Big Analysis con Spark.

Néstor Rodríguez Vico - nrv23@correo.ugr.es

21 de mayo de 2019

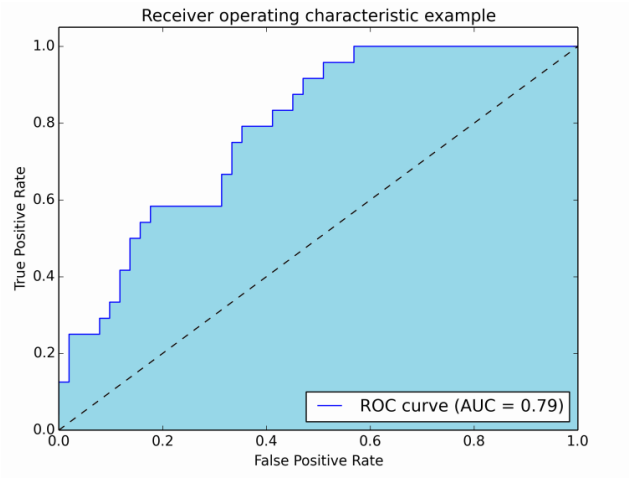
1. Introducción.

En esta práctica hemos probado diferentes algoritmos de preprocesamiento y diferentes clasificadores de *Spark* (https://spark.apache.org/) para abordar un problema de Big Data. Dichas prácticas se han ejecutado en un clúster proporcionado por la universidad con *216* núcleos y *696.0 GB* de memoria RAM.

2. Metodología de trabajo.

Para abordar la práctica, se ha usado la máquina virtual proporcionada por el profesorado. En dicha máquina, he desarrollado el código, compilado el mismo y ejecutado con los conjuntos de datos reducidos hasta asegurarme de que los algoritmos no producían ningún error descontrolado. Una vez tenía un código completamente ejecutable, lo he compilado, subido al *clúster* y ejecutado, guardando los resultados en un fichero de salida para poder revisarlos posteriormente.

Para comparar los algoritmos de clasificación y ver el rendimiento de los algoritmos de preprocesamiento aplicados he usado el valor de la métrica AUC (*área under the curve*). La representación de dicha métrica la podemos ver en la siguiente figura:



Dicha métrica representa el área bajo la curva que se obtiene al representar el ratio de verdaderos positivos frente al ratio de falsos positivos. A mayor valor de AUC, mejor es el modelo aprendido.

3. Análisis inicial.

Lo primero que he hecho es estudiar el balanceo y si hay valores perdidos en el conjunto de datos. En el conjunto de datos disponible en el clúster contamos con *900000* instancias asociada a la clase *0.0* y *100000* instancias asociada a la clase *1.0*, lo cual nos indica que nos enfrentamos a un problema bastante desbalanceado, con un ratio de desbalanceo $IR = 9.0$. Con respecto a los valores perdidos se ha comprobado que estos no existen, así que no tenemos que lidiar con algoritmos de preprocesamiento que apliquen imputación de valores perdidos.

4. Preprocesamiento.

En esta sección voy a comentar los distintos algoritmos aplicados de preprocesamiento. Cuando hable de los parámetros de los algoritmos, voy a marcar con un asterísco (*) los parámetros cuyos valores han sido obtenidos de forma empírica realizando varias pruebas, debido a la dificultad de hacer optimización de parámetros en un problema de estas dimensiones.

4.1. Desbalanceo.

Para corregir el desbalanceo se han implementado algoritmos dos algoritmos, *RandomOversampling* y *RandomUndersampling*. Las implementaciones propuestas se basan en las realizadas por *Sara Del Río García*, las cuales podemos encontrar en su GitHub.

