Trabajo autónomo II: Minería de Flujo de Datos

Máster DATCOM

Serie Temporales y Minería de Flujo de Datos

Autor
Alberto Armijo Ruiz
armijoalb@correo.ugr.es

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

18 de abril de 2019

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Parte Práctica

1.1. Ejercicio 1

En este ejercicio se debe entrenar un modelo *HoeffdingTree* con un flujo de datos generado con *WaveformGenerator*, semilla inicializada a 2 y un millón de instancias. Tras esto se debe utilizar otro flujo generado con *WaveformGenerator* utilizando como semilla el 4 y un millón de datos. Tras esto se debe probar el modelo *HoeffdingAdaptiveTree* con los mismo paramétros. Dado que con un solo ejemplo no vale para saber que algoritmo es mejor que otro, se realizarán 10 pruebas con cada uno de los clasificadores y se estudiaran los resultados obtenidos.

Las sentencias necesarias para realizar lo anterior son las siguientes:

java -cp moa.jar moa.DoTask "EvaluateModel -m (LearnModel -l trees.HoeffdingTree -s (generators.WaveformGenerator -i 2) -m 1000000) -s (generators.WaveformGenerator -i 4)"

java -cp moa.jar moa.DoTask "EvaluateModel -m (LearnModel -l trees.HoeffdingAdaptiveTree -s (generators.WaveformGenerator -i 2) -m 1000000) -s (generators.WaveformGenerator -i 4)"

Las opciones que se pueden ver en las sentencias son las siguientes:

- *EvaluateModel* -m: para expecificar el modelo.
- *LearnModel* -l: para expecificar modelo que se va a entrenar.
- *LearnModel* -s: para el flujo de datos con el que se entrena el modelo.
- *LearnModel* -m: para expecificar el número de instancias con el que se entrena el modelo.
- *WaveformGenerator* -i: para expecificar la semilla con la que se inicializa el generador.
- *EvaluateModel* -s: para expecificar el flujo de datos con el que se evalúa el modelo que ha sido entrenado.

Para realizar las pruebas con diferentes semillas se han creados dos scripts para automatizar el proceso. Tras esto se han mirado los resultados y se ha realizado un test para ver si existen diferencias entre ambos algoritmos.

```
#!/bin/bash

echo "PRUEBAS CON HOEFFDING TREE\n" > resultados.txt

for i in {1..10}; do
java -cp moa.jar moa.DoTask \
"EvaluateModel -m (LearnModel -l trees.HoeffdingTree -s
(generators.WaveformGenerator -i $i) -m 1000000) -s
(generators.WaveformGenerator -i 4)" >> resultados.txt
echo "\n semilla $i \n" >> resultados.txt
done
```

(a) script para HoeffdingTree

```
#!/bin/bash

echo "PRUEBA CON HOEFFDING TREE ADAPTATIVO\n" > resultados_adapt.txt

for i in {1..10}; do
java -cp moa.jar moa.DoTask \
"EvaluateModel -m (LearnModel -l trees.HoeffdingAdaptiveTree -s
(generators.WaveformGenerator -i $i) -m 1000000) -s
(generators.WaveformGenerator -i 4)" >> resultados_adapt.txt
done
```

(b) script para HoeffdingAdaptiveTree

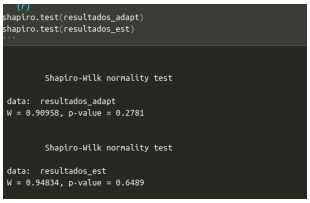
Figura 1.1: Scripts para generar las pruebas de forma automática

Los resultados obtenidos por los clasificadores son los siguientes.

