeniería Informática de la Universidad de Murcia nos dedicamos a estudiar diferentes técnicas de aprendizaje automático y sus posibles aplicaciones a diferentes problemas. En esta primera parte práctica de dicha asignatura vamos a estudiar una familia de estos problemas (clasificación) con el objetivo de crear un modelo de clasificación sobre un problema concreto. El modelo se obtendrá mediante aprendizaje supervisado. Se proporcionará una base de datos que tendrá que ser tratada adecuadamente para facilitar el aprendizaje del modelo. Se hará un pre-análisis de las características de las variables del problema. También se deberá realizar una comparación entre diferentes técnicas/modelos. Igualmente se deberá realizar un ajuste de los hiperparámetros de dichas técnicas/modelos. Finalmente se seleccionará un modelo final justificando dicha decisión a través de estimaciones de su rendimiento.

El planteamiento genérico de resolución de esta primera práctica por parte del alumno va a estar basado en la entrega de:

- Una **memoria** de aplicación de las técnicas vistas en clase al problema propuesto
- Un **script** con los comandos con que se analizaron y transformaron los datos, que generaron los modelos descritos en la memoria, y con los que se analizaron los resultados.
- Un **fichero** con la **copia del espacio de trabajo final** (o de los objetos necesarios para comprobar lo expuesto en la memoria).

2 Base de datos

La base de datos sobre la que se va a trabajar es la "Credit Approval" dataset cuya descripción está disponible en la dirección `https://archive.ics.uci.edu/dataset/27/credit+approval`. Los datos se encuentran en el fichero accesible en la dirección `https://archive.ics.uci.edu/static/public/27/credit+approval.zip`. La base de datos contiene 16 variables y 690 observaciones, siendo la última variable la clase (+ o -). De los 15 predictores hay 6 numéricos y 9 discretos.

MUY IMPORTANTE

En el directorio de recursos hay un fichero llamado `credit.trainIdx.rds` que contiene los índices de los datos a usar para entrenar (y el resto serán de test). Ejecuta los siguientes comandos para asegurarte que cargas correctamente el fichero de índices y separas la base de datos adecuadamente (asumiendo que la has cargado en una variable llamada `credit`).

```
# Hay que cargar en credit la base de datos descargada del UCI

# credit <- COMANDO DE CARGA DE DATOS

credit.trainIdx<-readRDS("credit.trainIdx.rds")
credit.Datos.Train<-credit[credit.trainIdx,]
credit.Datos.Test<-credit[-credit.trainIdx,]

nrow(credit.Datos.Train)
nrow(credit.Datos.Test)
```

```
> nrow(credit.Datos.Train)
[1] 553
> nrow(credit.Datos.Test)
[1] 137
```

Asegurate que tienes 553 observaciones en el conjunto de entrenamiento antes de empezar el preproceso de datos.

De manera orientativa comentar que se debe conseguir con relativa facilidad algún modelo que tenga un **85% de Accuracy** sobre el **conjunto de Test**.

3 Elementos a trabajar en la práctica

La práctica consiste en realizar, al menos, un ciclo del proceso de creación de un modelo de clasificación utilizando R y la librería caret. En la figura 1 se muestran las tareas a realizar en la práctica. También incluyen el mínimo de modelos/técnicas a comparar. Recuerda que **debes cumplir esos mínimos para superar la práctica**.

En la figura 2 aparece un esquema del proceso general de ajuste de un modelo, que debéis repetir 4 veces (una por cada modelo que se ha pedido) y comparar los resultados del modelo final.

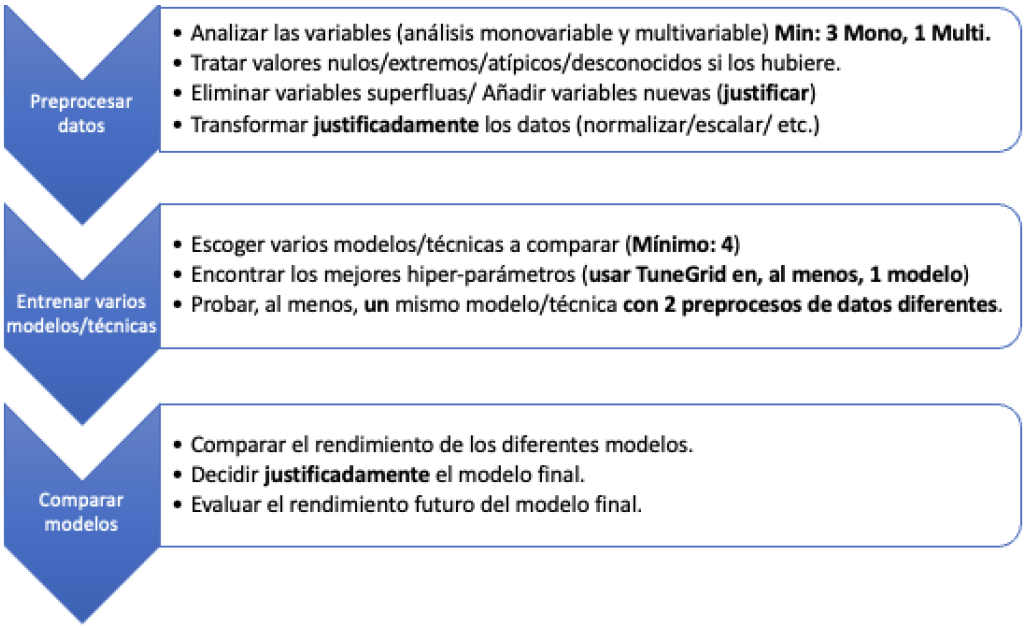

Figure 1: Tareas a realizar en la práctica entregrable uno.

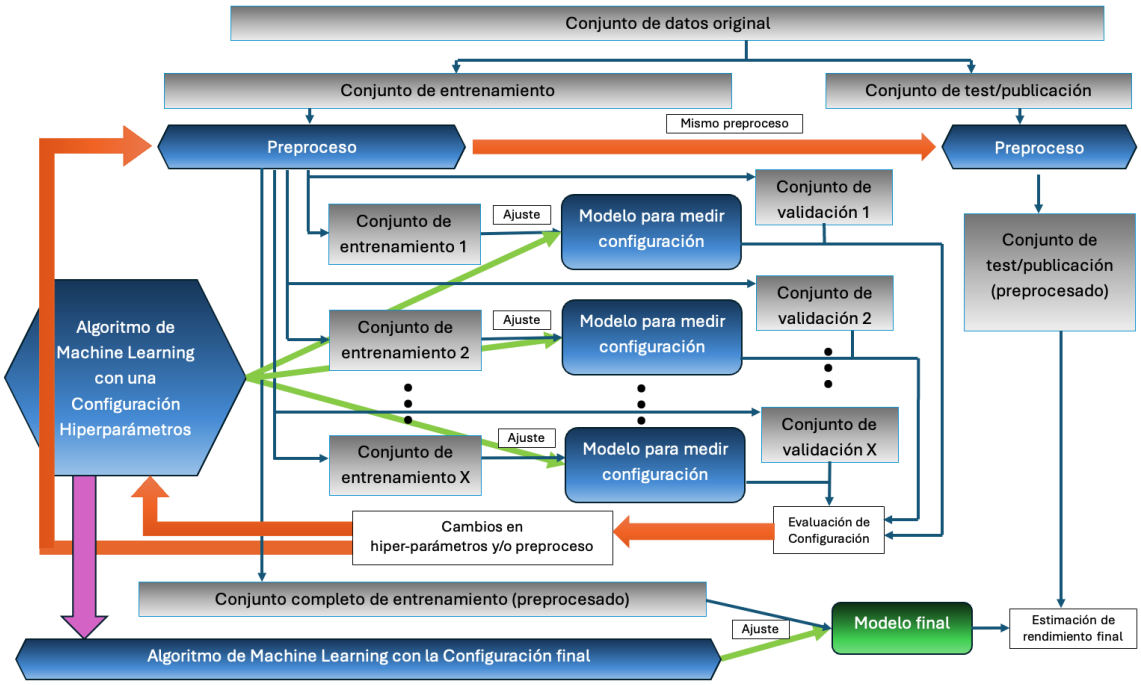

Figure 2: Proceso general de entrenamiento de un modelo.

