Escalabilidad en grandes conjuntos de datos

Ignacio Cordón Castillo

El código con el que se han efectuado todas las particiones y ejecutado los algoritmos de la segunda parte de la práctica está disponible en `bin/main.R`. El del estudio de escalabilidad, en `bin/scalability.sh`.

Características del ordenador

- SO: Linux Ubuntu 15.04, 64 bits con núcleo `4.4.0-040400-generic`
- Procesador: `Intel Core i7-4700HQ CPU, 2.40GHz × 8`
- RAM: `11.6 GiB`
- Versión de Java 64 bits: `openjdk version "1.8.0_45-internal"`
- Versión de `Weka`: 3.6.11
- Versión de `R`: `3.2.3, Wooden Christmas-Tree`
- Versión de `RWeka`: 0.4.27

Datasets

Para el desarrollo de la práctica se han empleado 4 datasets:

Covertype

581012 instancias, 12 características y 7 categorías.

Kddcup99

4898431 instancias, 41 características y 23 categorías.

Protein

1000000 instancias, 20 características y 2 categorías.

Pokerhand

1025010 instancias, 10 características y 10 categorías.

Estudio de escalabilidad

Consistía en dividir cada conjunto de datos en 20% de test y otro 80% de training. Se estudia la escalabilidad entrenando clasificadores `J48` y `Random Forest` con 50 árboles, sobre particiones del train del 20%, 40%, 60%, 80% y 100% para evaluar los resultados obtenidos sobre test, y efectuar una comparación en cuanto precisión, ejecución y tamaño del train. Se ha usado como semilla aleatoria `12345678`

Se ha programado una función de `R`, disponible en `bin/partitioning.R` que efectúa la división de un dataset parado como parámetro `data` al 20% test y 80% training, dividiendo a su vez training en 5 particiones disjuntas y estratificadas (test también se ha extraído con muestreo estratificado, conservando la distribución de clases original). Para ello se han empleado las funciones `createDataPartition` y `createFolds` del paquete `caret` de `R`.

```
make.partition <- function(data, name){
  train.index <- createDataPartition(data$class, p = 0.8, list = F, times = 1)
  train <- data[ train.index, ]
  test  <- data[-train.index, ]

  folds <- createFolds(train$class, 5)

  # Returns map of folds to the original data
  partition <- list(
    train = lapply(folds, function(selected){
      train[selected, ]
    }),
    test = test)

  save (list = c('partition'), file = paste(name,".RData", sep = ""))
}
```

Se han leído los datasets y se ha aplicado la función anterior.

```
covertype <- read.arff("../data/covertype.arff")
kddcup <- read.arff("../data/kddcup99.arff")
protein <- read.arff("../data/protein.arff")
pokerhand <- read.arff("../data/pokerhand.arff")


datasets <- c(covertype, kddcup, protein, pokerhand)
datasets.names <- c("covertype", "kddcup", "protein", "pokerhand")


partitions <- lapply(1:length(datasets), function(i) {
  make.partition(datasets[i], datasets.names[i])
})
```

Una vez obtenidas las particiones, se han fusionado las dos primeras para obtener una con el 40% de train, las tres primeras para obtener otra con el 60% de train, y sucesivamente, y se ha volcado cada una de las particiones para cada dataset con la función `write.arff` del paquete `RWeka` en un dataset de nombre `./data/train{porcentaje}-{nombre-dataset}` o `./data/test-{nombre-dataset}` (p.e. `train20-covertype`, `test-covertype`).

A su vez, se han guardado las particiones correspondientes a un dataset en un archivo de la forma `{nombre-dataset}.RData` para liberar toda la memoria RAM posible y disponer de la mayor cantidad posible para la ejecución de algoritmos.

Se ha automatizado la ejecución de `Weka` sobre cada una de las particiones, con un script `bash` para obtener los resultados en ficheros homónimos en la carpeta `results`

```
#!/bin/bash
```

```
training=(train20 train40 train60 train80 train100)
datasets=(covertype kddcup protein pokerhand)


for train in ${training[*]}
do
    for d in ${datasets[*]}
    do
        echo "Haciendo J48 sobre ${train}-${d}"
        java -cp ~/weka-3-8-1/weka.jar -Xmx8g weka.classifiers.trees.J48 \
             -t ../data/${train}-${d}.arff -T ../data/test-${d}.arff \
             > ../results/J48-${train}-${d}
        echo "Haciendo Random Forest sobre ${train}-${d}"
        java -cp ~/weka-3-8-1/weka.jar -Xmx8g weka.classifiers.trees.RandomForest -I 50 \
             -t ../data/${train}-${d}.arff -T ../data/test-${d}.arff \
             > ../results/RF-${train}-${d}

    done
done
```