Semilla	**Accuracy**	**Kappa**	**Accuracy Adaptive**	**Kappa Adaptive**
1	84,509	76,765	84,521	76,783
2	84,512	76,77	84,474	76,712
3	84,59	76,887	84,416	76,625
4	84,666	77,001	84,465	76,699
5	84,481	76,723	84,262	76,395
6	84,342	76,514	84,368	76,554
7	84,799	77,2	84,271	76,408
8	84,153	76,231	84,243	76,36
9	84,641	76,963	84,478	76,719
10	84,578	76,869	84,326	76,491

Cuadro 1.1: Resultados pruebas ejercicio 1

Como se puede ver los valores son muy similares, aún así se utilizará un test para confirmar que no hay diferencias de rendimiento entre ambos clasificadores.



```
shapiro.test(resultados_adapt)
shapiro.test(resultados_est)
```

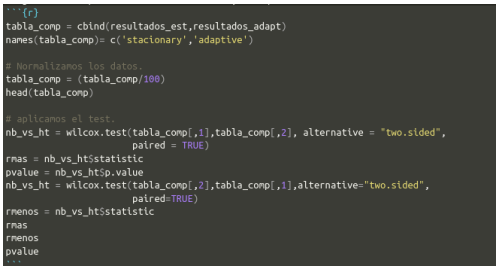
```
        Shapiro-Wilk normality test

data:  resultados_adapt
W = 0.90958, p-value = 0.2781


        Shapiro-Wilk normality test

data:  resultados_est
W = 0.94834, p-value = 0.6489
```

(a) test de normalidad



```
tabla_comp = cbind(resultados_est,resultados_adapt)
names(tabla_comp)= c('stacionary','adaptive')

# 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
```

(b) test de wilcoxon

El resultado del test de Wilcoxon es: 0.03710938

Por lo que se puede ver en los resultados, los dos algoritmos están ofreciendo los mismos resultados; esto puede ser posiblemente porque estamos entrenando ambos algoritmos de forma offline, por lo cual tanto HoeffdingTree como HoeffdingAdaptiveTree están aprendiendo sobre el mismo conjunto de datos y no son capaces de responder a cambios de concepto como el que tiene el flujo de datos generado con la semilla 4. Además, si realizamos un estudio sobre los resultados, el test muestra que existen diferencias significativas entre ambos algoritmos, pero después la diferencia entre la mediana de un algoritmo y otro son casi inexistentes.

1.2. Ejercicio 2

Para este ejercicio se debe hacer lo mismo que en el anterior pero cambiando el método de evaluación por *InterleavedTestThenTrain*. A diferencia que en el ejercicio anterior, no se necesita entrenar un modelo, ya que el método de evaluación *InterleavedTestThenTrain* primero testea los datos y después los utiliza para entrenar, por lo tanto está entrenando y testeando datos durante todo el proceso.Las sentencias necesarias para evaluar los modelos son los siguientes:

java -cp moa.jar moa.DoTask
“EvaluateInterleavedTestThenTrain -l trees.HoeffdingTree -s (generators.WaveformGenerator -i 2) -i 1000000 -f 10000“ ¿resultados_online.txt

java -cp moa.jar moa.DoTask
“EvaluateInterleavedTestThenTrain -l trees.HoeffdingAdaptiveTree -s (generators.WaveformGenerator -i 2) -i 1000000 -f 10000“ ¿resultados_online_adative.txt

Las opciones que se muestran para estas sentencias son las mismas que para el ejercicio anterior menos la opción -l de *EvaluateInterleavedTestThenTrain*; esta función se utiliza para evaluar un modelo con Test-Then-Train especificado mediante la opción -l. Las otras opciones que se pueden ver son -i que especifica el número de instancias totales con las que se evaluará el modelo y -f que especifica el número de instancias que llegan en cada momento.

Para realizar un estudio sobre ambos modelos; *HoeffdingTree* y *HoeffdingAdaptiveTree*, se han creado dos scripts para realizar pruebas con diferentes valores para la semilla de inicialización del generador *WavefromGenerator*. Los scripts son los siguientes:

```
#!/bin/bash

for i in {1..10}; do
java -cp moa.jar moa.DoTask \
"EvaluateInterleavedTestThenTrain -l
trees.HoeffdingAdaptiveTree -s
(generators.WaveformGenerator -i $i)
-i 1000000 -f 10000" >> resultados_online_adative$i.csv
done
```

(c) script para HoeffdingTree

```
#!/bin/bash

for i in {1..10}; do
java -cp moa.jar moa.DoTask \
"EvaluateInterleavedTestThenTrain -l
trees.HoeffdingTree -s
(generators.WaveformGenerator -i $i)
-i 1000000 -f 10000" >> resultados_online$i.csv
done
```

(d) script para HoeffdingAdaptiveTree

Figura 1.2: Scripts para generar las pruebas de forma automática con EvaluateInterleavedTestThenTrain

Los resultados obtenidos por los clasificadores son los siguientes.