4 Documentación y material a entregar

Como se ha mencionado en la sección 1 debes entregar 3 elementos. **La fecha de entrega es el 10 de Diciembre a las 23:55** mediante la tarea de aula virtual que se abrirá a tal efecto.

Si tuvieses problemas debido al tamaño de los ficheros de datos, **debes entregar la memoria y los scripts por el aula virtual** y los datos mediante la herramienta FileSender de la red Iris (a la que se tiene acceso con las credenciales de la universidad de murcia desde la dirección `https://filesender.rediris.es/`) al correo corporativo del profesor `jgmarin@um.es`. .

Al ser un **trabajo de grupo** solo debe enviar el trabajo **uno, y solamente uno** de los componentes del grupo y debe ser el **portavoz** (en la documentación deben ir, por supuesto, los nombres de los componentes del grupo). Repito **NO ENVIÉIS LOS TRABAJOS REPETIDOS**

IMPORTANTE

Los ficheros .rar pueden dar problemas de descompresión al enviarlos por el aula virtual, por tanto los ficheros a entregar **deben comprimirse con ZIP**.

Los TRES elementos a entregar son:

1. Una **memoria** de aplicación de las técnicas vistas en clase al problema propuesto.

2. Un fichero de **scripts** que incluya todos los comandos R con que se analizaron y transformaron los datos, que generaron los modelos descritos en la memoria, y con los que se analizaron los resultados.

Estos dos primeros elementos pueden entregarse de dos formas:
a) Con los scripts en uno/varios ficheros `R` y la memoria en un fichero `PDF`.
b) Con los scripts en uno/varios ficheros `Rmd` y la memoria en el `html` generado por ese `Rmd`.

3. Un **fichero** con la **copia del espacio de trabajo final** (o de los objetos necesarios para comprobar lo expuesto en la memoria). Se recomienda limpiar la memoria de trabajo, ejecutar los comandos exclusivamente necesarios para la práctica (bien los incluidos en el Rmd, o el script R según sea vuestro caso) y, solo luego, guardar la copia del espacio de trabajo final (para evitar incluir datos acumulados de pruebas u otros trabajos).

Es decir, la entrega debe incluir, como mínimo, esos 3 ficheros (o grupos de ficheros). También podéis incluir cualquier fichero adicional que necesitase vuestra práctica (p.e. el fichero `credit.trainIdx.rds` que se usa para que todos uséis los mismos datos de Train/Test, o el propio fichero de credit que descarguéis del UCI).

4.1 Memoria de aplicación de las técnicas vistas en prácticas

La memoria debe describir el proceso de generación del modelo, es decir, describir el tratamiento realizado a los datos, la experimentación realizada, y explicar las razones para escoger el modelo final utilizado. Es aconsejable comenzar con un primer apartado de introducción, seguido de secciones para cada uno de los elementos que aparecen en la figura 1, y un apartado final de conclusiones. También puedes incluir otro apartado aparte con tus impresiones personales sobre el trabajo y sugerencias, que son siempre bienvenidas.

Por supuesto las decisiones tomadas (p.e. variables eliminadas, transformaciones realizadas, etc.) deben justificarse. Recuerda que en la ventana donde aparecen los diagramas (plots) en Rstudio tienes la opción de exportar los diagramas a distintos formatos (si tienes un script Rmd puedes exportar a Html y te los incluirá).

Incluye en la documentación los diagramas que apoyen tus justificaciones para la toma de decisiones. **Vas a entregar un fichero PDF, o un fichero Rmd+html**, no una copia impresa, con lo que incluir bastantes diagramas (pero sin pasarse ni ser innecesariamente exhaustivo) y aumentar el número de páginas no afectará los bosques del mundo. Ni escatimes diagramas, **ni te pases**. **Sé ilustrativo/a sin caer en la repetición que no aporte nada**. Trata de sacarle partido a cada gráfico (mostrando comparativas en diferentes colores sobre el mismo diagrama). Hay ejemplos en sesiones de prácticas y en el tutorial proporcionado en recursos que te pueden servir de guía. `Lattice` te permite mostrar mucha información a la vez. El comando `melt()` te puede ayudar mucho para crear los datos para usar en `lattice`. **Una buena indicación de que un gráfico sobra es cuando en el texto de la práctica no se le referencia ni se comenta**. No, los gráficos no son auto-explicativos, es precisamente tu trabajo el explicarlos (porque yo ya sé interpretarlos, pero quiero saber si vosotros sabéis qué estáis mostrando).

En la figura 1 se indican el número mínimo de elementos a tratar para superar la práctica. Es importante fijarse que para la práctica cada subsección es "semi-independiente" de la anterior. Sería independiente en el sentido de que no es necesario realizar los pasos posteriores en todo lo analizado con anterioridad (algo que si podría ser necesario en un trabajo real). La primera sección sería una Análisis Exploratorio de Datos y un preproceso básico de datos. La segunda sería el entrenamiento propio de los potenciales modelos a usar. En este entrenamiento se puede usar, (o no, de ahí lo de "semi-independiente") el preproceso de datos realizado en la sección de preproceso/AED. En la tercera sección, comparar modelos, sí que se deben usar los modelos entrenados en la sección 2 para realizar la comparación (al fin y al cabo ya se tienen dichos modelos entrenados).

Es decir, que se debe:

SEC1) Explorar 3 variables con algún análisis monovariable y 1 variable con análisis multivariable.

SEC1) Hacer algún tratamiento (cualquiera de los vistos) a valores nulos/extremos/atípicos/desconocidos.

SEC1) Eliminar o añadir alguna variable (justificadamente)

SEC1) Hacer alguna transformación de alguna/s variable/s.

SEC2) Usar (entrenar con) al menos 4 modelos/técnicas diferentes de caret,

- Al menos **uno de dichos modelos** se debe entrenar explorando los hiper-parámetros mediante grid.

SEC2) Entrenar (y comparar), al menos, **un mismo modelo/técnica con dos maneras diferentes de preprocesar los datos** (p.e. eliminando tal variable en uno y dejándola en otro, o escalando tales variables en uno y no escalando en otros)

SEC3) Comparar el rendimiento de los diferentes modelos entrenados, al menos habrá 4:

1. Modelo/Técnica 1 (Simple o Complejo)
2. Modelo/Técnica 2 (Complejo)
3. Modelo/Técnica 3 (Complejo)
4. Modelo/Técnica 4 (Complejo)
   (a) con preprocesado 1
   (b) con preprocesado 2

SEC3) Decidir entre ellos cual sería el modelo que terminaríamos usando (justificarlo).

SEC3) Estimar cual sería el rendimiento de dicho modelo final sobre datos futuros.

Recuerda que algoritmo debe pertenecer a un paradigma de aprendizaje diferente. Paradigmas y algoritmos de ejemplo son un algoritmo de red neuronal MLP (Multi-layer perceptron), árboles de decisión (por ejemplo CART o random forest), ensamblajes (boosting, bagging, stacking, etc.), clasificadores lineales (e.g. linear discriminant analysis), algoritmos de la familia de naive bayes, etc.

Para distinguir entre "Simple" y "Complejo" podéis guiaros por la Sección 11 del tutorial `caretML.PDF`, pero hay más modelos "Simples" que no aparecen ahí, como el Naive Bayes, p.e.

Uno de los modelos complejos debe ser una red neuronal. Si se quiere se pueden usar redes profundas usando Keras/TensorFlow lo que es un plus, pero puede hacer más difícil encontrar un buen conjunto de hiperparámetros ya que habría que crear el código R para automatizar la búsqueda de hiperparámetros de forma análoga a la que proporciona caret, y así poder encontrar una estructura aceptable de la red neuronal y ser justos al comparar con otros modelos en los que se ha realizado dicha búsqueda de hiperparámetros.

Para una entrega de la práctica sin problemas se aconseja, antes de enviarla, **asegurarse de que compila de principio a fin**.

A la hora de la entrega:

- No olvides incluir una portada donde indiquéis el nombre del grupo y los nombres de **todos** los autores de la práctica.
- El documento final debe estar, bien en **formato PDF**, bien en el **HTML** que genere el (`.Rmd`), al estilo de los guiones de prácticas.
- Merece la pena que trabajes con el fichero R MarkDown (`.rmd`) para generar la memoria.