4.2. Limpieza de ruido.

Para limpiar el ruido se han usado los siguientes algoritmos:

- *HME_BD*: implementado por *Diego García*. Dicha implementación se encuentra disponible en su repositorio NoiseFramework en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:
  - *data*: Conjunto de datos a procesar.
  - *(*) nTrees = 100*: Número de árboles a crear.
  - *(*) k = 4*: Número de particiones.
  - *(*) maxDepth = 10*: Profundidad máxima de los árboles.
  - *seed = 0*: Semilla para inicializar el generador de números aleatorios.
- *HTE_BD*: implementado por *Diego García*. Dicha implementación se encuentra disponible en su repositorio NoiseFramework en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:
  - *data*: Conjunto de datos a procesar.
  - *(*) nTrees = 100*: Número de árboles a crear.
  - *(*) partitions = 4*: Número de particiones.
  - *(*) vote = 0*: Estragia en el proceso de votación. Un cero indica *mayoría*.
  - *(*) partitions = 4*: Número de vecinos.
  - *(*) maxDepth = 10*: Profundidad máxima de los árboles.
  - *seed = 0*: Semilla para inicializar el generador de números aleatorios.

- *ENN_BD*: implementado por *Diego García*. Dicha implementación se encuentra disponible en su repositorio NoiseFramework en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:
  - *data*: Conjunto de datos a procesar.
  - *(*) k = 4*: Número de vecinos.
- *NCNEdit_BD*: implementado por *Diego García*. Dicha implementación se encuentra disponible en su repositorio SmartFiltering en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:
  - *data*: Conjunto de datos a procesar.
  - *(*) k = 4*: Número de vecinos.

También he probado *RNG_BD*, implementado por *Diego García*, disponible en su repositorio SmartFiltering en GitHub. Dicho algoritmo lanza una excepción *java.lang.OutOfMemoryError: Java heap space*, la cual no he podido solucionar y, por lo tanto, no he podido ejecutarlo.

4.3. Discretización.

Para discretizar los datos se ha usado el algoritmo *EqualWidthDiscretizer* implementado por *Diego García* el cual se encuentra disponible en su repositorio Equal-Width-Discretizer en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:

- *data*: Conjunto de datos a procesar.
- *(*) nBins = 25*: Número de *bins* a crear..

4.4. Selección de características.

Nos encontramos ante un dataset de un millón de instancias y dieciocho características. Como podemos ver, el cuello de botella del problema es el número de instancias, no el número de características. Aplicar un algoritmo de selección de características para un millón de instancias teniendo sólo dieciocho características no tiene mucho sentido, ya que vamos a tener que invertir mucho tiempo de cómputo en reducir la dimensionalidad, a pesar de que esta ya es pequeña.

Para probar lo comentado anteriormente, se ha usado un algoritmo de *PCA* implementado en la librería mllib. Este algotimo necesita ningún parámetro para funcionar, sólo el conjunto de datos a procesar y el número de componente principales a elegir, en mi caso he elegido *5*, para reducir la dimensionalidad a un *30 %* de la original.

4.5. Selección de instancias.

Para la selección de instancias se han usado los siguientes algoritmos:

- *FCNN_MR*: implementado por *Diego García*. Dicha implementación se encuentra disponible en su repositorio SmartReduction en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:
  - *data*: Conjunto de datos a procesar.

- *(*) k = 3*: Número de vecinos.

- *RMHC_MR*: implementado por *Diego García*. Dicha implementación se encuentra disponible en su repositorio SmartReduction en GitHub. Dicho algoritmo se ha ejecutado con los parámetros:

  - *data*: Conjunto de datos a procesar.
  - *(*) p = 0.1*: Porcentaje de instancias.
  - *(*) it = 100*: Número de iteraciones.
  - *(*) k = 3*: Número de vecinos.
  - *seed = 0*: Semilla para inicializar el generador de números aleatorios.

También he intentado probar el algoritmo *SSMASFLSDE_MR*, implementado por *Diego García*, disponible en su repositorio SmartReduction en GitHub. El prblema de dicho algoritmo es que se trata de un algoritmo evolutivo y consume mucho tiempo, así que he decidido no ejecutarlo, ya que tenemos otros algoritmos de selección de instancias que funcionan bastante bien y son más rápidos.