Semilla	**Accuracy**	**Accuracy Adaptive**	**Kappa**	**Kappa Adaptive**
1	83,8903	83,8042	75,83623569	75,7071683
2	83,7851	83,7313	75,67749802	75,5967623
3	83,8876	83,7875	75,82954147	75,6792025
4	84,0451	83,7961	76,06946036	75,69604195
5	83,8402	83,7144	75,75999473	75,57128698
6	83,9062	83,8406	75,85905507	75,76071273
7	83,8867	83,7784	75,82927699	75,66688337
8	83,8687	83,8968	75,8028419	75,84506754
9	83,7875	83,8282	75,6817212	75,7427829
10	83,8479	83,9	75,77149369	75,84955066

Cuadro 1.2: Resultados pruebas ejercicio 2

Al igual que en el ejercicio anterior, no parece haber diferencias entre los dos algoritmos; aún así se realizará un test para ver si existen diferencias entre ellos.

Para ver si hay diferencias significativas entre ambos algoritmos, cargaremos los datos y comprobaremos si existen diferencias significativas.La forma de comprobar la normalidad de las soluciones y el test se realiza de la misma forma que en el ejercicio anterior. Al igual que en el ejercicio anterior, los resultados nos siguen una distribución normal, por lo que se utilizará el test de Wilcoxon para comprobar si hay diferencias de rendimiento entre ambos algoritmos.

Por los resultados del test (0.02734375), debería de haber diferencias significativas, pero al hacer la media de la precisión de ambos algoritmos se puede ver que son casi

iguales, por lo que descartaremos la hipótesis del test y diremos que no hay diferencias entre los algoritmos. Es normal que no haya diferencias significativas entre los modelos, ya que a no ser que en el flujo de datos haya un cambio de concepto ambos modelos se comportan de forma igual.

1.3. Ejercicio 3

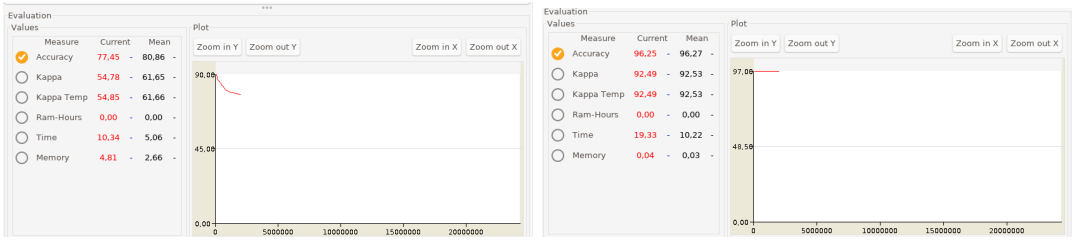
Para este ejercicio se debe entrenar un *HoeffdingTree* online con test-then-train sobre 2M de instancias y freq. muestreo de 100k generando los datos con un generador *RandomRBFGeneratorDrift*, con semilla aleatoria igual a 1 para generación de instancias, generando 2 clases, 7 atributos, 3 centroides, drift en todos los centroides y velocidad de cambio igual a 0.001. También se probará el modelo *HoeffdingAdaptiveTree*. Las sentencias necesarias para ejecutar ambos modelos son las siguientes:

java -cp moa.jar moa.DoTask
"EvaluateInterleavedTestThenTrain -l trees.HoeffdingTree -s
(generators.RandomRBFGeneratorDrift -s 0.001 -k 3 -r 1 -i 1 -a 7 -n 3) -i 2000000 -f 100000"

java -cp moa.jar moa.DoTask
"EvaluateInterleavedTestThenTrain -l trees.HoeffdingAdaptiveTree -s
(generators.RandomRBFGeneratorDrift -s 0.001 -k 3 -r 1 -i 1 -a 7 -n 3) -i 2000000 -f 100000"