A veces se tiene la tentación, en las memorias, de detallar en exceso el trabajo realizado, en un intento de demostrar que se ha trabajado "mucho". En realidad el efecto de caer en esa tentación es más bien el contrario, empobrece el resultado y es evidencia de un trabajo superficial. En cambio una memoria concisa, a la vez que completa, precisamente es indicio de que se ha trabajado "mucho" y "bien". Así por ejemplo, si muestras un diagrama debes comentarlo y explicarlo, aunque sean unas pocas palabras. Mostrar luego otros diez gráficos iguales diciéndo que se interpretan igual que el anterior no aporta nada (y causa más bien, como os estoy diciéndo, una impresión pobre) y degrada la calidad del trabajo. Debéis entender que generar una gráfica es fácil y se obtiene con facilidad de un copy/paste o consulta a chatGPT, pero ser capaz de explicarla y justificarla (y sin que parezca que lo ha hecho chatGPT) es lo que demuestra competencia.

4.2 Scripts en R/Rmd

Como se acaba de mencionar, para una entrega de la práctica sin problemas se aconseja, antes de enviarla, asegurarse de que compila de principio a fin. Eso incluye asegurarse de que, si usas ficheros en subdirectorios, utilizas el comando `file.path()` para construir el `path` a los ficheros para conseguir que vuestro código sea independiente de la plataforma (Windows o Unix/Mac). P.e.:

```
path.a.mi.fichero <- file.path(".","dir","subdir","subsubdir","miFichero.ext")
misDatos<-read(path.a.mi.fichero,...)

# path.a.mi.fichero será ./dir/subdir/subsubdir/miFichero.ext en Unix/Mac
```

```
# path.a.mi.fichero será .\ir\subdir\subsubdir\miFichero.ext en Windows
```

Recuerda también, antes de esa prueba, tanto reiniciar la sesión de R (Session→Restart R) como limpiar la memoria de variables (con la escoba del panel superior derecho). De esa forma os aseguráis de que vuestro código inicializa por sí mismo todas las variables y carga las librerías necesarias. Si no limpiáis podríais acceder a variables viejas del historial de sesiones antiguas y, si no reinicializáis R, podrías no solicitar la carga de librerías previamente cargadas en sesiones anteriores con lo que vuestra entrega contendría errores cuando la cargue el profesor y os recuerdo que **no perderé el tiempo en arreglar errores de la entrega**.

4.2.1 Script con los comandos de R utilizados en la práctica y a través de los cuales se generaron los modelos y diagramas de la memoria.

Si no usas `Rmd` recuerda que debes incluir un script con todos los comandos utilizados para realizar la práctica. Dicho de otro modo, un fichero que, al ejecutarlo mediante el comando `source()`, replique la práctica.

Recuerda que RStudio tiene, en la ventana superior derecha, el historial de comandos. Desde allí te será fácil acceder a todos los comandos introducidos en la consola y pasarlos a un fichero script (en la ventana superior izquierda).

4.2.2 Si usas Rmd

Es muy aconsejable que hagas la práctica en Rmd (recuerda entregar también el `html` generado con tu `Rmd`) porque integra la creación de la documentación con el código usado en R. Aquí van algunos consejos si usas Rmd:

- Si el documento Markdown os resulta muy largo, se aconseja dividir el desarrollo en documentos Markdown más pequeños y una vez estén listos todos los análisis, se puede ensamblar todo en el documento final.
- Seguramente tendremos **chunks** de código llamadas a funciones que tardan demasiado y que resultan un fastidio si necesitamos ejecutarlas repetidas veces cuando estamos a medias de desarrollar algo. Se aconseja, para que la resolución de la práctica sea más ágil, los siguientes pasos:

  1. la variable que almacena el resultado de dicha función se ha de guardar en un fichero, con `saveRDS`
  2. el chunk de código se anula con `eval=FALSE` en el prólogo del chunk para que no se evalue más
  3. el siguiente chunk de código hace uso del resultado leyéndolo del fichero con `readRDS`.

  Os ahorraréis mucha repetición de experimentos si lo hacéis así.

4.3 Fichero de copia de la sesión de trabajo final

El último fichero a entregar es una copia de la sesión de trabajo final. Opcionalmente se podría aceptar un fichero con los objetos necesarios para comprobar los resultados de la práctica y lo expuesto en la memoria (incluiría modelos, conjuntos de entrenamiento/test, objetos necesarios para transformar los datos, diagramas utilizados, etc.), pero es mejor que entregues la sesión completa.

La forma más fácil de obtener este fichero lo más "limpio" posible es crear una sesión nueva, borrar el espacio de datos de todos los objetos (en Rstudio darle a borrar el "Environment", la escoba de

la ventana arriba derecha, pestaña "Environment") pues probablemente cargue datos de sesiones anteriores, cargar el script del apartado anterior (el entregable) mediane el comando `source()` (o ejecutar todos los chunks del .Rmd) y, por último, cuando termine de hacer todo el script, ejecutar el comando `save.image(file="SesionFinal.RData", compress = "xz")`.

El fichero con la sesión final debe llamarse "`SesionFinal.R`". Por defecto el comando `save.image()` comprime el fichero al guardarlo (en el comando anterior, además, se ha especificado que se utilice un método de compresión más lento pero habitualmente más eficaz que el método por defecto) así que no necesitarás hacerlo. Si el tamaño es demasiado grande para el aula virtual lo puedes enviar por FileSender de rediris (hasta 100 GB). Si usas FileSender indica 30 días como tiempo para ser accedido (por defecto son 10 días).

La idea es proporcionar una forma de tener la copia exacta de los modelos obtenidos en la práctica. Recuerda que, si no se ha tenido cuidado al establecer las semillas del generador de números pseudoaleatorios, al ejecutar el script para reproducir la práctica se obtendrán modelos y resultados diferentes. Dicho de otro modo, a pesar de seguir todos los comandos es posible que no se reproduzcan los experimentos (en especial si se han usado multicores y no se ha inicializado correctamente `seeds`). Con este fichero se podrán comprobar los resultados de la memoria. No usar correctamente `set.seed()` en vuestros scripts impedirá que reproduzca vuestros resultados de la memoria y es un error grave.

4.4 Codificación usada en los ficheros a entregar

El script (o scripts) final o el Rmd y el Html deben ser ficheros grabados con codificación UTF-8 y con la extensión "`.R`", "`.Rmd`" o "`.html`". En caso de tener varios Scripts `indicar claramente el script raiz`.

Si usáis Windows aseguraros de que usáis codificación UTF-8 y no la de Windows. Insisto, **no perderé el tiempo arreglando código o entregas que no compilen**, incluidas las que no carguen correctamente por problemas de codificación de los ficheros.

La idea es proporcionar el código que me permita seguir paso a paso el trabajo realizado, pero solo los pasos útiles (no proporciones el fichero .history que incluya comandos de prueba o cosas descartadas).

5 Criterios de evaluación

5.1 Elementos mínimos para superar la práctica

- Se deben entregar los 3 elementos de la sección 4 y seguir los formatos y directrices indicados en las cajas de texto.
- El script debe ejecutarse sin problemas de consideración (atento a cargar todas las librerias necesarias para evitar errores tontos). Recuerda cargar el fichero `credit.trainIdx.rds` desde el mismo directorio que el script principial, y mejor si lo incluyes en la entrega.
- El trabajo y la memoria deben incluir los elementos y apartados mínimos indicados en la figura 1.
- Se debe entregar en tiempo y forma descritos en esta memoria.