Resultados

Los resultados obtenidos han sido:

dataset	algoritmo	tamaño train(%)	tiempo entrenamiento	precisión train	precisión test
covertype	RF	20	16.05	99.9935	92.6110
covertype	RF	40	32.98	99.9941	94.8141
covertype	RF	60	50.05	99.9964	95.8184
covertype	RF	80	69.84	99.9954	96.3339
covertype	RF	100	93.01	99.9983	96.7702
covertype	J48	20	12.00	96.3921	87.8494
covertype	J48	40	30.30	97.7572	91.1566
covertype	J48	60	61.09	98.2534	92.8150
covertype	J48	80	108.84	98.5220	93.6979
covertype	J48	100	130.39	98.7092	94.4673
kddcup	RF	20	110.71	99.9999	99.9909
kddcup	RF	40	244.92	99.9998	99.9934
kddcup	RF	60	362.33	99.9999	99.9941
kddcup	RF	80	680.16	99.9998	99.9950
kddcup	RF	100	920.44	99.9997	99.9956
kddcup	J48	20	46.22	99.9915	99.9821
kddcup	J48	40	100.85	99.9943	99.9883
kddcup	J48	60	176.34	99.9949	99.9895
kddcup	J48	80	383.98	99.9959	99.9914
kddcup	J48	100	523.17	99.9967	99.9929
protein	RF	20	23.27	99.9788	79.7699
protein	RF	40	51.79	99.9819	80.1199
protein	RF	60	84.38	99.9821	80.4864
protein	RF	80	126.86	99.9817	80.7054
protein	RF	100	155.67	99.9836	80.9169
protein	J48	20	31.69	86.7400	77.6864
protein	J48	40	131.82	86.8900	78.0069

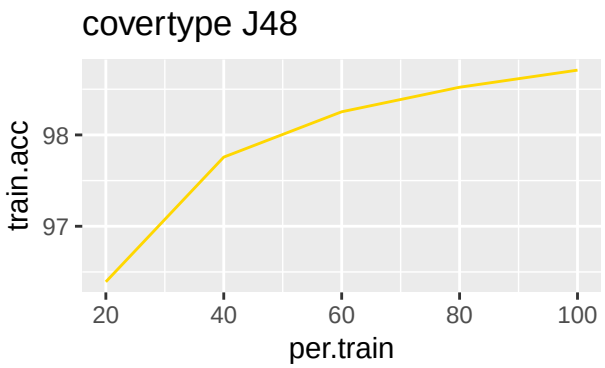
dataset	algoritmo	tamaño train(%)	tiempo entrenamiento	precisión train	precisión test
protein	J48	60	315.34	87.0554	78.1179
protein	J48	80	473.00	87.0181	78.3539
protein	J48	100	690.56	87.1009	78.1604
pokerhand	RF	20	24.68	99.9939	54.9810
pokerhand	RF	40	60.27	99.9957	59.2869
pokerhand	RF	60	104.05	99.9982	61.6060
pokerhand	RF	80	139.21	99.9957	61.9616
pokerhand	RF	100	178.17	99.9962	62.9918
pokerhand	J48	20	49.31	75.1479	61.5162
pokerhand	J48	40	189.50	76.4880	66.7367
pokerhand	J48	60	398.14	75.3603	64.9401
pokerhand	J48	80	835.26	76.0268	65.0894
pokerhand	J48	100	1446.76	76.0282	66.0865

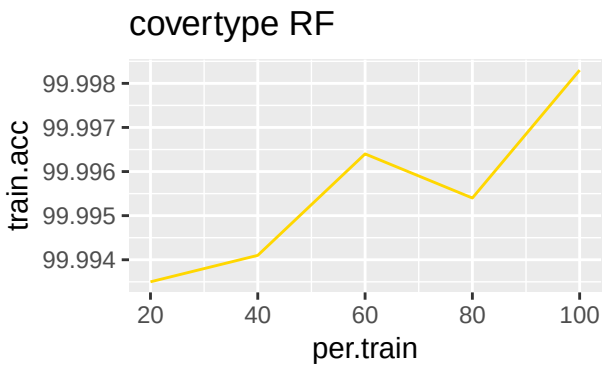
Las siguientes series de gráficas representan porcentaje de train usado respecto al /accuracy/ obtenido tanto en train como en test. La tercera serie de gráficas representa el tiempo que ha tardado en entrenarse el modelo respecto al porcentaje de train usado. A juzgar por las curvas de tiempo en función del tamaño del train usado, todos los algoritmos escalan de manera peor que lineal el tiempo en función del porcentaje de train usado, aunque las curvas que se aprecian son muy suaves, quizás debido a las características de potencia del ordenador usado; esto encaja con el hecho de que J48 es $O(n^2)$ en función del número de instancias (n).