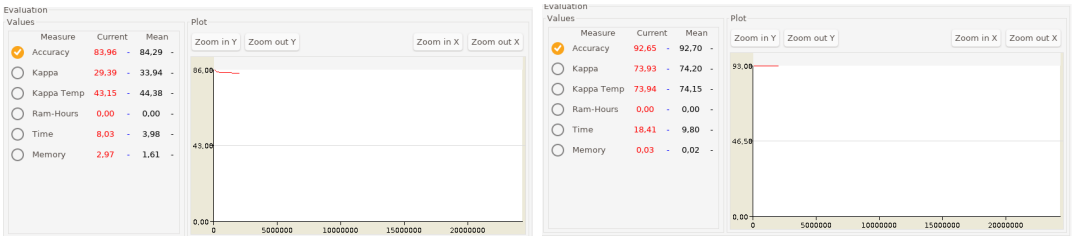
Las nuevas opciones que hay a diferencia de los ejercicios anteriores son las opciones del generador *RandomRBFGeneratorDrift*. La opción -s especifica la velocidad de cambio en los centroides, la opción -r especifica la semilla para la generación de modelos, la opción -i especifica la semilla para la generación de instancias, la opción -a especifica el número de atributos que tiene cada instancia, la opción -k especifica el número de centroides que cambian y la opción -n especifica el número de centroides del flujo de datos.

Para realizar un estudio sobre el comportamiento de los algoritmos, se ha utilizado la GUI de MOA para poder ver una gráfica sobre la precisión de los modelos sobre el flujo de datos que se genera. Para este ejercicio se han utilizado 3 semillas diferentes para cada uno de los modelos, los resultados obtenidos son los siguientes.



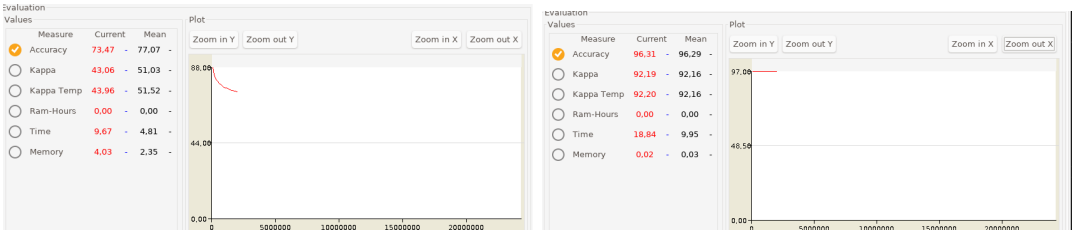
(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.3: Resultados InterleavedTestThenTrain con semilla 1



(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.4: Resultados InterleavedTestThenTrain con semilla 2



(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.5: Resultados InterleavedTestThenTrain con semilla 3

Si nos fijamos en los resultados que se pueden ver en las imágenes de la GUI de MOA, se puede ver que el clasificador *HoeffdingTree* obtiene peores resultados que su versión adaptativa, esto es así porque el algoritmo *HoeffdingTree* no está preparado para los cambios de concepto, por ello cuando hay un cambio de concepto la precisión del algoritmo baja bastante. En cambio, el clasificador *HoeffdingAdaptiveTree* sí que está preparado para detectar cambios de concepto, para ello utiliza el método *ADWIN*, una vez detectado el cambio de concepto puede re-entrenar el árbol, podar ciertas ramas que ya no sirvan, etc. Por ello, no sufre una bajada en la precisión como le pasa al algoritmo *HoeffdingTree* y es apto para clasificación con flujos que tienen cambios de concepto.

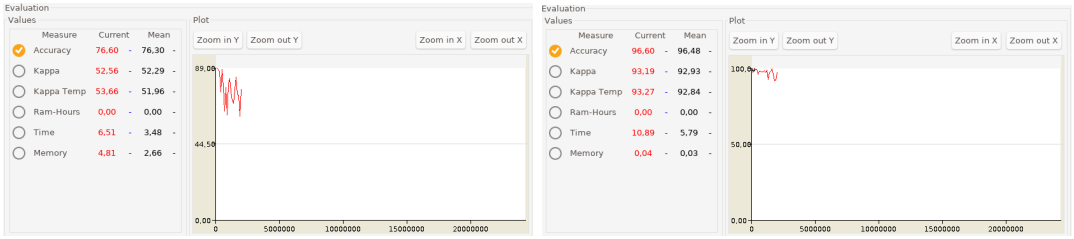
1.4. Ejercicio 4

Igual que el modelo anterior, pero utilizando la función Prequential en vez de la función TestThenTrain. El método de evaluación Prequential testea con cada ejemplo que se le va entrando y después reentrena el modelo; utiliza una ventana para ir olvidando ejemplos antiguos y centrarse en ejemplos nuevos. La sentencias necesarias para evaluar es la siguiente.