- **SE DEBEN USAR COMO CONJUNTO DE ENTRENAMIENTO/VALIDACIÓN LOS ELEMENTOS INDICADOS EN EL FICHERO** `"credit.trainIdx.rds"`

5.2 Elementos a valorar

- Corrección y la calidad técnica: el abordaje de cada cuestión, tanto en forma como en fondo. Es decir, no solo se valorará que el análisis sea correcto. Además se valorará lo adecuado de la técnica usada para ejecutarlo. Por ejemplo, la visualización de datos resulta de gran utilidad cuando se quiere transmitir un mensaje. Sin embargo, abusar de ella puede ser contraproducente.
- Mensaje visual: La capacidad de ser ilustrativo en los diagramas/explicaciones utilizados **sin caer en el exceso** o la reiteración. Además, se valorará el cuidado puesto en las posibles visualizaciones que se vayan a usar. Esto incluye el uso de colores, tamaños de letra, elección del tipo de plot a usar y la capacidad para transmitir el mensaje que se busca transmitir. En otras palabras, **ser conciso a la par que completo**.
- La profundidad del trabajo (p.e. dar varias pasadas al ciclo de refinamiento del modelo, en base a resultados intermedios) sin caer en el exceso de opciones (p.e. no aporta nada probar 30 métodos de manera superficial, siendo mejor trabajar con menos métodos pero explorándolos con cuidado, criterio, justificación y más profundidad).
- Como se acaba de mencionar, se valorará usar más métodos pero si se hace con criterio, mesura, justificación y profundidad.
- Defensa del trabajo en caso de que el profesor solicite una entrevista presencial de defensa (primera semana tras la vuelta de vacaciones de navidad).
- Solidez en las conclusiones: La calidad técnica del trabajo realizado, especialmente las justificaciones. Una pregunta se ha de responder con evidencias. Las conclusiones obtenidas a partir de los datos deben estar justificadas.
- La **pulcritud** en los ficheros entregados (aunque cuidado al limpiar comentarios o código descartado con no eliminar código útil. Recuerda probar que todo compila de principio a fin limpiando previamente el espacio de trabajo).

5.3 Elementos que *no aportan nada* a la evaluación

- Si el grupo tiene menos de 3 miembros (**si decides hacerlo de forma individual, o en grupo de 2, no se considerará un mérito extra**).
- Elementos artísticos en la memoria (portadas de diseño, etc.).
- Diagramas sin explicación ni justificación. Esto, en particular, contará **negativamente**.

5.4 Tipo de preguntas que uno podría plantearse y contestar a lo largo de la práctica

- En relación al DEA

  - ¿Cuántos ejemplos y variables predictoras tiene el conjunto de datos? Distingue a las numéricas de las categóricas.
  - Para cada predictor categórico, reporta, de la mejor forma posible, la distribución de valores y coméntalo brevemente.
  - Para cada predictor numérico, reporta mínimo, máximo, el primer cuartil, el tercer cuartil, media y mediana. Indica si la variable sigue o no una distribución normal.
  - Responde, mediante una sola línea de código en R, si el conjunto de datos tiene valores nulos y cuántos nulos por columna.
    * ¿Existe alguna columna digna de mención?
    * Elabora una estrategia para el tratamiento de datos nulos y aplícala en el resto de la práctica.
  - Representa las posibles relaciones de la variable de clase, de forma individual, con cada una de los predictores, haciendo uso de algún gráfico con el mecanismo de las facetas (mostrar varios gráficos agrupados por alguna característica), y comenta los resultados.
  - Explora si el análisis de componentes principales es útil, en este problema particular, como mecanismo de visualización e interpretación de los datos para visualizar la separabilidad de las clases y comenta cómo interpretarías los dos primeros componentes principales. Busca la visualización más atractiva posible. Interpreta los datos de manera detallada a la luz de lo que te sugiere el mecanismo de visualización usado.
  - Recuerda analizar con cierta profundidad 3 predictores individualmente (análisis monovariable) y realizar al menos 1 análisis de forma multivariable. No es necesario analizarlos todos (para esta práctica analizar todos es excesivo), solo los tres más interesantes. Si analizas más de los pedidos debería estar MUY JUSTIFICADO.

- En relación al preproceso de datos y feature engineering
  - ¿Hay predictores que no tengan utilidad y serían eliminables? ¿Por y en base a qué?
  - ¿Qué predictores habría que normalizar? ¿Por qué? ¿Cuál sería la estrategia de normalización en cada caso?
  - ¿Podría ser interesante transformar algún atributo o grupos de atributos en uno nuevo? ¿Por qué?

- En relación al Entrenamiento de Modelos/Técnicas
  - ¿Por qué has escogido estudiar éste o aquel modelo frente a otros potencialmente alternativos?
  - ¿Has identificado y explicado cada uno de los hiper-parámetros de configuración a explorar de los modelos escogidos (mínimo los que explora caret para ese modelo)?
  - ¿Por qué has escogido probar esos valores específicos de éste o aquél hiperparámetro?
  - ¿Has experimentado con hiperparámetros "ocultos" que no ofrece directamente caret? P.e. el parámetro `ntree` de Random Forest (la implementación `rf` solo permite modificar `mtry`, pero `ntree` es también muy influyente). Ver sección 11.2.5 del tutorial `caretML.pdf`.
  - ¿Has seguido alguna estrategia para la generación del grid de valores de los hiperparámetros a explorar? ¿Las has detallado? ¿La has justificado?

  - ¿Has hecho exploraciones más profundas/varios ciclos de éste o aquél hiperparámetro afinando el grano? (P.e.: 100, 200, 300, 400, ... y luego 125, 150, 175, 200, 225, 250, 275, ... al detectar un pico en 200) ¿Ha funcionado el ajuste de grano más fino?

- En relación a la comparación de Modelos
  - ¿Hay algún modelo mejor que los demás? Justifícalo.
  - ¿Hay algún modelo en particular que sea mejor que otro en particular? Justifícalo.
  - ¿En base a qué criterio consideras que éste o aquel modelo es mejor? Justifícalo
-

6 Apéndice A: Sobre los algoritmos disponibles en Caret

El paquete Caret, que estamos usando como herramienta básica para generar modelos de ML para la práctica 2, dispone de más de 230 algoritmos diferentes, entre regresión y clasificación. Ver `https://topepo.github.io/caret/available-models.html`

Con el objeto de que no os perdáis entre tanto algoritmo, con esta nota os recomendamos posibles algoritmos que podríais incluir en vuestro análisis.

- El algoritmo `ranger` dentro del paquete R ranger
- El algoritmo `xgbLinear` dentro de `xgboost`
- El algoritmo `glm` que todos conocemos como modelos lineales generalizados (disponible en R)
- El algoritmo `glmnet` del paquete `glmnet`
- El algoritmo `svmLinear`, o bien `svmRadial` del paquete `kernlab`
- El algoritmo `mlpdel` paquete `RSNNS` (o sus variantes `mlpWeightDecay` y `mlpWeightDecayML`)
- El algoritmo `AdaBag` del paquete `adabag`
- El algoritmo `ada` dentro del paquete `ada`
- El algoritmo `C5.0` del paquete `C50`
- El algoritmo `rpart` del paquete `rpart`
- El algoritmo `naive_bayes` del paquete `naivebayes`
- El algoritmo `lda`, un discriminador lineal, el modelo más sencillo posible.
- el algoritmo `rf`, del paquete `randomForest`.
- El algoritmo `gbm` (Stochastic Gradient Boosting) de los paquetes `gbm` y `plyr`.
- El algoritmo `nnet` (Neural Networks) del paquete `nnet`. Cuidado porque es muy sensible a los hiperparámetros que utilicéis y debéis aseguraros de que la red está realmente siendo entrenada (dependiendo de los hiperparámetros puede acabar prematuramente el ajuste y obtener resultados pésimos).

Nótese que aunque en la práctica se menciona que se han de usar cuatro algoritmos, se podría experimentar con más. Téngase en cuenta además que cada algoritmo tiene asociada un coste computacional que puede variar mucho de un algoritmo a otro. Parte de esta complejidad se puede reducir, en alguno de los algoritmos, con un uso apropiado de los posibles hiperparámetros.

De hecho, es habitual probar bastantes algoritmos con configuraciones rápidas de prueba y luego quedarse con los más prometedores. Recuerda, no obstante, que si se incluyen algoritmos adicionales en la memoria, éstos deben analizarse en profundidad y hacer un estudio completo con dicho algoritmo adicional. Debes justificar su inclusión extra y mostrar que se ha trabajado concienzudamente. Cualquier algoritmo adicional tratado superficialmente afectara **NEGATIVAMENTE** a la evaluación.