5. Clasificadores.

En esta sección voy a comentar los clasificadores usados en esta práctica. Cuando hable de los parámetros de los algoritmos, voy a marcar con un asterísco (*) los parámetros cuyos valores han sido obtenidos de forma empírica realizando varias pruebas, debido a la dificultad de hacer optimización de parámetros en un problema de estas dimensiones.

5.1. Random Forest.

El primer clasificador probado es un *Random Forest* estándar. El algoritmo usado ha sido *RandomForest*, disponible en la biblioteca *mllib*. El código usado para entrenar dicho clasificador es el siguiente:

```
RandomForest.trainClassifier(train, numClasses = 2,
        categoricalFeaturesInfo = Map[Int, Int](), numTrees = 80,
        featureSubsetStrategy = "auto", impurity = "gini", maxDepth = 8,
        maxBins = 32, seed = 0)
```

Explicación de los parámetros:

- *train*: Conjunto de entrenamiento.

- *numClasses*: Número de clases de nuestro problema.

- *categoricalFeaturesInfo*: Información sobre las variables categóricas. Nuestro problema no tiene variables categóricas, por eso pasamos un *Map* vacío.

- *(*) numTrees*: Número de árboles a generar.

- *featureSubsetStrategy*: Número de características a considerar para las divisiones en cada nodo. Si se usa *auto* el algoritmo determinará el mejor valor basándose en el valor de *numTrees*.

- *impurity*: Criterio utilizado para el cálculo de la ganancia de información. Se ha usado *gini* ya que es el recomendado por la documentación.

- *(*) maxDepth*: Profundida máxima del árbol.
- *(*) maxBins*: Número máximo de *bins* utilizadas para el particionamiento.
- *seed*: Semilla para inicializar el generador de números aleatorios.

5.2. SVMwithSGD.

A continuación se ha usado una *máquina de soporte vectorial* (*Support Vector Machine - SVM*) que hace uso del *gradiente descendente estocástico* (*Stochastic Gradient Descent - SGD*). El algoritmo usado ha sido *SVMWithSGD*, disponible en la biblioteca *mllib*. El código usado para entrenar dicho clasificador es el siguiente:

```
SVMWithSGD.train(train, numIterations = 100)
```

Explicación de los parámetros:

- *train*: Conjunto de entrenamiento.
- *(*) numIterations*: Número de iteraciones.

5.3. PCARD.

El algoritmo *PCARD* es un ensemble. El algoritmo usado ha sido *PCARD*, disponible en el repositorio de *GitHub* de *Diego García*. El código usado para entrenar dicho clasificador es el siguiente:

```
PCARD.train(train, nTrees = 100, cuts = 5)
```

Explicación de los parámetros:

- *train*: Conjunto de entrenamiento.
- *(*) numTrees*: Número de árboles a generar.
- *(*) cuts*: Número de umbrales por característica.

5.4. Decision Tree.

Este algoritmos es un árbol de decicisón clásico. El algoritmo usado ha sido *DecisionTree*, disponible en la biblioteca *mllib*. El código usado para entrenar dicho clasificador es el siguiente:

```
DecisionTree.trainClassifier(train, numClasses = 2,
        categoricalFeaturesInfo = Map[Int, Int](), impurity = "gini",
        maxDepth = 8, maxBins = 32)
```

Explicación de los parámetros:

- *train*: Conjunto de entrenamiento.
- *numClasses*: Número de clases de nuestro problema.
- *categoricalFeaturesInfo*: Información sobre las variables categóricas. Nuestro problema no tiene variables categóricas, por eso pasamos un *Map* vacío.
- *impurity*: Criterio utilizado para el cálculo de la ganancia de información. Se ha usado *gini* ya que es el recomendado por la documentación.