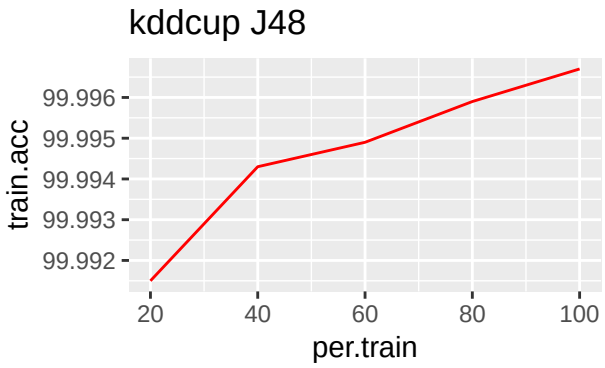
Se observa asimismo que no siempre un mayor porcentaje de train implica mayor precisión en test. `pokerhand` parece un dataset muy difícil de aprender; tanto Random Forest como J48 obtienen resultados que no alcanzan el 70% en este dataset, y de hecho Random Forest está efectuando *overfitting* claramente en dicho dadaset, con *accuracies* en train que rozan el 100%, y empeorando los resultados de J48 en test.

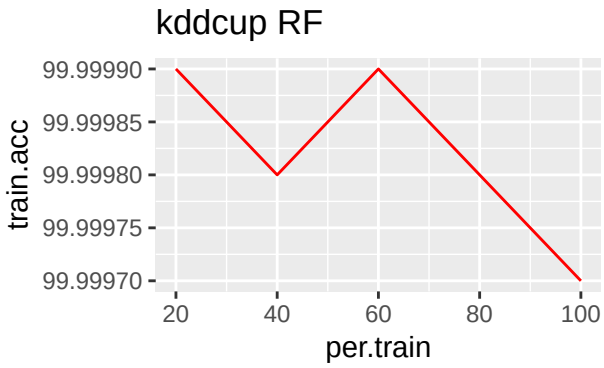
El *accuracy* en test parece estabilizarse (las gráficas describen formas logarítmicas) e incluso empeorar (en el caso de `protein` con J48) a medida que aumentamos el conjunto de entrenamiento, lo que contrasta con el hecho de que el tiempo de entrenamiento escala de forma superior a lineal.

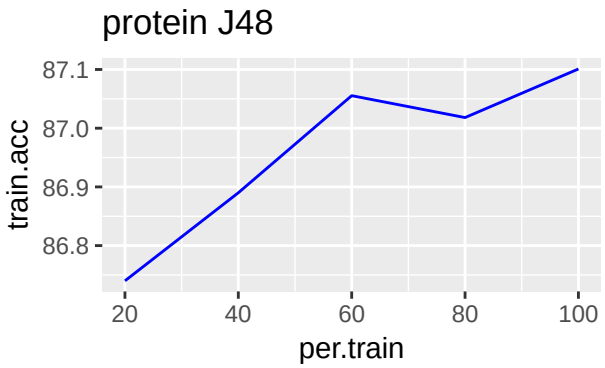
Asimismo se observa en cualquier caso que Ranfom Forest parece ser una técnica más robusta que J48 (recordemos que es un *bagging* y está diseñado para aprender instancias difíciles), obteniendo mejores tasas de precisión en test (exceptuando el caso del *overfitting*), y mejores tiempos de ejecución que J48; una excepción a esto último es el caso de `kddcup`, que es el dataset de los analizados con mayor número de instancias, características y categorías.