java -cp moa.jar moa.DoTask "EvaluatePrequential -l trees.HoeffdingTree -e (WindowClassificationPerformanceEvaluator -w 1000) -s (generators.RandomRBFGeneratorDrift -s 0.001 -k 3 -r 1 -i 1 -a 7 -n 3) -i 2000000"

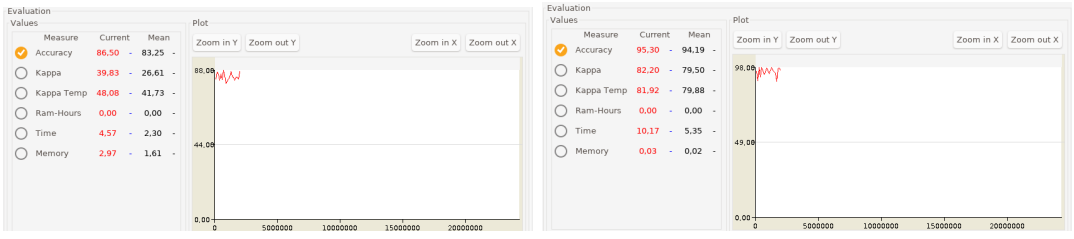
java -cp moa.jar moa.DoTask "EvaluatePrequential -l trees.HoeffdingAdaptiveTree -e (WindowClassificationPerformanceEvaluator -w 1000) -s (generators.RandomRBFGeneratorDrift -s 0.001 -k 3 -r 1 -i 1 -a 7 -n 3) -i 2000000"

Al igual que en el ejercicio anterior, se han hecho pruebas con 3 semillas diferentes. Los resultados son los siguientes.



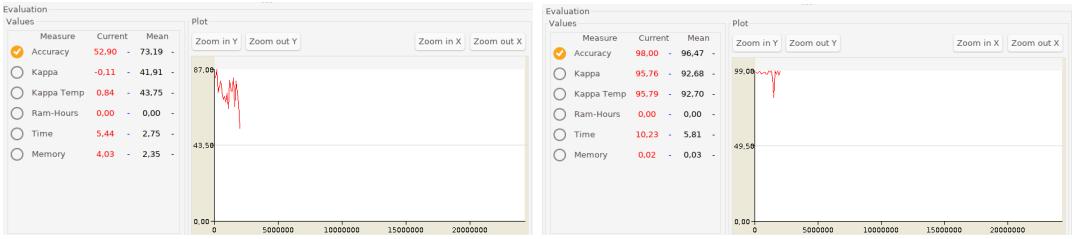
(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.6: Resultados Prequential con semilla 1



(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.7: Resultados Prequential con semilla 2



(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.8: Resultados Prequential con semilla 3

Realizando pruebas con ambos algoritmos, se puede ver una gran diferencia entre cada uno de los algoritmos. El clasificador *HoeffdingTree* sufre una gran bajada en la precisión cuando llega un cambio de concepto, y para cuando puede recuperarse vuelve a haber otro cambio de concepto; por lo que su rendimiento vuelve a bajar. Por otro lado, el clasificador *HoeffdingAdaptiveTree* al detectar los cambios de concepto, tiene pequeñas bajadas en la precisión de las cuales se recupera rápidamente. Por lo tanto para este tipo de flujo de datos el clasificador *HoeffdingAdaptiveTree* es mejor que *HoeffdingTree*.