7 Apéndice B: Grids de hiperparámetros en Caret

Más arriba se ha sugerido contestar a la pregunta:

¿Has identificado y explicado cada uno de los hiper-parámetros de configuración a explorar de los modelos escogidos (mínimo los que explora caret para ese modelo)?

que se refiere a que, para cada uno de los algoritmos de ML que se hayan incluido en la resolución de la práctica, se han de enumerar y explicar qué cometido tiene cada uno de los (hiper)parámetros que usa Caret para optimizar automáticamente el rendimiento del algoritmo. Supongamos que nuestro algoritmo es `gbm`, que se refiere a *gradient boosting machines*. Recordemos que el boosting es una técnica de ML que consiste en combinar varios modelos sencillos (en este caso árboles de decisión) para crear un modelo más potente. En particular va entrenando de forma secuencial varios modelos donde cada modelo "posterior" se ajusta tratando de corregir los errores de los modelos "anteriores". A medida que se entrena cada modelo se la da más peso a las instancias mal clasificadas para que los modelos posteriores se enfoquen en corregir esos errores. De esta forma se trata de reducir el sesgo del modelo y mejorar su rendimiento. Pues bien, para obtener información de los parámetros usados por nuestro algoritmo `gbm` podemos hacer:

```
>library(caret)
>gbminfo = getModelInfo("gbm")
>length(gbminfo)
[1] 2
>names(gbminfo)
[1] "gbm_h2o" "gbm"
```

En la variable `gmbinfo` obtenemos la descripción de dos algoritmos (al menos en mi máquina hay dos instalados), uno del paquete `h2o` y otro del paquete `gbm`. Solamente nos interesa el del paquete `gbm` así que restringimos la explicación a este, haciendo

```
>gbminfo = gbminfo$gbm
```

Y ahora ya podemos consultar sus parámetros de configuración.

```
>gbminfo$parameters
          parameter   class                    label
1           n.trees numeric   # Boosting Iterations
2 interaction.depth numeric          Max Tree Depth
```

```
3         shrinkage numeric               Shrinkage
4    n.minobsinnode numeric Min. Terminal Node Size
```

Y como vemos, son accesibles para este algoritmo cuatro parámetros de configuración que se refieren, respectivamente:

1. al número de árboles a crear (cada iteración de boosting genera un árbol que se añade al conjunto total),
2. la profundidad máxima por árbol (esta será, como mucho, el número de predictores),
3. el `shrinkage` (reducción, o encogimiento) se refiere a un parámetro que controla el crecimiento de los árboles y
4. finalmente, el número de ejemplos que debe haber como mínimo, bajo un nodo, para que éste pueda considerarse un nodo hoja.

En todo caso, y dado que sabemos que es el paquete R `gbm` el que implementa el algoritmo `gbm` tal y como está integrado en Caret, siempre podemos ir al repositorio CRAN a consultar la documentación en la web <https://cran.r-project.org/web/packages/gbm/index.html¿ para averiguar más sobre sus parámetros. Concretamente, y para este algoritmo hay una vignette (suele ser un tutorial en pdf, o un HTML que explica el algoritmo de una forma más didáctica que el manual de referencia y es lo mínimo que uno debe leerse antes de utilizar una técnica, para enterarse un poco de cómo funciona internamente el modelo) accesible desde la página <https://cran.r-project.org/web/packages/gbm/vignettes/gbm.pdf¿. Ahí podemos encontrar información más que de sobra para documentar dicho algoritmo.

Una vez que hemos visto cómo obtener información sobre los hiperparámetros de nuestro algoritmo, vamos a ver cómo diseñar nuestra propia estrategia para dar valores a dichos parámetros cuando la estrategia de optimización del modelo está basada en una búsqueda en malla (i.e. grid search). Este apartado responde a la sugerencia de detallar la estrategia seguida para la generación del grid de valores para hiperparámetros a explorar.

Sigamos el ejemplo de la documentación (accesible en <https://topepo.github.io/caret/model-training-and-tuning.html#basic-parameter-tuning¿) que usa el conjunto Sonar para ejemplificar el uso de `gbm`, pero antes de eso recordemos que nuestro método `train()` en caret, por defecto, crea una parrilla de combinaciones de valores de parámetros de configuración y que usa cada fila de dicha parrilla para configurar diferentes instancias de nuestro algoritmo, las ejecuta todas, las evalúa según el método que hayamos indicado con `trainControl` y luego reporta un modelo final con la fila de la parrilla correspondiente al algoritmo cuyos valores para los hiperparámetros, genera mejores estimadores del rendimiento.

Veamos cuál es la estrategia concreta de la que se hace uso por defecto

```
>gbminfo$grid
function (x, y, len = NULL, search = "grid")
{
    if (search == "grid") {
        out <- expand.grid(interaction.depth = seq(1, len), n.trees = floor((1:len) *
            50), shrinkage = 0.1, n.minobsinnode = 10)
    }
    else {
```

```
        out <- data.frame(n.trees = floor(runif(len, min = 1,
            max = 5000)), interaction.depth = sample(1:10, replace = TRUE,
            size = len), shrinkage = runif(len, min = 0.001,
            max = 0.6), n.minobsinnode = sample(5:25, replace = TRUE,
            size = len))
    }
    out
}
```

Dado que nuestra búsqueda es del tipo grid, y que el valor por defecto que `train` le va a dar al parámetro `len` cuando llame a `grid()` es de 3 (véase el parámetro `tuneLength` en la función `caret::train()` con `help(train)`), si invocamos entonces a dicha función con `len=3`, obtenemos la parrilla:

```
>gbminfo$grid(x=NULL,y=NULL,len=3)
  interaction.depth n.trees shrinkage n.minobsinnode
1                 1      50       0.1             10
2                 2      50       0.1             10
3                 3      50       0.1             10
4                 1     100       0.1             10
5                 2     100       0.1             10
6                 3     100       0.1             10
7                 1     150       0.1             10
8                 2     150       0.1             10
9                 3     150       0.1             10
```

que como vemos genera 9 posibles combinaciones de valores para 4 hiperparámetros ya que `shrinkage` y `n.minbosinnode` tienen valores fijos.

Si queremos comprobar que, en realidad, está haciéndose uso del grid de esta forma, podemos usarlo con el conjunto Sonar como sigue

```
>library(caret)
>library(mlbench)
>data(Sonar)
>set.seed(998)
>inTraining <- createDataPartition(Sonar$Class, p = .75, list = FALSE)
>training <- Sonar[ inTraining,]
>testing  <- Sonar[-inTraining,]
>fitControl <- trainControl(## 10-fold CV
                           method = "repeatedcv",
                           number = 10,
                           ## repeated ten times
                           repeats = 10)
>gbmFit1 <- train(Class ~ ., data = training,
                 method = "gbm",
                 trControl = fitControl,
                 ## This last option is actually one
                 ## for gbm() that passes through
```

```
                verbose = FALSE)
>gbmFit1
Stochastic Gradient Boosting

157 samples
 60 predictor
  2 classes: 'M', 'R'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 10 times)
Summary of sample sizes: 142, 141, 141, 140, 142, 141, ...
Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy   Kappa
  1                   50      0.7811127  0.5576360
  1                  100      0.7933137  0.5824706
  1                  150      0.8045882  0.6046318
  2                   50      0.7930956  0.5815535
  2                  100      0.8147966  0.6252444
  2                  150      0.8237966  0.6431595
  3                   50      0.8072868  0.6104065
  3                  100      0.8309706  0.6576945
  3                  150      0.8344191  0.6644916

Tuning parameter 'shrinkage' was held constant at a value of
 0.1
Tuning parameter 'n.minobsinnode' was held constant at a
 value of 10
Accuracy was used to select the optimal model using the
 largest value.
The final values used for the model were n.trees =
 150, interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode = 10.
```