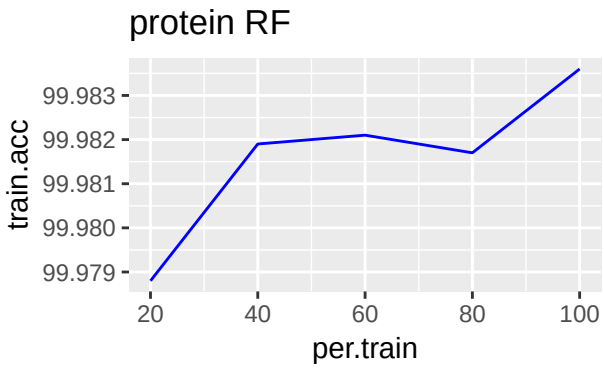

covertype J48
train.acc
98
97
20
40
60
80
100
per.train

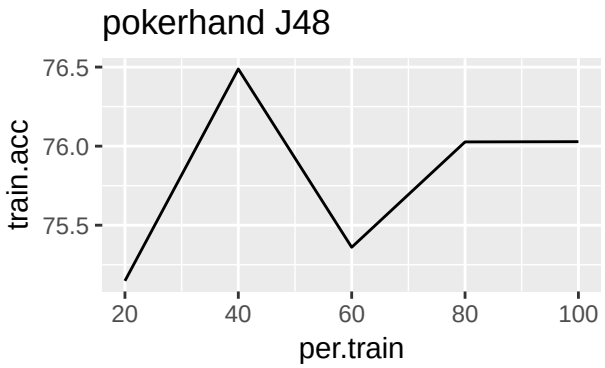

covertype RF
train.acc
99.998
99.997
99.996
99.995
99.994
20
40
60
80
100
per.train

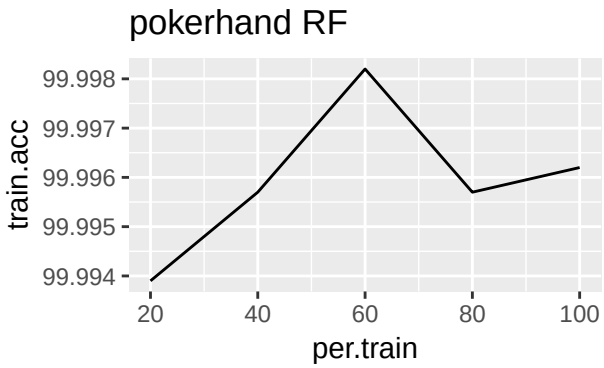

kddcup J48
train.acc
99.996
99.995
99.994
99.993
99.992
20
40
60
80
100
per.train