1.5. Ejercicio 5

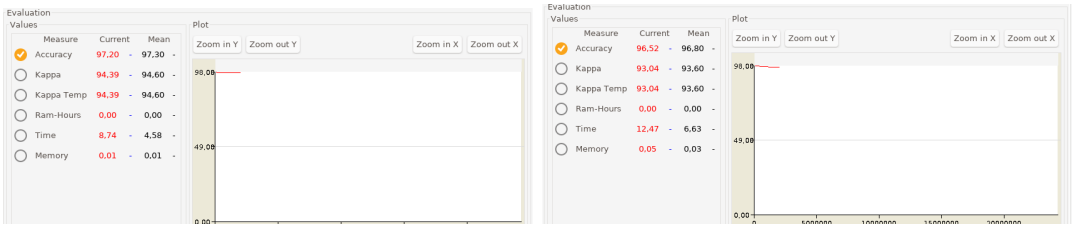
Repetir el ejercicio anterior cambiando el modelo con *SingleClassfierDrift*, utilizando el modelo *HoeffdingTree*. Este método, reemplaza el clasificador cuando se detecta un cambio de concepto. La sentencia necesaria para ejecutar la evaluación del modelo es la siguiente:

java -cp moa.jar moa.DoTask "EvaluateInterleavedTestThenTrain -l (drift.SingleClassifierDrift -l trees.HoeffdingTree) -s (generators.RandomRBFGeneratorDrift -s 0.001 -k 3 -r 1 -i 1 -a 7 -n 3) -i 2000000 -f 100000"

Para la modificación con el modelo HoeffdingAdaptiveTree habría que utilizar la siguiente sentencia: *java -cp moa.jar moa.DoTask "EvaluateInterleavedTestThenTrain -l (drift.SingleClassifierDrift -l trees.HoeffdingAdaptiveTree) -s (generators.RandomRBFGeneratorDrift -s 0.001 -k 3 -r 3 -i 3 -a 7 -n 3) -i 2000000 -f 100000"*

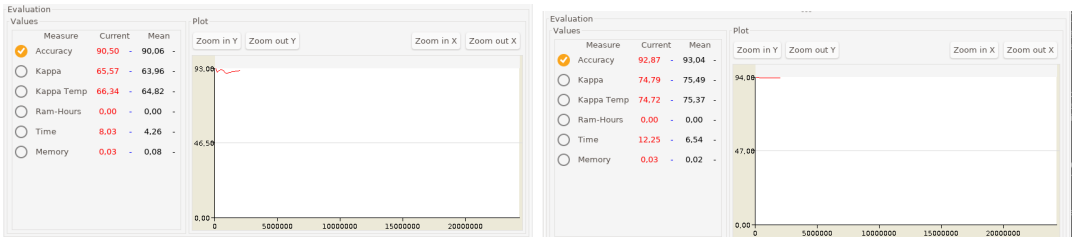
La única opción nueva con respecto al resto de ejercicios es la opción -l de la acción *SingleClassifierDrift* que especifica que modelo tiene que utilizar este método.

Al igual que en el ejercicio 3, se han ejecutado los algoritmos con tres semillas diferentes para el generador *RandomRBFGeneratorDrift*. Los resultados obtenidos por los clasificadores son los siguientes.



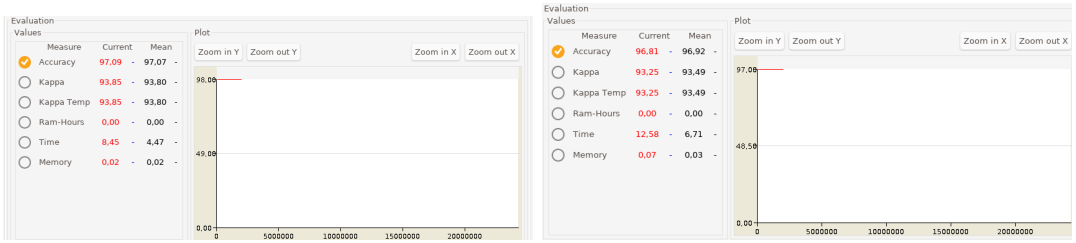
(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.9: Resultados SingleClassifierDrift con semilla 1



(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.10: Resultados SingleClassifierDrift con semilla 2