Y, como podemos comprobar, son esos los experimentos que se ejecutan en realidad. Ahora bien, si en realidad queremos pasar nuestra parrilla propia de valores, como en el ejemplo siguiente,

```
>mygrid = expand.grid(interaction.depth = c(1, 2),
            n.trees = floor((1:4) * 50),
            shrinkage = c(0.1,0.001),
            n.minobsinnode = c(5,10))
>print(mygrid)
   interaction.depth n.trees shrinkage n.minobsinnode
1                  1      50     0.100              5
2                  2      50     0.100              5
3                  1     100     0.100              5
4                  2     100     0.100              5
```

```
5                   1     150     0.100              5
6                   2     150     0.100              5
7                   1     200     0.100              5
8                   2     200     0.100              5
9                   1      50     0.001              5
10                  2      50     0.001              5
11                  1     100     0.001              5
12                  2     100     0.001              5
13                  1     150     0.001              5
14                  2     150     0.001              5
15                  1     200     0.001              5
16                  2     200     0.001              5
17                  1      50     0.100             10
18                  2      50     0.100             10
19                  1     100     0.100             10
20                  2     100     0.100             10
21                  1     150     0.100             10
22                  2     150     0.100             10
23                  1     200     0.100             10
24                  2     200     0.100             10
25                  1      50     0.001             10
26                  2      50     0.001             10
27                  1     100     0.001             10
28                  2     100     0.001             10
29                  1     150     0.001             10
30                  2     150     0.001             10
31                  1     200     0.001             10
32                  2     200     0.001             10
```

lo podemos hacer como sigue

```
>fitControl2 <- trainControl(   ## 10-fold CV
                            method = "repeatedcv",
                            number = 5,
                            ## repeated ten times
                            repeats = 5)
>set.seed(1234)
>gbmFit2 <- train(Class ~ ., data = training,
                 method = "gbm",
                 trControl = fitControl2,
                 tuneGrid=mygrid,
                 verbose = FALSE)
>gbmFit2
Stochastic Gradient Boosting

157 samples
```

```
 60 predictor
  2 classes: 'M', 'R'

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 5 times)
Summary of sample sizes: 127, 125, 125, 125, 126, 126, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.minobsinnode  n.trees  Accuracy   Kappa
  0.001      1                   5               50      0.5350887  0.00000000
  0.001      1                   5              100      0.5389597  0.01091938
  0.001      1                   5              150      0.6547984  0.27746926
  0.001      1                   5              200      0.6916935  0.36189647
  0.001      1                  10               50      0.5350887  0.00000000
  0.001      1                  10              100      0.5453710  0.02516562
  0.001      1                  10              150      0.6598763  0.28761779
  0.001      1                  10              200      0.7005645  0.38153936
  0.001      2                   5               50      0.5350887  0.00000000
  0.001      2                   5              100      0.6155134  0.18419235
  0.001      2                   5              150      0.6851586  0.34420759
  0.001      2                   5              200      0.7245591  0.43119201
  0.001      2                  10               50      0.5350887  0.00000000
  0.001      2                  10              100      0.5925188  0.13287965
  0.001      2                  10              150      0.6889866  0.35241682
  0.001      2                  10              200      0.7106452  0.40219851
  0.100      1                   5               50      0.7583333  0.51112941
  0.100      1                   5              100      0.7759194  0.54704736
  0.100      1                   5              150      0.7784194  0.55213010
  0.100      1                   5              200      0.7836667  0.56287342
  0.100      1                  10               50      0.7772823  0.54953266
  0.100      1                  10              100      0.7988629  0.59318724
  0.100      1                  10              150      0.8013656  0.59807688
  0.100      1                  10              200      0.8025699  0.60070464
  0.100      2                   5               50      0.7628414  0.52151658
  0.100      2                   5              100      0.7845833  0.56434266
  0.100      2                   5              150      0.7857097  0.56626994
  0.100      2                   5              200      0.7946694  0.58447995
  0.100      2                  10               50      0.7911586  0.57796301
  0.100      2                  10              100      0.8102339  0.61573353
  0.100      2                  10              150      0.8075323  0.61041401
  0.100      2                  10              200      0.8125323  0.62021013

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 200, interaction.depth =
 2, shrinkage = 0.1 and n.minobsinnode = 10.
```

8 Apéndice C: RFE

Como ya sabemos `RFE` (Recursive Feature Elimination) es uno de los algoritmos de selección de variables que podemos usar.

En `RFE`, y para esta práctica en particular, se ha de utilizar, internamente un algoritmo de ML para clasificación.

`RFE` espera dicho algoritmo encapsulado en una lista de funciones R, denominada `XXFunc` en donde `XX` hace referencia al algoritmo de ML particular que usa internamente. Por ejemplo, para random forest tenemos `rfFuncs`. Si vemos cómo está definido ese objeto R, que ya está disponible cuando cargamos la librería,

```
>library(caret)
>rfFuncs
$summary
function (data, lev = NULL, model = NULL)
{
    if (is.character(data$obs))
        data$obs <- factor(data$obs, levels = lev)
    postResample(data[, "pred"], data[, "obs"])
}
<bytecode: 0x14f104398>
<environment: namespace:caret>

$fit
function (x, y, first, last, ...)
{
    loadNamespace("randomForest")
    randomForest::randomForest(x, y, importance = TRUE, ...)
}
<bytecode: 0x14f105048>
<environment: namespace:caret>

$pred
function (object, x)
{
    tmp <- predict(object, x)
    if (is.factor(object$y)) {
        out <- cbind(data.frame(pred = tmp), as.data.frame(predict(object,
            x, type = "prob"), stringsAsFactors = TRUE))
    }
    else out <- tmp
    out
}
<bytecode: 0x14f101008>
<environment: namespace:caret>

$rank
```

```
function (object, x, y)
{
    vimp <- varImp(object)
    if (is.factor(y)) {
        if (all(levels(y) %in% colnames(vimp))) {
            avImp <- apply(vimp[, levels(y), drop = TRUE], 1,
                mean)
            vimp$Overall <- avImp
        }
    }
    vimp <- vimp[order(vimp$Overall, decreasing = TRUE), , drop = FALSE]
    if (ncol(x) == 1) {
        vimp$var <- colnames(x)
    }
    else vimp$var <- rownames(vimp)
    vimp
}
<bytecode: 0x14f101dd0>
<environment: namespace:caret>

$selectSize
function (x, metric, maximize)
{
    best <- if (maximize)
        which.max(x[, metric])
    else which.min(x[, metric])
    min(x[best, "Variables"])
}
<bytecode: 0x14f100348>
<environment: namespace:caret>

$selectVar
function (y, size)
{
    finalImp <- ddply(y[, c("Overall", "var")], .(var), function(x) mean(x$Overall,
        na.rm = TRUE))
    names(finalImp)[2] <- "Overall"
    finalImp <- finalImp[order(finalImp$Overall, decreasing = TRUE),
        ]
    as.character(finalImp$var[1:size])
}
<bytecode: 0x14f100c40>
<environment: namespace:caret>
```

vemos que viene con las funciones `summary()`, `fit()`, `pred()`, etc. La primera nos permite imprimir un resumen del resultado. La segunda es la llamada que encapsula la invocación al algoritmo de random forest

particular que usa RFE para crear modelos de ML y la tercera la usa para predecir la clase de los ejemplos que se le pasan como argumento. Si nos fijamos en el método fit para `rfFuncs`,

```
>rfFuncs$fit
function (x, y, first, last, ...)
{
    loadNamespace("randomForest")
    randomForest::randomForest(x, y, importance = TRUE, ...)
}
<bytecode: 0x7fdf04b31b90>
<environment: namespace:caret>
```

Vemos, por el fragmento de código `randomForest::randomForest(x, y, importance = TRUE, ...)` que hace uso de `randomForest` en el paquete `randomForest`. Si hacemos ahora desde la consola

```
>library(randomForest)
>help(randomForest)
```

Veremos que en la invocación a la función R hay un parámetro `ntree`, que se refiere al número de árboles a crear, y que tiene el valor de 500 por defecto. ¿Tenemos una máquina lo suficientemente grande para soportar esto? Dependiendo del conjunto de datos (aunque en este caso es pequeño para evitar este problema) y vuestra máquina podría ser que no. Si fuese el caso tenemos dos opciones. O bien redefinimos nuestro propio objeto `rfFuncs`, en el que le pasamos un valor más modesto al parámetro `ntree`, nosotros mismos, cuando el método fit haga la correspondiente llamada `randomForest::randomForest()` o bien buscamos una alternativa.