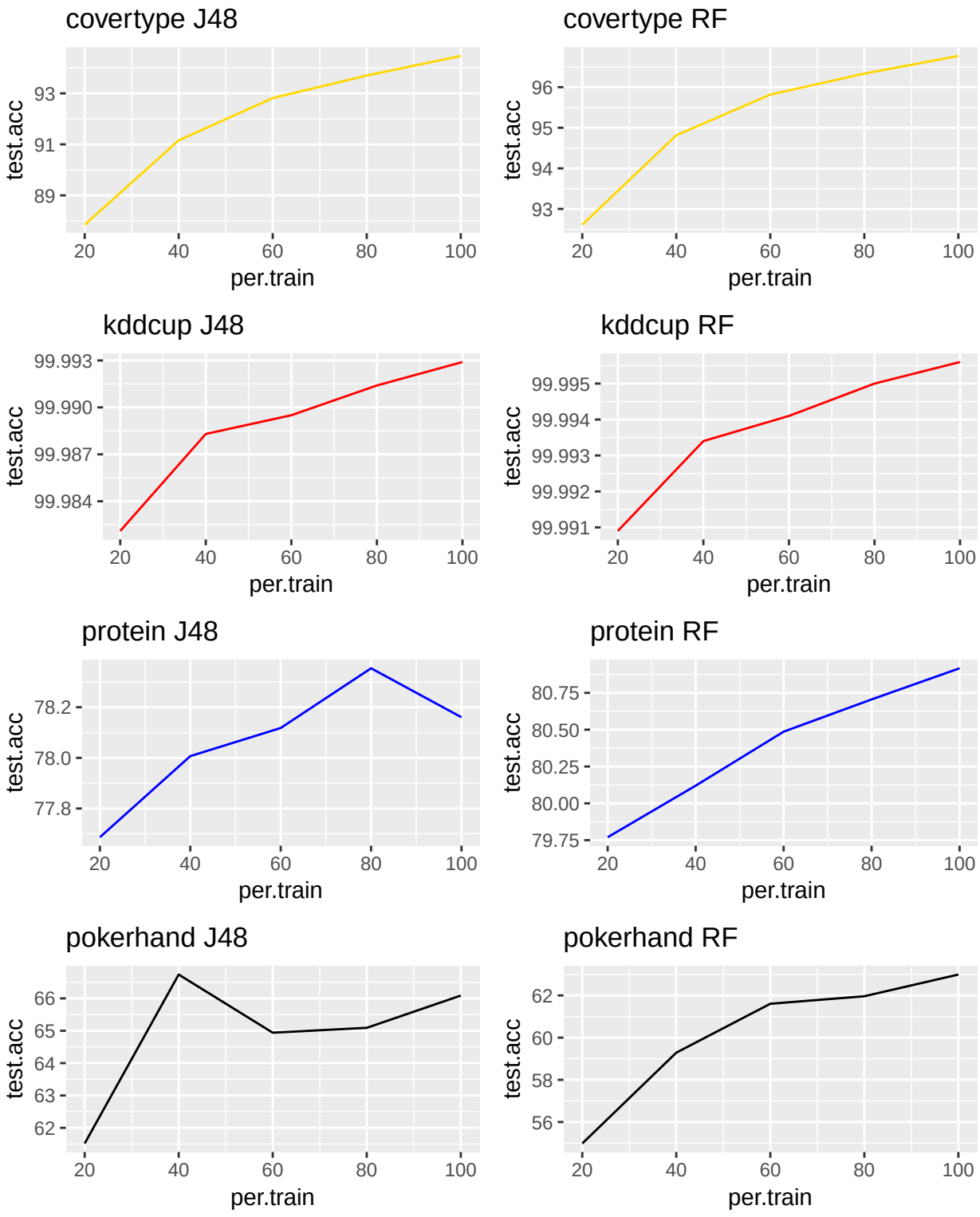

kddcup RF
train.acc
99.99990
99.99985
99.99980
99.99975
99.99970
20
40
60
80
100
per.train

