Trabajo Guiado Minería de Flujo de Datos

Alberto Armijo Ruiz
26256219V
armijoalb@correo.ugr.es

Ejercicio 1 **3**

Ejercicio 2 **7**

Ejercicio 3 **7**

Ejercicio 4 **8**

Ejercicio 1

Se pide comparar una los algoritmos *Naive Bayes* y *Hoeffding Trees*. Para ello se utilizará un 1000000 de datos, con una frecuencia de muestreo de 10000 datos. Como generador de los datos se utilizará el generador *RandomTreeGenerator*, el cual genera un flujo de datos mediante un árbol generado de forma aleatoria. Para evaluar los algoritmos se utilizará la técnica *Test-Then-Train*; con este método primero se testea el algoritmo para ver su precisión y después se vuelve a entrenar el algoritmo.

Para evaluar los algoritmos se debe utilizar las siguientes sentencias:
java -cp moa.jar moa.DoTask \
"EvaluateInterleavedTestThenTrain -l bayes.NaiveBayes \
-s generators.RandomTreeGenerator \
-i 1000000 -f 10000" > nbresults.csv (media:73,55)

java -cp moa.jar moa.DoTask \
"EvaluateInterleavedTestThenTrain -l trees.HoeffdingTree \
-s generators.RandomTreeGenerator \
-i 1000000 -f 10000" > htresults.csv (media:92,17)