Alternativas incluyen, como se puede ver en la documentación cuando hacemos `help(rfeControl)`, el conjunto de funciones `treebagFuncs` (entre otros). Si consultamos su método `fit()`, vemos

```
>treebagFuncs$fit
function (x, y, first, last, ...)
{
    loadNamespace("ipred")
    ipred::ipredbagg(y, x, ...)
}
<bytecode: 0x7fdef4a62378>
<environment: namespace:caret>
```

que el paquete es `ipred` y que en este usa un algoritmo de `bagging` (ver documentación en CRAN, como siempre <https://cran.r-project.org/web/packages/ipred/index.html¿). En el manual de referencia, la función `ipredbagg` aparece especificada como sigue:

```
ipredbagg(y, X=NULL, nbagg=25, control=
                     rpart.control(minsplit=2, cp=0, xval=0),
                     comb=NULL, coob=FALSE, ns=length(y), keepX = TRUE, ...)
```

Y como vemos, el parámetro `nbagg` indica el número de muestreos de `bootstrapping` que ha de hacer. Lo que implica que, como mínimo, cada llamada hace 25 muestreos, que está lejos de los 500 experimentos a realizar en una llamada por defecto al random forest de arriba. Y además, en la ayuda vemos que utiliza el algoritmo `rpart` que es más sencillo y rápido que un random forest. Por lo tanto, os aconsejo utilizar `treebagFuncs` si queréis una ejecución rápida de RFE.

Además, podemos perfectamente paralelizar nuestras ejecuciones de Caret. Téngase en cuenta que Caret realiza múltiples modelos de ML y que todos ellos se generan de manera independiente por lo tanto, tanto las llamadas `train()` como las llamadas `rfe()` son susceptibles de ser paralelizadas ya que cada experimento puede ejecutarse en paralelo. Para leer más puedes ver el capítulo 12 y (muy importante) la sección 11.2.2.3 del tutorial caretML.pdf o bien ir a la documentación de caret aquí `https://topepo.github.io/caret/parallel-processing.html`. Podemos paralelizar de manera sencilla la ejecución de `rfe` como sigue en este ejemplo en el que se carga el conjunto MNIST, se crea posteriormente un cluster de 12 cores (mi máquina puede usar hasta 14, es bueno dejar al menos uno o dos libres para el SO y vuestras tareas), se llama a `rfe` y después se desactiva el cluster. Puedes usar el comando `detectCores()` para saber cuantos cores reconoce R en vuestra máquina.

Para ejecutar este ejemplo hay que bajarse una base de datos que está referida en el repositorio de UCI en `https://archive.ics.uci.edu/dataset/683/mnist+database+of+handwritten+digits` que os lleva a la web original de los autores `https://yann.lecun.com/exdb/mnist/`. En esa web hay 4 ficheros con los datos de Entrenamiento y Test que incluyen un fichero con los datos y otro con las etiquetas. Es interesante (suele serlo siempre) ver la página donde están los datos porque muchas veces nos dan mucha información útil para su proceso. En este caso, por ejemplo, os ofrece una tabla donde os describe los resultados obtenidos por diversos investigadores y las tasas de acierto sobre el conjunto de Test (el que ofrezcan dos bases de datos específicas de Training y Test permite estas cosas porque todo el mundo usa el mismo conjunto de training y de test, por eso, en esta práctica os he indicado qué ejemplos debéis usar en el conjunto de Training y de Test, y así podéis comparar resultados con otros grupos). Bajaos los datos (si os da problemas hay una copia en `https://github.com/golbin/TensorFlow-MNIST/tree/master/mnist/data`), descomprimirlos y cargad el fichero correspondiente al ejemplo.

El formato de esos ficheros es algo especial con lo que os pongo el código para transformarlo en un formato adecuado para CARET (y recomiendo guardarlo una vez convertido).

```
# Primero los datos de Entrenamiento

# Define el path al fichero ubyte de imágenes
mnist_file_path <- file.path("~","Downloads","train-images-idx3-ubyte")
# Define el path al fichero ubyte de etiquetas
mnist_file_path_labels <- file.path("~","Downloads","train-labels-idx1-ubyte")
# Abrimos el fichero en modo binario
con <- file(mnist_file_path, "rb")
# Leemos e ignoramos el "magic number" (primeros 4 bytes)
mnist_magic_number <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de imagenes (siguientes 4 bytes)
mnist_num_images_trn <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de filas (siguientes 4 bytes)
mnist_num_rows <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de columnas (siguientes 4 bytes)
mnist_num_cols <- readBin(con, integer(), size = 4, endian = "big")
```

```
# Leemos los datos de las imágenes (resto de bytes)
# Cada pixel es 1 byte, por lo que leemos el número de bytes apropiado
mnist_image_data <- readBin(con, integer(), size = 1,
                            n = mnist_num_images_trn * mnist_num_rows * mnist_num_cols,
                            signed = FALSE)
# Cerramos la conexión
close(con)
# Reformateamos los datos de las imágenes en una lista de matrices
# Cada matriz es una imagen
mnist_image_list_trn <- list()
for (i in 1:mnist_num_images_trn) {
  start_idx <- (i - 1) * mnist_num_rows * mnist_num_cols + 1
  end_idx <- i * mnist_num_rows * mnist_num_cols
  mnist_image_list_trn[[i]] <- matrix(mnist_image_data[start_idx:end_idx],
                                      nrow = mnist_num_rows, byrow = TRUE)
}
# Ya se podrían visualizar o procesar las imágenes
# Por ejemplo: Mostrar la primera imágen
image(mnist_image_list_trn[[1]], col = gray.colors(256), axes = FALSE)

# Ahora vamos a cargar las etiquetas de cada imagen del training
# Abrimos el fichero en modo binario
con <- file(mnist_file_path_labels, "rb")
# Leemos e ignoramos el "magic number" (primeros 4 bytes)
magic_number <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de etiquetas (siguientes 4 bytes)
mnist_num_labels <- readBin(con, integer(), size = 4, endian = "big")
# Leemos los datos de las etiquetas (resto de bytes)
mnist_labels <- readBin(con, integer(), size = 1, n = mnist_num_labels, signed = FALSE)
# Cerramos la conexión
close(con)

# Los datos, no obstante, no están en un formato adecuado para CARET
# Vamos a crear un dataframe que pueda usarse en CARET
# Convertimos la lista de imágenes (que so matrices) en una matriz 2D
# Cada fila, ahora, es una matriz aplantada (en forma de vector)
mnist_num_images_trn <- length(mnist_image_list_trn)
mnist_num_pixels <- mnist_num_rows * mnist_num_cols  # Total pixels por imágen
# Inicializamos a una matriz vacía que albergará los datos aplanados de la imágen
mnist_image_matrix <- matrix(NA, nrow = mnist_num_images_trn,
                             ncol = mnist_num_pixels)
# Aplana cada imagen y la guardamos en la matriz
for (i in 1:mnist_num_images_trn)
  mnist_image_matrix[i, ] <- as.vector(mnist_image_list_trn[[i]])
# Solo queda combinar la información de las imágenes con su etiqueta (clase) correspondiente
# Con CARET hay que asegurarse que las etiquetas (para clasificación) son factores
```

```
mnist_dataTrn <- data.frame(mnist_image_matrix)
mnist_predictor_labels<-names(mnist_dataTrn)
# Le añadimos las etiquetas como una columna (y nos aseguramos que son factores)
mnist_dataTrn$class <- factor(mnist_labels)
# Reordenamos las columnas para que la clase (class) sea la primera
mnist_dataTrn<-mnist_dataTrn[,c(c("class"),mnist_predictor_labels)]

# Ahora lo mismo con el conjunto de test

# Define el path al fichero ubyte de imágenes
mnist_file_path <- file.path("~","Downloads","t10k-images-idx3-ubyte")
# Define el path al fichero ubyte de etiquetas
mnist_file_path_labels <- file.path("~","Downloads","t10k-labels-idx1-ubyte")
# Abrimos el fichero en modo binario
con <- file(mnist_file_path, "rb")
# Leemos e ignoramos el "magic number" (primeros 4 bytes)
mnist_magic_number <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de imagenes (siguientes 4 bytes)
mnist_num_images_tst <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de filas (siguientes 4 bytes)
mnist_num_rows <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de columnas (siguientes 4 bytes)
mnist_num_cols <- readBin(con, integer(), size = 4, endian = "big")
# Leemos los datos de las imágenes (resto de bytes)
# Cada pixel es 1 byte, por lo que leemos el número de bytes apropiado
mnist_image_data <- readBin(con, integer(), size = 1,
                            n = mnist_num_images_tst * mnist_num_rows * mnist_num_cols,
                            signed = FALSE)
# Cerramos la conexión
close(con)
# Reformateamos los datos de las imágenes en una lista de matrices
# Cada matriz es una imagen
mnist_image_list_tst <- list()
for (i in 1:mnist_num_images_tst) {
  start_idx <- (i - 1) * mnist_num_rows * mnist_num_cols + 1
  end_idx <- i * mnist_num_rows * mnist_num_cols
  mnist_image_list_tst[[i]] <- matrix(mnist_image_data[start_idx:end_idx],
                                      nrow = mnist_num_rows, byrow = TRUE)
}
# Ya se podrían visualizar o procesar las imágenes
# Por ejemplo: Mostrar la primera imágen
image(mnist_image_list_tst[[1]], col = gray.colors(256), axes = FALSE)

# Ahora vamos a cargar las etiquetas de cada imagen del testing
# Abrimos el fichero en modo binario
con <- file(mnist_file_path_labels, "rb")
```

```
# Leemos e ignoramos el "magic number" (primeros 4 bytes)
magic_number <- readBin(con, integer(), size = 4, endian = "big")
# Leemos el número de etiquetas (siguientes 4 bytes)
mnist_num_labels <- readBin(con, integer(), size = 4, endian = "big")
# Leemos los datos de las etiquetas (resto de bytes)
mnist_labels <- readBin(con, integer(), size = 1, n = mnist_num_labels, signed = FALSE)
# Close the connection
close(con)

# Los datos, no obstante, no están en un formato adecuado para CARET
# Vamos a crear un dataframe que pueda usarse en CARET
# Convertimos la lista de imágenes (que so matrices) en una matriz 2D
# Cada fila, ahora, es una matriz aplantada (en forma de vector)
mnist_num_images_tst <- length(mnist_image_list_tst)
mnist_num_pixels <- mnist_num_rows * mnist_num_cols    # Total pixels por imágen
# Inicializamos a una matriz vacía que albergará los datos aplanados de la imágen
mnist_image_matrix <- matrix(NA, nrow = mnist_num_images_tst,
                             ncol = mnist_num_pixels)
# Aplana cada imagen y la guardamos en la matriz
for (i in 1:mnist_num_images_tst)
  mnist_image_matrix[i, ] <- as.vector(mnist_image_list_tst[[i]])
# Solo queda combinar la información de las imágenes con su etiqueta (clase) correspondiente
# Con CARET hay que asegurarse que las etiquetas (para clasificación) son factores
mnist_dataTst <- data.frame(mnist_image_matrix)
mnist_predictor_labels<-names(mnist_dataTst)
# Le añadimos las etiquetas como una columna (y nos aseguramos que son factores)
mnist_dataTst$class <- factor(mnist_labels)
# Reordenamos las columnas para que la clase (class) sea la primera
mnist_dataTst<-mnist_dataTst[,c(c("class"),mnist_predictor_labels)]

# limpiamos la memoria de variables grandes sin uso
remove(mnist_image_matrix)
remove(mnist_image_data)

# Ahora podemos grabar estos dataframes en disco para facilitar
# futuros experimentos
write.csv(mnist_dataTrn,file="pretty_mnist_train.csv",row.names = F)
write.csv(mnist_dataTst,file="pretty_mnist_test.csv",row.names = F)

# Por desgracia, al ser las clases "números" considera la columna class
# como numérica con lo que hay que, o bien especificarlo al cargar, o bien
# convertirla luego en un factor (o CARET hará regresion en vez de clasificación)

# Es decir, luego los podemos cargar con comandos como (cada uno con una forma)
trndata = read.csv("pretty_mnist_train.csv",colClasses=c("factor",rep("integer",784)))
tstdata = read.csv("pretty_mnist_test.csv")
```

```
tstdata$class<-factor(tstdata$class)
```