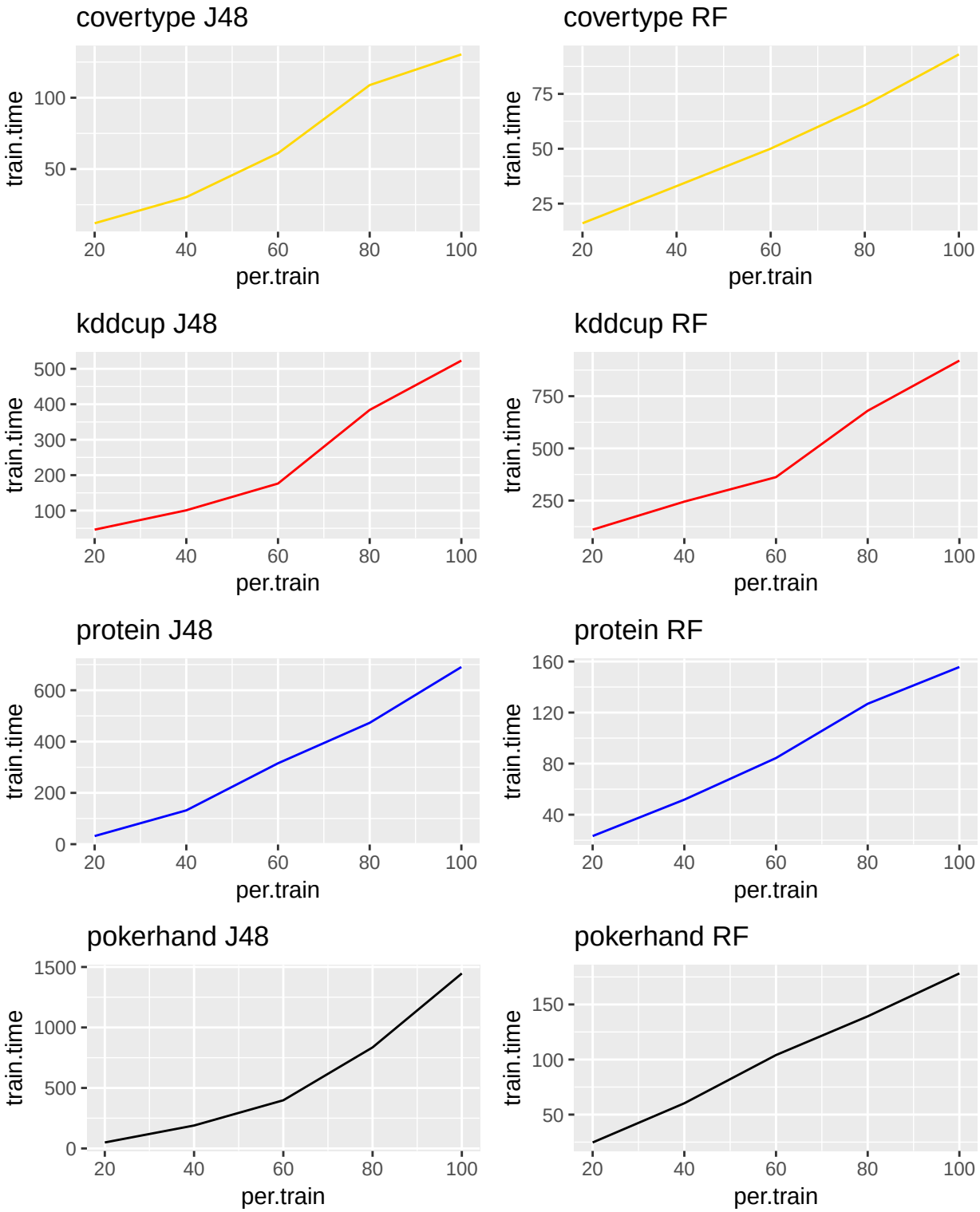

protein J48
train.acc
87.1
87.0
86.9
86.8
20
40
60
80
100
per.train


protein RF
train.acc
99.983
99.982
99.981
99.980
99.979
20
40
60
80
100
per.train


pokerhand J48
train.acc
76.5
76.0
75.5
20
40
60
80
100
per.train


pokerhand RF
train.acc
99.998
99.997
99.996
99.995
99.994
20
40
60
80
100
per.train


covertype J48
test.acc
93
91
89
20
40
60
80
100
per.train
covertype RF
test.acc
96
95
94
93
20
40
60
80
100
per.train
kddcup J48
test.acc
99.993
99.990
99.987
99.984
20
40
60
80
100
per.train
kddcup RF
test.acc
99.995
99.994
99.993
99.992
99.991
20
40
60
80
100
per.train
protein J48
test.acc
78.2
78.0
77.8
20
40
60
80
100
per.train
protein RF
test.acc
80.75
80.50
80.25
80.00
79.75
20
40
60
80
100
per.train
pokerhand J48
test.acc
66
65
64
63
62
20
40
60
80
100
per.train
pokerhand RF
test.acc
62
60
58
56
20
40
60
80
100
per.train


covertype J48
train.time
100
50
20
40
60
80
100
per.train
covertype RF
train.time
75
50
25
20
40
60
80
100
per.train
kddcup J48
train.time
500
400
300
200
100
20
40
60
80
100
per.train
kddcup RF
train.time
750
500
250
20
40
60
80
100
per.train
protein J48
train.time
600
400
200
0
20
40
60
80
100
per.train
protein RF
train.time
160
120
80
40
20
40
60
80
100
per.train
pokerhand J48
train.time
1500
1000
500
0
20
40
60
80
100
per.train
pokerhand RF
train.time
150
100
50
20
40
60
80
100
per.train

Técnica de estratificación

Se pretende reducir el tiempo de entrenamiento de los clasificadores entrenando el clasificador correspondiente (J48 o Random Forest) sobre cada uno de las cinco particiones de entrenamiento de tamaño 16% (20% del 80% que representaba el total del training) obtenidas en el estudio de la escalabilidad y efecuando una predicción sobre el test.

Para fusionar las predicciones hechas sobre test, se emplea una estrategia de voto:

- **Voto simple**: se toma la clase mayoritaria de entre la predicha por los cinco modelos entrenados.
- **Voto ponderado**: se obtiene la confianza de la predicción de cada clasificador para todas las clases, se suman las confianzas para cada clase de los cinco clasificadores, y se toma la clase que tenga mayor suma de confianzas.

En caso de empate en ambos modos de voto, se toma como clase la de aquel clasificador que prediciendo alguna de las clases empatadas, más confianza ha obtenido en el total del train.

Esta parte de la práctica se ha hecho enteramente con `R` y `RWeka`, programando:

- Una función para obtener las predicciones sobre las 5 particiones para cada dataset. Nótese que en el caso de `RandomForest` se ha empleado la opción `I=50` para ejecutar el algoritmo con 50 árboles.

```
get.predictions <- function(algorithm, param){
  result <- lapply(1:length(datasets.names), function(n.index){
    name <- datasets.names[n.index]
    load(file=paste(name, ".RData", sep=""))

    current.pred <- lapply(1:5, function(i){
      train.model <- algorithm(class ~ ., data = partition$train[[i]], control = param)
      predict(train.model, newdata = partition$test, type = c("probability"))
    })

    # Force R's garbage collector
    gc()
    current.pred
  })

  names(result) <- datasets.names
  result
}
```

```
J48.predictions <- get.predictions(J48, Weka_control())
# Random Forest with 50 trees
RF.predictions <- get.predictions(RF, Weka_control(I=50))
```