Analizando los resultados, el algoritmo *HoeffdingTrees* parece ser mejor que el algoritmo *Naive Bayes*; para asegurarnos de que los resultados son reales, se realizará un estudio haciendo 30 pruebas para cada algoritmo, y con un test paramétrico/no paramétrico para ver si existen diferencias significativas, y si existen diferencias significativas, podremos decir que el algoritmo *HoeffdingTree* es mejor que *Naive Bayes*. A continuación se muestra un script para ejecutar las pruebas y otro script en R para realizar el test sobre los datos obtenidos.

```
#!/bin /bash

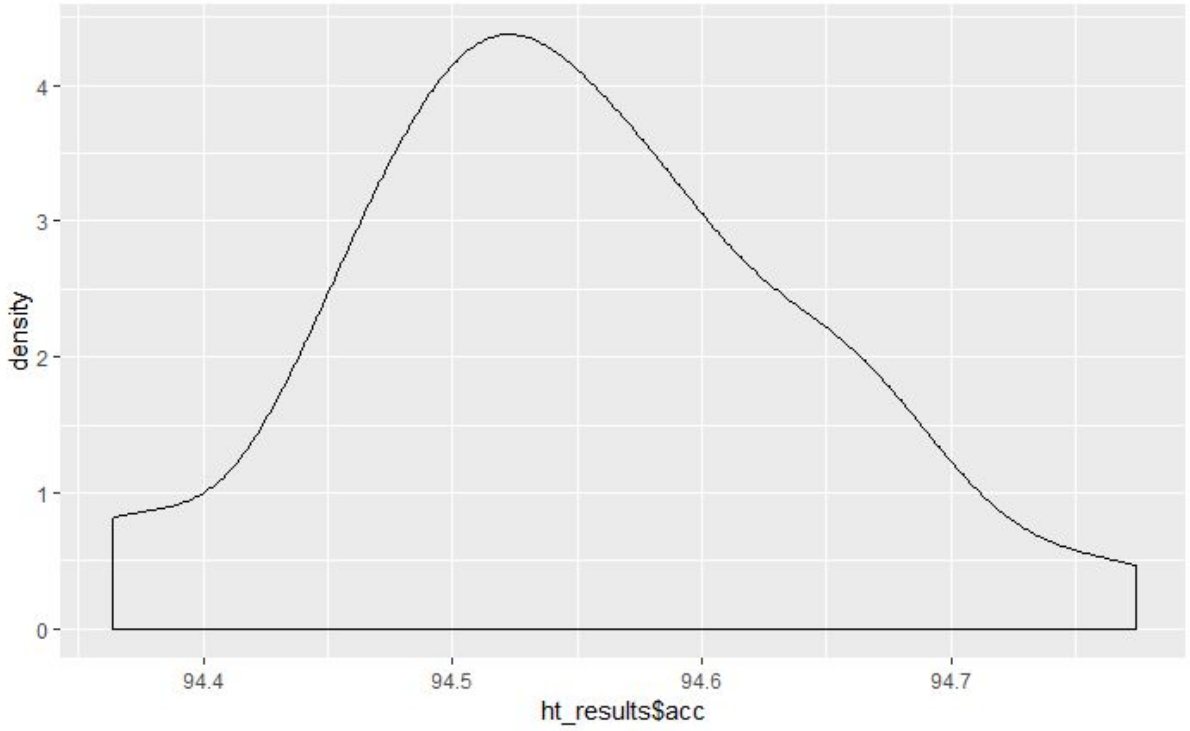
echo "Ejecutando script para generar diferentes semillas"

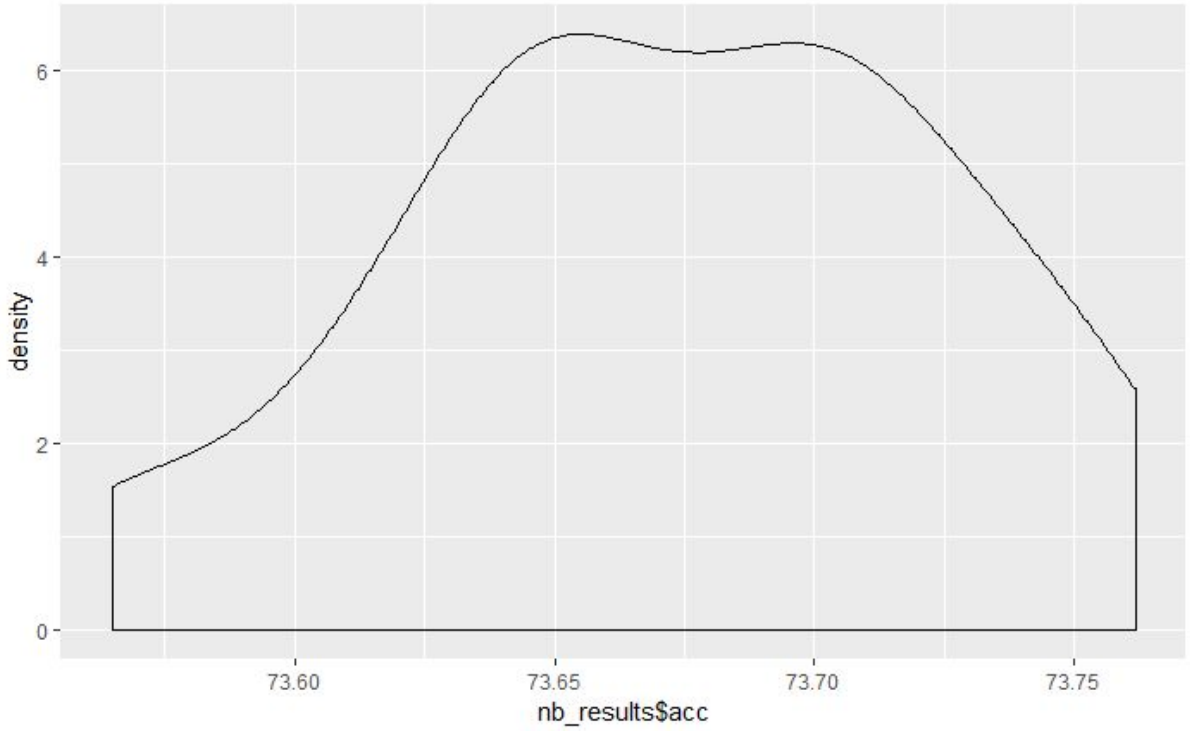
for i in {1..30}; do
echo "Creando ht$i.csv"
java -cp moa.jar moa.DoTask "EvaluateInterleavedTestThenTrain -l trees.HoeffdingTree \
-s (generators.RandomTreeGenerator -i $i) -i 1000000 -f 10000" > ht$i.csv
done

for i in {1..30}; do
echo "Creando nb$i.csv"
java -cp moa.jar moa.DoTask "EvaluateInterleavedTestThenTrain -l bayes.NaiveBayes \
-s (generators.RandomTreeGenerator -i $i) -i 1000000 -f 10000" > nb$i.csv
done
```

```
```{r}
dataset_ht = sapply(seq(1,30),function(x){
  read.csv(paste('resultados/ht',x,'.csv',sep=''),header=TRUE,stringsAsFactors=FALSE)[5]})
dataset_nb = sapply(seq(1,30),function(x){
  read.csv(paste('resultados/nb',x,'.csv',sep=''),header=TRUE,stringsAsFactors=FALSE)[5]})
```