- *(*) maxDepth*: Profundida máxima del árbol.
- *(*) maxBins*: Número máximo de *bins* utilizadas para el particionamiento.

5.5. GradientBoostedTrees.

Este algoritmo se basa en el algoritmo *Stochastic Gradient Boosting* propuesto por *J.H. Friedman* en 1999. Dicho algoritmo aprende conjuntos de árboles minimizando la funcion de pérdida. El algoritmo usado ha sido *GradientBoostedTrees* disponible en la biblioteca *mllib*. El código usado para entrenar dicho clasificador es el siguiente:

```
val boostingStrategy = BoostingStrategy.defaultParams("Classification")
boostingStrategy.numIterations = 100
boostingStrategy.treeStrategy.numClasses = 2
boostingStrategy.treeStrategy.maxDepth = 5
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]()

GradientBoostedTrees.train(train, boostingStrategy)
```

Explicación de los parámetros:

- *(*) boostingStrategy.numIterations*: Número de iteraciones.
- *boostingStrategy.treeStrategy.numClasses*: Número de clases de nuestro problema.
- *(*) boostingStrategy.treeStrategy.maxDepth*: Profundida máxima del árbol.
- *boostingStrategy.treeStrategy.categoricalFeaturesInfo*: Información sobre las variables categóricas. Nuestro problema no tiene variables categóricas, por eso pasamos un *Map* vacío.
- *train*: conjunto de entrenamiento.

6. Experimentos.

En esta sección voy a detallar los experimentos realizados. Los parámetros usados en los experimentos han sido los indicados en las secciones anteriores si no se indica lo contrario.

6.1. Experimentos básicos.

Lo primero que he hecho ha sido aplicar todos los clasificadores a todas las técnicas de preprocesamiento de forma independiente, es decir, sin combinaciones entre ellos. Tras asplicar una técnica de preprocesamiento que modifica el ratio de desbalanceo (como puede ser *RandomOversampling*, *RandomUndersampling* o técnicas de selección de instancias), se ha estudiado como de balanceado ha quedado el nuevo conjunto. Los valores de la métrica *auc* obtenidos son los siguientes:

		Sampling		ClearNoise				Discretization	FeatureSelection	InstanceSelection	
model	**Original**	**Under**	**Over**	**HMEBD**	**HTEBD**	**ENNBD**	**NCNEdit**	**EWD**	**pca**	**FCNN**	**RMHC**
RF	0.790958	0.613495	1.000000	0.774220	0.811742	1.000000	0.805488	0.805488	0.761004	0.733423	0.796452
SVMwithSGD	0.832559	0.833361	0.750734	0.750833	0.748794	0.750481	0.687568	0.687568	0.805371	0.815443	0.829624
decisionTree	0.733296	0.610165	0.720580	0.759912	0.786172	0.738511	0.735937	0.735937	0.719260	0.700328	0.708826
pcard	0.507549	0.778233	0.500000	0.515958	0.500424	0.500000	0.508259	0.508259	0.500000	0.522451	0.509817
gbts	0.750616	0.621049	0.950001	0.767141	0.795334	1.000000	0.736438	0.736438	0.737500	0.720658	0.738069
Media	0.722995	0.691260	0.784263	0.713613	0.728493	0.797798	0.694738	0.694738	0.704627	0.698461	0.716558

A continuación, vamos a ver el número de instancias asociadas a cada clase tras aplicar los algoritmos de preprocesamiento:

	Clase		
Dataset	**1.0**	**0.0**	**IR**
Original	900000	100000	9,00000
RandomUndersampling	99344	100000	0,99344
RandomOversampling	900000	9017	99,81147
ClearNoise.HMEBD	895535	7347	121,89125
ClearNoise.HTEBD	898529	2545	353,05658
ClearNoise.ENNBD	832977	7422	112,23080
ClearNoise.NCNEdit	828959	31025	26,71907
Discretization.EWD	900000	100000	9,00000
FeatureSelection.pca	900000	100000	9,00000
InstanceSelection.FCNN	214812	89106	2,41075
InstanceSelection.RMHC	90014	9853	9,13569

Cómo podemos ver, sólo tras aplicar *RandomUndersampling* nos encontramos ante un dataset balanceado. En el caso de *RandomOversampling* se ha usado un ratio de *1.0*, lo cual ha producido el efecto inverso al deseado, se ha reducido el número de instancias de la clase *0.0* en vez de aumentarlo. También podemos ver como los algoritmos de selección de características y discretización no modifican el balanceo.