- Funciones de cálculo de las predicciones hechas, a través de las cuales hemos obtenido para cada dataset, y cada *chunk* del 20% del training, la precisión obtenida por el clasificador, tanto J48, como Random Forest.

```
### Get class with more confidence for each dataset
get.classes <- function(predictions){
  lapply(predictions, function(set.pred){
```

```
    columns <- colnames(set.pred[[1]])

    lapply(set.pred, function(prob.matrix){
      apply(prob.matrix, 1, function(x){ columns[ which.max(x) ] })
    })
  })
}


acc.rate <- function(true, calculated){
  length( which(calculated == true) ) / length(calculated)
}

get.accuracies <- function(algorithm.class){
  result <- lapply( 1:length(datasets.names), function(n.index){
    name <- datasets.names[ n.index ]
    load(file=paste(name, ".RData", sep=""))

    true.class <- partition$test$class

    ifelse (length(algorithm.class[n.index]) > 1,
            sapply( 1:length(algorithm.class[[n.index]]), function(i){
              calculated.class <- algorithm.class[[ n.index ]][[ i ]]
              acc.rate( true.class,calculated.class )
            }),
            acc.rate( true.class, algorithm.class[[ n.index ]] ))
  })

  names(result) <- datasets.names

  result
}
```

```
RF.class  <- get.classes(RF.predictions)
J48.class <- get.classes(J48.predictions)


J48.accuracies <- get.accuracies(J48.class)
RF.accuracies  <- get.accuracies(RF.class)
```

- Función de cálculo de votos simples y ponderados, a partir de la cual obtenemos posteriormente los correspondientes *accuracies*.

```
vote.prediction <- function(predictions, accuracies, ponderate){
  # For each dataset, calc most voted class and break ties with most accurate pred
  result <- lapply( 1:length( predictions ), function(i){
    dataset.pred <- predictions[[i]]

    if(! ponderate){
      votes <- lapply( dataset.pred, function(m){
        matrix(
          sapply( seq_len(nrow(m)), function(row.index){
            row <- m[row.index,]
```