```{r}
ht_results = lapply(dataset_ht,function(x){
  x[length(x)]
})
ht_results = data.frame(as.numeric(ht_results))
colnames(ht_results) = c('acc')
nb_results = lapply(dataset_nb,function(x){
  x[length(x)]
})
nb_results = data.frame(as.numeric(nb_results))
colnames(nb_results) = c('acc')
```
```


density
4
3
2
1
0
94.4
94.5
94.6
94.7
ht_results$acc


density
6
4
2
0
73.60
73.65
73.70
73.75
nb_results$acc

```{r}
shapiro.test(nb_results$acc)
shapiro.test(ht_results$acc)
```

```
        Shapiro-Wilk normality test

data:  nb_results$acc
W = 0.97381, p-value = 0.6478


        Shapiro-Wilk normality test

data:  ht_results$acc
W = 0.9824, p-value = 0.8852
```

Por lo que se puede ver, las distribuciones no son normales, por lo que utilizaremos un test no paramétrico. Se utilizará el test de *Wilcoxon* para ver si hay diferencias significativas entre ambos modelos, si el p-valor es menor que 0.05; podremos decir que existen diferencias significativas entre ambos algoritmos, y el algoritmo con un mejor comportamiento será el que tenga una mejor media.

```{r}
tabla_comp = cbind(nb_results,ht_results)
names(tabla_comp)= c('NaiveBayes','HoeffdingTree')

# Normalizamos los datos.
tabla_comp = (tabla_comp/100)
head(tabla_comp)

# aplicamos el test.
nb_vs_ht = wilcox.test(tabla_comp[,1],tabla_comp[,2], alternative = "two.sided",
                       paired = TRUE)
rmas = nb_vs_ht$statistic
pvalue = nb_vs_ht$p.value
nb_vs_ht = wilcox.test(tabla_comp[,2],tabla_comp[,1],alternative="two.sided",
                       paired=TRUE)
rmenos = nb_vs_ht$statistic
rmas
rmenos
pvalue
```

```
1.862645e-09
```

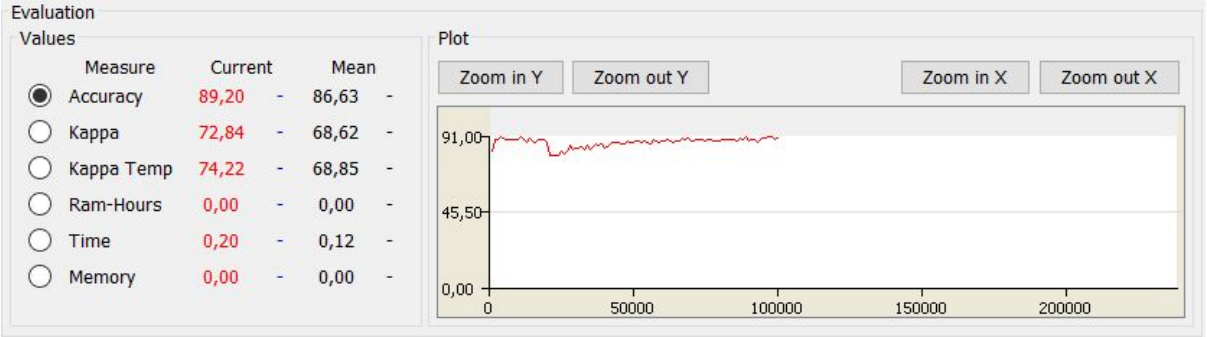
Como se puede ver, el p-valor es menor que 0.05, por lo cual podemos decir que el algoritmo *Hoeffding Tree* es mejor que el algoritmo *Naive Bayes*. Además de gracias al test y las pruebas realizadas, sabemos que el algoritmo *Hoeffding Tree* se trata de un algoritmo incremental, preparado para ser usado con flujos de datos mientras que el algoritmo *Naive Bayes* no.

Ejercicio 2

Para este ejercicio se pide generar un flujo de datos con el generador *SEAGenerator* con las funciones 2 y 3. Utilizaremos el algoritmo *Naive Bayes* para medir cómo afecta el Concept Drift a los algoritmos. La sentencia necesaria para ejecutar esto es la siguiente:

java -cp moa.jar moa.DoTask \
"EvaluatePrequential -l bayes.NaiveBayes -s \
(ConceptDriftStream -s (generators.SEAGenerator -f 2) -d \
(generators.SEAGenerator -f 3) -p 20000 -w 100) -i 100000 -f 1000"

La salida en la GUI de MOA es la siguiente:



Como se puede ver, cuando se produce el cambio de concepto se puede ver una bajada en la precisión que para la cual conforme van llegando más datos el algoritmo se va adaptando a los nuevos datos.