En cuanto a los resultados de los clasificadores, podemos ver como, en el caso de *Oversampling* y de *ENNBD* obtenemos un resultado de *1.0* (o muy cercano a *1.0*) en algunos de los clasificadores. También podemos ver que, excepto en *UnderSampling*, los valores obtenidos por el algoritmo *PCARD* son bastante malos. Esto no se debe a que el algoritmo sea malo, sino al desbalanceo que hay en el conjunto de datos.

6.2. Balancear el conjunto de datos.

Como hemos visto, balancear el conjunto de datos es algo imprescindible, ya que afecta bastante a los resultados obtenidos. Por eso, mi primera tarea para intentar combinar preprocesamientos es genera un conjunto de datos balanceado. Como hemos visto, *RandomUndersampling* ya nos lo proprociona. El problema de *RandomUndersampling* es que partimos de un conjunto en el que se han eliminado demasiadas instancias y sin ningún criterio, lo cual puede producir que hayamos perdido información relevante de nuestro problema. Es por ello que debemos centrarnos en generar un conjunto de datos balanceado con *RandomOversampling*. Para conseguir esto he jugado con el valor del parámetro que indica el ratio de instancias a generar, los resultados son los siguientes:

Ratio	**1.0**	**0.0**	**IR**	**Ratio**	**1.0**	**0.0**	**IR**	**Ratio**	**1.0**	**0.0**	**IR**
1.0	900000	9017	99,81147	**35.0**	900000	315832	2,84962	**70.0**	900000	631166	1,42593
5.0	900000	45113	19,94990	**40.0**	900000	360708	2,49509	**75.0**	900000	675943	1,33147
10.0	900000	90245	9,97285	**45.0**	900000	405522	2,21936	**80.0**	900000	721041	1,24820
15.0	900000	136034	6,61599	**50.0**	900000	450767	1,99660	**85.0**	900000	766189	1,17464
20.0	900000	181014	4,97199	**55.0**	900000	495750	1,81543	**90.0**	900000	811179	1,10950
25.0	900000	225521	3,99076	**60.0**	900000	541132	1,66318	**95.0**	900000	855972	1,05144
30.0	900000	270752	3,32408	**65.0**	900000	585522	1,53709	**100.0**	900000	900593	0,99934

Cómo podemos ver, el conjunto se va balanceando conforme aumentamos el ratio, llegando a un balanceo total con un valor de *100.0*. El problema de balancear el conjunto de datos es que aumentamos la cantidad de datos y, por lo tanto, los algoritmos de preprocesamiento y aprendizaje van a tardar más en ejecutar. Para ver sobre que conunto de datos balanceado debemos partir, voy

a ejecutar los distintos claisficadores comentados sobre el conjunto de datos con *RandomUndersampling*, el conjunto de datos con *RandomOversampling* y un ratio de *50.0* y el conjunto de datos con *RandomOversampling* y un ratio de *100.0*. Los resultados son los siguientes:

Modelo	Under	Over 50,0	Over 100,0
RF	0,613494	0,638766	0,614911
SVM	0,833360	0,843779	0,833520
decisionTree	0,610164	0,634775	0,612566
pcard	0,778413	0,747475	0,778778
gbts	0,621047	0,649388	0,621949
Media	0,691296	0,702837	0,692345

Podemos ver que, a diferencia de lo que pensabamos, los resultados obtenidos con *RandomUndersampling* son bastante buenos, similares a los obtenidos con *RandomOversampling*. Finalmente, podemos ver que tampoco hay una gran diferencia de usar un ratio de *50.0* o de *100.0*. Lo que si podemos ver, es que obtenemos valores coherentes en los algoritmos, ahora si obtenemos unos resultados interesantes para el algoritmo *PCARD* y no obtenemos valores de *auc* igual a *1.0*.