(a) HoeffdingTree (b) HoeffdingAdaptiveTree

Figura 1.11: Resultados SingleClassifierDrift con semilla 3

Como se puede ver en las gráficas obtenidas en la GUI de MOA, la precisión de ambos clasificadores es la misma. A diferencia del ejercicio 2.3, se utiliza un modelo *SingleClassifierDrift*; por ello, el clasificador se cambia cuando se detecta un cambio de concepto. Por lo tanto, cada vez que hay un cambio de concepto se crea un nuevo modelo, tanto para *HoeffdingTree* como para *HoeffdingAdaptiveTree* y entonces el modelo *HoeffdingAdaptiveTree* se comporta como el modelo *HoeffdingTree* ya que este no detecta el cambio de concepto. En cambio, en el ejercicio 4 y en el ejercicio 3 sí que hay una diferencia de rendimiento entre ambos clasificadores, ya que el clasificador *HoeffdingAdaptiveTree* puede detectar los cambios de concepto y el otro no. Como similitud en los tres ejercicios

está el tipo de flujo que se ha utilizado para comparar el rendimiento de los clasificadores, que siempre ha sido el generado por el generador *RandomRBFGeneratorDrift* con las mismas semillas y parámetros.

Capítulo 2

Parte Teórica

2.1. Preguntas tipo test

2.1.1. Pregunta 1

El aprendizaje incremental es útil cuando se quiere ganar eficiencia.

2.1.2. Pregunta 2

La minería de flujo de datos se considera cuando el problema genera datos continuamente.

2.1.3. Pregunta 3

La cota Hoeffding sirve para saber cuando hay suficientes datos para una estimación fiable

2.1.4. Pregunta 4

¿Qué características de clústers mantiene el algoritmo BIRCH? Suma lineal, suma cuadrática y número de objetos.

2.1.5. Pregunta 5

¿El algoritmo Stream maneja el concept drift? No.

2.1.6. Pregunta 6

¿Qué es el concept drift? Cambios en la dinámica del problema.

2.1.7. Pregunta 7

¿Cómo gestiona CVFDT el concept drift? Mantiene árboles alternativos.

2.1.8. Pregunta 8

¿Por qué es útil el ensemble learning en concept drift? Porque aprovecha la diversidad que se genera en los cambios.

2.1.9. Pregunta 9

¿Cuál es más eficiente entre DDM y ADWIN? Los dos son muy ineficientes.

2.1.10. Pregunta 10

¿Por qué es controvertida la clasificación en flujo de datos? Porque se requiere al oráculo por siempre.

2.1.11. Pregunta 11

¿Cómo gestiona ClueStream el concept drift? Mantiene información sobre el tiempo.

2.1.12. Pregunta 12

¿Por qué es complejo generar reglas de asociación en flujo de datos? Porque para calcular la confianza se requieren muchos datos.

2.2. Clasificación y experimentos

El problema que de clasificación dentro de la minería de flujo de datos trata de predecir la clase de unos datos que van llegando cada cierto tiempo y los algoritmos tratan de maximizar la precisión en la predicción de dichas clases. Para el ejemplo de esta práctica, el objetivo es también maximizar la precisión en la predicción de datos generados con diferentes generadores de flujo de datos, algunos con cambios de concepto, y analizar el comportamiento de dos clasificadores.

Los clasificadores utilizados en la práctica son *HoeffingTree* y *HoeffdingTree adaptativo*, el segundo es la versión adaptada a cambios de concepto del primero.

El clasificador *HoeffdingTree* es un tipo de árbol de decisión utilizados en aprendizaje incremental; estos modelos utilizan la desigualdad de Hoeffding para realizar las particiones del árbol, con esta medida se puede estimar si la media de una variable aleatoria no ha cambiado después de n instantes de tiempo; esto es interesante para problemas incrementales ya que el árbol realiza particiones para variable que no han cambiado durante el tiempo, por lo cual la partición que hace sobre dicha variable es fiable. Lo malo de este tipo de algoritmos es que puede necesitar muchos datos para poder realizar una partición sobre los datos, además este primer modelo no es capaz de detectar los cambios en la dinámica de los datos, por lo cual no es apto para problemas donde haya cambios de concepto (la gran mayoría en la realidad).