Ahora ya podemos ir con el ejemplo:

```
trndata = read.csv("pretty_mnist_train.csv",
                   colClasses=c("factor",rep("integer",784)))
tstdata = read.csv("pretty_mnist_test.csv",
                   colClasses=c("factor",rep("integer",784)))


predictors = trndata[,-1]
outcome = trndata[,1]

library(doParallel)
cl <- makePSOCKcluster(detectCores()-2)
registerDoParallel(cl)

lmProfile = rfe(predictors, outcome,
                sizes = c(70,90,110,150,200),
                rfeControl = rfeControl(functions = treebagFuncs,
                                        method = "cv",
                                        number = 5,
                                        verbose = FALSE))

stopCluster(cl)
summary(lmProfile)
```

Nótese que los valores que doy a los parámetros de las llamadas `rfe` y `rfeControl` no tienen por qué ser los adecuados. Es simplemente un ejemplo.

Para ver los resultados finales podemos sencillamente imprimir el profile. También es interesante hacer un summary de éste para ver qué tipo de información almacena.

```
> lmProfile

Recursive feature selection

Outer resampling method: Cross-Validated (5 fold)

Resampling performance over subset size:

 Variables Accuracy  Kappa AccuracySD  KappaSD Selected
        70   0.9198 0.9108   0.002935 0.003262
        90   0.9290 0.9210   0.002966 0.003298
       110   0.9338 0.9264   0.004336 0.004819
       150   0.9417 0.9352   0.002844 0.003161
       200   0.9473 0.9414   0.001291 0.001436
       784   0.9502 0.9447   0.003312 0.003682        *
```

```
The top 5 variables (out of 784):
   X381, X406, X321, X380, X409
```

```
> summary(lmProfile)
             Length Class      Mode
pred           0    -none-     NULL
variables      4    data.frame list
results        5    data.frame list
bestSubset     1    -none-     numeric
fit            5    classbagg  list
optVariables 784    -none-     character
optsize        1    -none-     numeric
call           5    -none-     call
control       14    -none-     list
resample     104    data.frame list
metric         1    -none-     character
maximize       1    -none-     logical
perfNames      2    -none-     character
times          3    -none-     list
resampledCM    0    -none-     NULL
obsLevels     10    -none-     character
dots           0    -none-     list
```

También podemos predecir con el conjunto de ejemplos de Test con el mejor modelo que encontró al evaluar la mejor combinación de variables. Luego hacemos una matriz de confusión con los resultados:

```
> tstRes<-predict(lmProfile$fit,tstdata)
> caret::confusionMatrix(tstRes,tstdata$class)
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  965    0    9    3    2    8   11    2    5    6
         1    0 1119    1    0    1    5    2   10    2    2
         2    0    2  975   16    4    0    1   20    9    2
         3    0    4    9  948    0   13    1    6    9    8
         4    0    1    7    0  925    2    6    0    6   24
         5    6    2    2   19    2  833    7    0   12    7
         6    3    2    6    0    5   10  918    0    8    2
         7    3    1    9    9    1    3    0  982    2    6
         8    1    4   12   12   13    8   11    2  910   11
         9    2    0    2    3   29   10    1    6   11  941
```

```
Overall Statistics

               Accuracy : 0.9516
                 95% CI : (0.9472, 0.9557)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9462

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: 0 Class: 1 Class: 2 Class: 3 Class: 4
Sensitivity            0.9847   0.9859   0.9448   0.9386   0.9420
Specificity            0.9949   0.9974   0.9940   0.9944   0.9949
Pos Pred Value         0.9545   0.9799   0.9475   0.9499   0.9526
Neg Pred Value         0.9983   0.9982   0.9936   0.9931   0.9937
Prevalence             0.0980   0.1135   0.1032   0.1010   0.0982
Detection Rate         0.0965   0.1119   0.0975   0.0948   0.0925
Detection Prevalence   0.1011   0.1142   0.1029   0.0998   0.0971
Balanced Accuracy      0.9898   0.9917   0.9694   0.9665   0.9684
                     Class: 5 Class: 6 Class: 7 Class: 8 Class: 9
Sensitivity            0.9339   0.9582   0.9553   0.9343   0.9326
Specificity            0.9937   0.9960   0.9962   0.9918   0.9929
Pos Pred Value         0.9360   0.9623   0.9665   0.9248   0.9363
Neg Pred Value         0.9935   0.9956   0.9949   0.9929   0.9924
Prevalence             0.0892   0.0958   0.1028   0.0974   0.1009
Detection Rate         0.0833   0.0918   0.0982   0.0910   0.0941
Detection Prevalence   0.0890   0.0954   0.1016   0.0984   0.1005
Balanced Accuracy      0.9638   0.9771   0.9757   0.9630   0.9627
```