6.3. Combinaciones de preprocesamientos.

En esta sección voy a combinar distintos preprocesamientos explicados en las secciones anteriores.

6.3.1. Experimento 1.

La primera idea que se nos viene a la mente es balancear el conjunto de datos antes de empezar a trabajar con el. Es por eso que el primer paso de mi experimento es aplicar *RandomOverSampling* con un raiot de *50.0*, es decir, hacer que el número de instancias de la clase minoritaria sea igual al *50 %* del número de instancias de la clase mayoritaria. Tras aplicar dicho preprocesamiento, lo siquiente que se me ha ocurrido es remover instancias que son ruido. Para ello, he aplicado el algoritmo *ENNBD*. A continuación, he pensado que sería interesante reducir la dimensionalidad, es decir, aplicar algún método de selección de características, para eliminar las innecesarias. Para ello, he aplicado *PCA*. Finalmente, para eliminar las instancias que no son interesante, he aplicado un algoritmo de selección de instancias, como es *RMHC*. Tras aplicar estos preprocesamientos de forma secuencial, he entrenado los clasificadores comentados anteriormente y estos son los resultados:

Modelo	auc
RF	0.717247
SVM	0.805566
decisionTree	0.674830
pcard	0.504802
gbts	0.697342
Media	0.679957

Cómo podemos ver, los resultados no son del todo buenos a pesar de que la idea tiene bastante sentido. Veamos como se ha ido modificando el balanceo del conjunto de dato tras aplicar cada etapa del proprocesamiento conjunto:

Preprocesamiento	1.0	0.0	IR
Oversampling 50.0	900000	450767	1.996597
ClearNoise.ENNBD	671382	115057	5.835212
FeatureSelection.pca	671382	115057	5.835212
InstanceSelection.RMHC	66644	11323	5.885719

Cómo podemos ver, aunque hayamos balanceado el conjunto inicialmente, tras ir aplicando preprocesamientos se ha vuelto a desbalancear el conjunto de datos. Por lo tanto, los modelos son entrenados con un conjunto de datos no balanceado y, de ahí, los malos resultados.

6.3.2. Experimento 2.

Viendo el problema que ha surgido en el experimento anterior, vamos a darle la vuelta al experimento. Primero vamos a aplicar un filtro de ruido para borrar instancias ruidosas. Para ello, voy a usar el algoritmo *ENNBD*, que es el que mejor resultados genera. A continuación, vamos a aplicar un algoritmo de selección de instancias, en concreto *RMHC* para eliminar más aún las instancias que no son necesarias. Una vez tenemos un conjunto limpio, aplicamos un algoritmo de *RandomOversampling* con un ratio de *100.0* para tener el mismo número de instancias positivas que negativas. Antes de mostrar los resultados, voy a mostrar tabla de como se ha comportado el balanceo del conjunto de datos tras ir aplicando los algoritmos de preprocesamiento:

Preprocesamiento	1.0	0.0	IR
InstanceSelection.RMHC	90013	9854	9.134666
ClearNoise.ENNBD	84262	586	143.791808
Oversampling 100.0	84262	84334	0.999146

Como podemos ver, tras aplicar los dos algotimos el conjunto se desbalancea de forma drástica, pero tras aplicar *RandomOversampling* obtenemos un conjunto balanceado. Los resultados son los siguientes:

Modelo	auc
RF	0.619567
SVM	0.824579
decisionTree	0.605482
pcard	0.775172
gbts	0.626915
Media	0.690343

Cómo podemos ver, los resultados son mejores que los obtenidos en el experimento anterior y *PCARD* genera unos resultados con sentido.