El clasificador *HoeffdingTree adaptativo* es la versión de *HoeffdingTree* capaz de identificar cambios de concepto. Este algoritmo guarda diferentes versiones del árbol, una vez se produce un cambio de concepto y el árbol actual deja de ser preciso en las predicciones, el algoritmo reemplaza los nodos necesarios para que el árbol se adapte a los nuevos datos. Para el caso de la práctica, el árbol utiliza el algoritmo *ADWIN*, este algoritmo utiliza una ventana principal sobre los datos y comprueba si existen dos ventanas suficientemente grandes y diferentes; si dichas ventanas existen, detecta el cambio de concepto en los datos. Una vez este algoritmo a detectado un cambio de concepto, el árbol modifica sus nodos para adaptarse a sus nuevos datos.

Dentro del desarrollo de la práctica se han utilizado tres métodos de evaluación, el primero de ellos es una evaluación estática de los modelos, el segundo mediante la estrategia *interleaved test-then-train* y el tercero mediante la estrategia *prequential*. La estrategia *interleaves test-then-train* utiliza primero los datos para testear el modelo, tras esto, re-entrena el modelo con los nuevos datos; la estrategia *prequential* utiliza la misma técnica, la diferencia entre estos dos es la forma en la que calculan la precisión de los clasificadores; en el caso de *interleaved test-then-train* utiliza todos los datos con los datos que se ha entrenado, para el caso de *prequential* utiliza una ventana de un tamaño fijo para utilizar solamente esos datos.

2.3. Concept Drift

El *concept drift* es un problema que se produce en los problemas de minería de flujo de datos, el *concept drift* se trata de una variación en la dinámica de los datos; este cambio de dinámica en los datos hace que los algoritmos clásicos tengan problemas en la predicción de datos una vez se produce este *concept drift*. Por ello, se debe crear nuevos modelos que sean capaces de detectar este cambio de concepto y adaptarse de forma rápida a los datos para que sean útiles en problema del mundo real. Este *concept drift* pueden darse de diferentes maneras, pueden ser de forma abrupta, gradual, incremental, etc... El *concept drift* puede deberse a diferentes razones, como por ejemplo ruido en los datos o variaciones en las características que inicialmente no habían sido contempladas por el modelo.

La primera solución al cambio de concepto se basan en algoritmos basados en aprendizaje incremental. Actualmente existen diferentes alternativas para manejar el *concept drift* en la minería de flujo de datos.

El primero de los enfoques es el aprendizaje online, este tipo de aprendizaje entrena continuamente los modelos mientras que van llegando datos, por ejemplo los *HoeffdingTree adaptativo*.

El segundo enfoque es el aprendizaje mediante ventanas; estos algoritmos se basan en que los datos más recientes tienen más importancia que los datos antiguos, para ello existen diferentes metodologías, como por ejemplo utilizar una ventana deslizantes sobre el conjunto de datos total o ponderar los datos según el tiempo de llegada.

Otro enfoque es el aprendizaje mediante modelos ensemble. Este tipo de enfoque es interesante porque el contexto se puede manejarse mediante la diversidad de los mode-

los que forman el ensemble. Dentro de este enfoque se utilizan variantes que utilizan predicción mediante votación o por ponderación.

La última de los enfoques es utilizar un algoritmo que detecte el *concept drift* y re-entrene el modelo que se está utilizando para realizar la predicción. Uno de los algoritmos que se encarga de esto es el algoritmo *DDM*, este se fija en la precisión del modelo para detectar el *concept drift*, si la precisión baja mucho durante un espacio de tiempo, se detecta el *concept drift*. Otro algoritmo utilizado para la detección del *concept drift* es *ADWIN*, que ya se ha comentado anteriormente. El problema de estos dos algoritmos es que son ineficientes y por lo tanto no sirven para problemas que manejan cantidades grandes de datos. Otro algoritmo para detección de *concept drift* es *HSP*, este es más eficiente que los dos anteriores comentados y no requiere monitorizar el algoritmo de predicción.

Por lo tanto, se puede ver que el *concept drift* es un problema que debe contemplarse en cualquier problema de minería de flujo de datos e implementarse para conseguir ofrecer buenos resultados tanto en clasificación como en cualquier otro problema de análisis de datos.