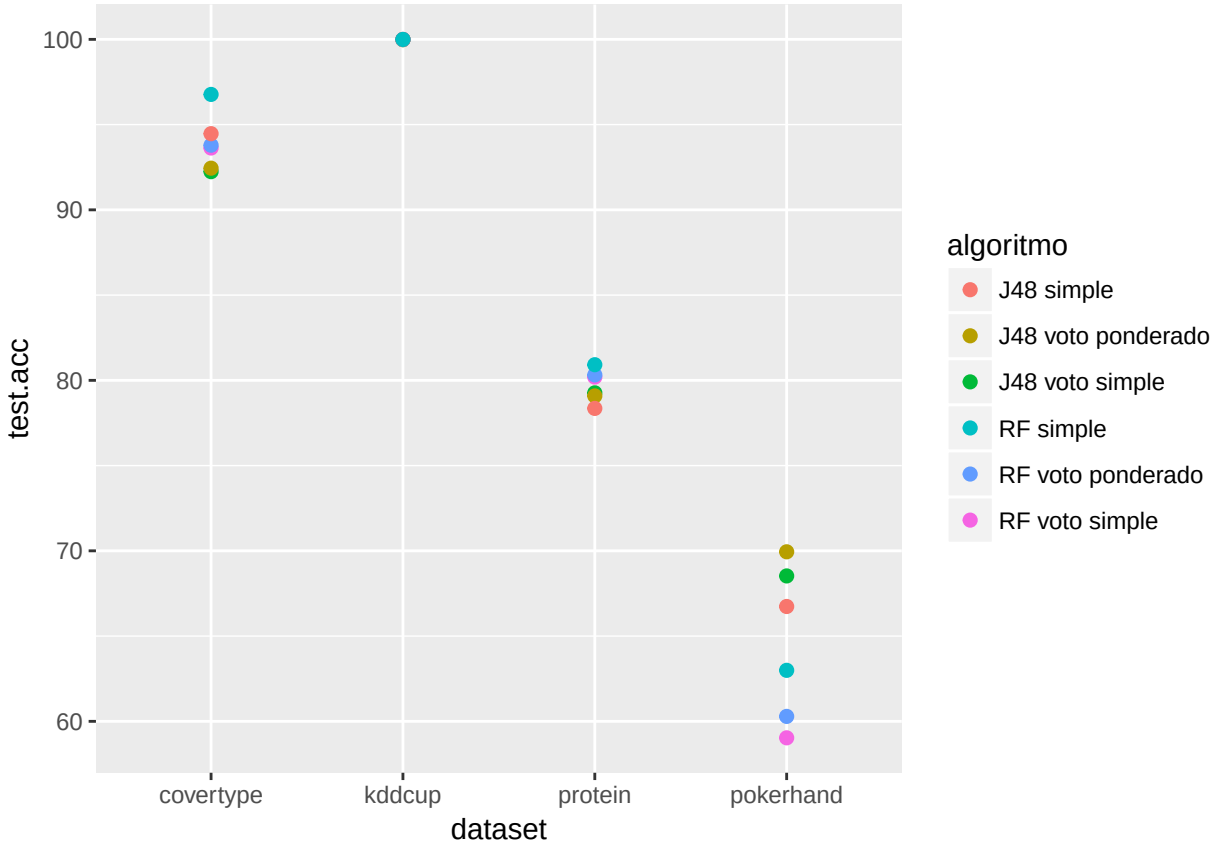
```
            # Substitute max probability prediction with 1, otherwise 0
            result <- rep(0,length(row))
            result[ which.max(row) ] <- 1
            result
          }),
          ncol = ncol(m), byrow=T)
      })
    }else{
      votes <- dataset.pred
    }

    # Sum predictions probabilities row by row for each of the five matrixes
    sum.votes <- Reduce('+', votes)
    categories <- colnames(dataset.pred[[1]])

    sapply( seq_len( nrow(sum.votes) ), function(row.index){
      row <- sum.votes[row.index,]

      index.max <- which(row == max(row))

      if(length(index.max) == 1){
        categories [ index.max ]
       }else{
         pred.categories <- sapply(votes, function(m){
           categories[ which.max(m[row.index, ]) ]
         })
         pred.indexes <- which(pred.categories %in% categories[ index.max ])
         categories[ which.max( accuracies[[i]][ pred.indexes ] ) ]
      }
    })
  })

  names(result) <- datasets.names
  result
}
```

```
J48.simple.vote.preds <- vote.prediction(J48.predictions, J48.accuracies, ponderate=F)
J48.simple.vote.acc <-  get.accuracies(J48.simple.vote.preds)

RF.simple.vote.preds <- vote.prediction(RF.predictions, RF.accuracies, ponderate=F)
RF.simple.vote.acc <-  get.accuracies(RF.simple.vote.preds)

J48.ponderate.vote.preds <- vote.prediction(J48.predictions, J48.accuracies, ponderate=T)
J48.ponderate.vote.acc <-  get.accuracies(J48.ponderate.vote.preds)

RF.ponderate.vote.preds <- vote.prediction(RF.predictions, RF.accuracies, ponderate=T)
RF.ponderate.vote.acc <-  get.accuracies(RF.ponderate.vote.preds)
```

Resultados

Los resultados en cuanto a precisión obtenidos han sido (donde RF/J48 simple representa el mejor resultado obtenido de las 5 ejecuciones con diferente tamaño de train en la evaluación anterior de la escalabillidad):

algoritmo	covertype	kddcup	protein	pokerhand
J48 voto simple	92.24010	99.98540	79.2659	68.52799
J48 voto ponderado	92.44406	99.98591	79.0864	69.94361
RF voto simple	93.62651	99.99173	80.1899	59.03277
RF voto ponderado	93.79346	99.99204	80.3174	60.28937
J48 simple	94.46730	99.99290	78.3539	66.73670
RF simple	96.77020	99.99560	80.9169	62.99180



Observamos que el modelo de voto ponderado o simple en el caso de `covertype`, que es el dataset más pequeño, no mejora a los mejores resultados obtenidos con Random Forest y J48 durante las ejecuciones de escalabilidad, y el caso de `kddcup` tampoco aporta mucha información el hecho de que el modelo de votos (tanto simple como ponderado) sea ligeramente inferior a los resultados sin votos puesto que sobre dicho dataset los algoritmos simples ya funcionaban muy bien, produciendo tasas de clasificación cercanas al 100%.

En los casos de los datasets `protein` y `pokerhand` sí se aprecia mayor mejoría de los algoritmos con votación respecto a los correspondientes simples, puesto que sobre las gráficas de escalabilidad observabamos que para estos datasets grandes tamaños del conjunto de entrenamiento disminuían la precisión sobre el test.

Podemos conjeturar por tanto que para datasets sobre los que producimos *overfitting* es preferible, además de porque mejora los tiempos de ejecución, porque mejora los *accuracies* obtenidos sobre los conjuntos de prueba, emplear el modelo de votos para entrenar los clasificadores.