6.3.3. Experimento 3.

Cómo podemos ver en los experimentos anteriores, aplicar algún algoritmo de selección de instancias o de filtrado de ruido afecta bastante al número de instancias disponibles para el conjunto de entrenamiento. En particular, los algoritmos de filtrado de ruido son los que más desbalancean el conjunto de datos. Es por ello que, en este experimento he ido un paso atrás y he aplicado sólo un algoritmo de selección de instancias (*RMHC*) seguido de un *RandomOversampling* con un ratio de

100.0 para balancear el conjunto. Antes de mostrar los resultados, voy a mostrar tabla de como se ha comportado el balanceo del conjunto de datos tras ir aplicando los algoritmos de preprocesamiento:

Preprocesamiento	1.0	0.0	IR
InstanceSelection.RMHC	90014	9853	9.135695
Oversampling100	90014	90662	0.992853

Los resultados son los siguientes:

Modelo	auc
RF	0.614138
SVM	0.833352
decisionTree	0.609949
pcard	0.777708
gbts	0.623109
Media	0.691651

6.3.4. Experimento 4.

Como hemos visto en los dos experimentos anteriores, la clase mayoritaria (*1.0*) se reduce drásticamente con los algoritmos de filtrado de ruido y selección de instancias, pero también sucede lo mismo con la clase minoritaria. En este experimento se ha aplicado el algoritmo *FCNN* de selección de instancias seguido del algoritmo *NCNEdit* de filtrado de ruido pero sólo en las instancias de la clase mayoritaria. Una vez aplicados ambos preprocesamientos, he juntado la salida con las instancias de la clase minoritaria y he aplicado *RandomOversampling* con un ratio *100.0* sobre la clase con menor instancias (que puede que no sea la clase minoritaria tras el proprocesamiento) para obtener un conjunto de datos balanceado. Este experimento no ha sido satisfactorio ya que se ha reducido demasiado el conjunto de datos y los resultados han sido malísimos.

Lo que más ha reducido el conjunto de datos ha sido aplicar el algoritmo *FCNN* de selcción de instancias, así que también he probado a quitar este paso del experimento pero los resultados han seguido siendo bastante malos.

7. Conclusiones.

En este trabajo he aprendido las facilidades que aporta trabajar con un *framework* como es *Spark* en ambitos de *Big Data*, ya que permite implementar algoritmos paralelos y distribuidos de una forma transparente al programador, permitiendo tener así una abstracción del *hardware* donde se van a ejecutar los algoritmos. Esto es una gran ventaja, ya que el código usado en esta práctica se ha podido ejecutar tanto en mi máquina local como en el *clúster* proporcionado por los profesores sin tener que realizar ningún ajuste al mismo.

También me he enfrentado a uno de los grande problemas comentados en clase cuando nos enfrentamos a problemas donde debemos aplicar técnicas de preprocesamiento, y es que es díficil encontrar el preprocesamiento correcto a usar. Como hemos visto en teoría, no hay unos pasos estandarizados a seguir que garanticen mejorar el conjunto de datos original, por ello debemos probar distintas combinaciones y preprocesamientos para intentar encontrar la mejor combinación para nuestro problema en particular.

8. Bibliografía.

- Spark: https://spark.apache.org/
- RandomOversampling y RandomUncersampling: https://github.com/saradelrio/Imb-sampling-ROS_and_RUS
- NoiseFramework: https://github.com/djgarcia/NoiseFramework
- SmartFiltering: https://github.com/djgarcia/SmartFiltering
- Equal-Width-Discretizer: https://github.com/djgarcia/Equal-Width-Discretizer
- SmartReduction: https://github.com/djgarcia/SmartReduction
- PCARD: https://github.com/djgarcia/PCARD
- Documentación de Spark para Scala: https://spark.apache.org/docs/2.2.0/api/scala/index.html