Ejercicio 3

Para este ejercicio se pide entrenar un clasificador *Naive Bayes* con datos generados por el generador *SEAGenerator* con la función 2. Tras esto se debe evaluar la calidad del modelo con un stream de datos generados con la función 2 y 3.

Para entrenar un modelo se debe utilizar la siguiente sentencia:
java -cp moa.jar moa.DoTask \
“LearnModel -l bayes.NaiveBayes -s (generators.SEAGenerator -f 2)
-m 100000”

Para evaluar el modelo con un stream de datos generados con *SEAGenerator* las funciones 2 y 3 (ConceptDrift).
java -cp moa.jar moa.DoTask \
“EvaluateModel -m (LearnModel -l bayes.NaiveBayes -s
(generators.SEAGenerator -f 2) -m 100000) -s
(ConceptDriftStream -s (generators.SEAGenerator -f 2) -d

(generators.SEAGenerator -f 3) -p 20000 -w 100) -i 100000”

Los resultados obtenidos son los siguientes:

```
classified instances = 100.000
classifications correct (percent) = 80,344
Kappa Statistic (percent) = 57,301
Kappa Temporal Statistic (percent) = 54,583
Kappa M Statistic (percent) = 39,098
model training instances = 100.000
model serialized size (bytes) = 0.0
```

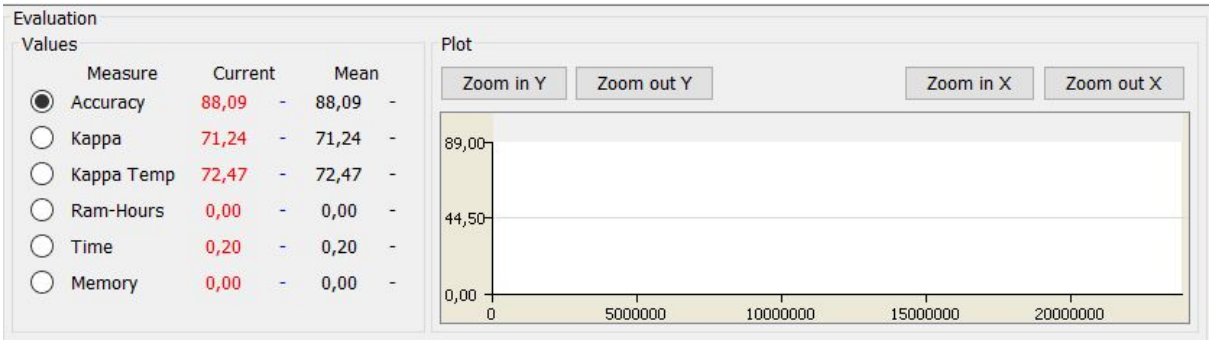
Como se puede ver, los resultados bajan bastante al producirse el concept drift; esto ocurre porque el modelo ha sido entrenado con datos con el generador *SEAGenerator* utilizando la función 2, por lo cual cuando los datos pasan a ser generados con la función 3 el modelo no está preparado para evaluar dichos datos. Si el modelo se re-entrenara, podría ser capaz de reaccionar al concept drift, como pasa en el ejercicio anterior.

Ejercicio 4

En este ejercicio se pide entrenar el modelo mediante el método *Test-Then-Train* para re-entrenar el modelo *Naive Bayes*; para ello se utilizará el algoritmo *DDM* para detectar el cambio de concepto y re-entrenar el modelo. Como flujo de datos se utilizará el mismo que para el ejercicio anterior. La sentencia necesaria para este ejercicio es la siguiente:

java -cp moa.jar moa.DoTask
“EvaluateInterleavedTestThenTrain -l
(moa.classifiers.drift.SingleClassifierDrift -l bayes.NaiveBayes
-d DDM) -s (ConceptDriftStream -s (generators.SEAGenerator -f 2) -d
(generators.SEAGenerator -f 3) -p 20000 -w 100) -i 100000”

La salida de la sentencia es la siguiente:



Como se puede ver en la salida, los resultados son mejores que en el algoritmo anterior, un 8% mayor. Esta mejora se debe al algoritmo *DDM* que detecta el cambio de concepto cuando el flujo de datos pasa a ser generador por la función 3 del generador *SEAGenerator* y re-entrena el modelo para que se adapte al cambio producido.