Big Data y Cloud Computing

Analítica de datos en Big Data

ugr | Universidad de Granada

Máster en Ciencia de Datos e Ingeniería de los Computadores

Besay Montesdeoca Santana besayms.39@gmail.com

Objetivos de la práctica

El objetivo de esta práctica es resolver el problema de clasificación no balanceada en el contexto de Big Data, es decir, se trabajará con una base de datos bastante grande que está alojada en la plataforma hadoop mediante HDFS. Para evaluar la calidad de las clasificaciones se utilizara como medida TPR x TNR en test (el producto de los porcentajes de clasificación en cada clase). Tenemos que resolver el problema utilizando la biblioteca MLLib para los algoritmos Decision Tree y Random Forest, y los algoritmos de preprocesamiento ROS y RUS.

Creación del proyecto Eclipse

El código de la práctica se ha implementado en el entorno de desarrollo Eclipse sobre el que se ha instalado la IDE de Scala. Después de crear el proyecto Scala se procedió a añadir las dependencias modificando el archivo pom.xml para añadir las librerías que necesitaremos en la práctica. Se añadieron mediante esta forma a parte de la librería MLLib, las librerías JMailloH:kNN_IS y sramirez:spark-MDLP-discretization para realizar tareas de preprocesamiento de selección de características y discretización con nuestra base de datos y así mejorar el rendimiento de los clasificadores.

Con respecto a la librería Imb-sampling-ROS_and_RUS para realizar el balanceado de las clases, tal como está implementado, cada vez que queremos probar una configuración de balanceo distinta para analizar su rendimiento con los clasificadores tenemos que lanzar una orden spark-submit que genera el dataset balanceado, el cual debemos leer nuevamente para trabajar con él en nuestro código. Como esto puede ser algo ineficiente se ha modificado ligeramente la estrategia de funcionamiento de la librería para permitir usarla desde codigo scala.

Una vez implementado el código que se desea probar, se procedía a crear el jar para luego subirlo al cluster y probarlo de la siguiente forma:

```
spark-submit --master spark://hadoop-master:7077 --class BigData --packages
JMailloH:kNN_IS:3.0,sramirez:spark-infotheoretic-feature-selection:1.3.1,sramirez:
spark-MDLP-discretization:1.2.1 --jars Imb-sampling-1.2.jar --executor-memory 40G
--total-executor-cores 50 BigData-0.0.1-SNAPSHOT.jar > result.txt
```

Pruebas y experimentos

Antes de entrar de lleno en las pruebas que se realizaron en la práctica, se explicarán cómo se implementaron algunas partes fundamentales para luego ejecutar los experimentos.

En primer lugar debemos leer los datasets que están almacenados de forma distribuida mediante HDFS. Seguidamente después los transformamos a RDD[LaberedPoint] para trabajar de forma más fácil y clara con las bases de datos:

```scala
// Leemos los datasets
val sc = new SparkContext(conf)
val trainData = sc.textFile("hdfs://hadoop-master/user/spark/datasets/ECBDL14_mbd/ecbdl14tra.data")
val testData = sc.textFile("hdfs://hadoop-master/user/spark/datasets/ECBDL14_mbd/ecbdl14tst.data")
val parser = new KeelParser(sc, "hdfs://hadoop-master/user/spark/datasets/ECBDL14_mbd/ecbdl14.header")
```

```
// Transformamos los datos de RDD[String] RDD[Laberedpoint] para trabajar de forma más facil
val trainDataLabered = trainData.map{x => parser.parserToLabeledPoint(x)}
val testDataLabered = testData.map{x => parser.parserToLabeledPoint(x)}
```

En segundo lugar se mostrará cómo se realiza la discretización de la base de datos, para ello utilizamos la librería creada por sramirez:

```
// Discretizamos la base de datos para poder luego extraer características
val categoricalFeat: Option[Seq[Int]] = None
val nBins = 25
val maxByPart = 10000
val discretizer = MDLPDiscretizer.train(trainDataLabered, categoricalFeat, nBins, maxByPart)

val trainDiscrete = trainDataLabered.map(i => LabeledPoint(i.label,
discretizer.transform(i.features)))
val testDiscrete = testDataLabered.map(i => LabeledPoint(i.label, discretizer.transform(i.features)))
```

Seguidamente una vez discretizada la base de datos, se procede a realizar una selección de características, en este caso se ha optado por quedarnos con las 50 mejores características:

```
// Realizamos la seleccion de caracteristicas
val featureSelector = new InfoThSelector(new InfoThCriterionFactory("mrmr"),50,6).fit(trainDiscrete)
val trainSelection = trainDiscrete.map(i => LabeledPoint(i.label,
featureSelector.transform(i.features))).cache()
val testSelection = testDiscrete.map(i => LabeledPoint(i.label,
featureSelector.transform(i.features))).cache()
```

Por último quedaría mostrar cómo se realiza el entrenamiento de los árboles. Para la ejecución de los mismos se han implementado dos funciones a las que se les pasa los parámetros necesarios para el entrenamiento de los árboles. En la misma función se realiza la predicción y se calcula las medidas TPR y TNR para su posterior análisis:

```
def runDecisionTree(train:RDD[LabeledPoint], test:RDD[LabeledPoint], mDepth:Int, mBins:Int){
 val numClasses = 2
 val categoricalFeaturesInfo = Map[Int, Int]()
 val impurity = "entropy"
 val maxDepth = mDepth
 val maxBins = mBins

 val model = DecisionTree.trainClassifier(train, numClasses, categoricalFeaturesInfo,
   impurity, maxDepth, maxBins)

 val predictions = test.map { point =>
   val prediction = model.predict(point.features)
   (prediction, point.label)
 }

 val metrics = new MulticlassMetrics(predictions)
 val precision = metrics.precision
 val cm = metrics.confusionMatrix
 println("\nDecisionTree maxDepth = " + mDepth + " maxBins = " + mBins)
 println(cm.toString())

 val tpr = cm(0,0)/(cm(0,0) + cm(0,1))
 val tnr = cm(1,1)/(cm(1,0) + cm(1,1))
```

```
 println("\nTPR = " + tpr + " -- " + "TNR = " + tnr)
 println("TPR*TNR = " + tpr*tnr)
 println("----------------------------------------------------------------")
}

def runRandomForest(train:RDD[LabeledPoint], test:RDD[LabeledPoint], nTree:Int, mDepth:Int,
mBins:Int){
   val numClasses = 2
   val categoricalFeaturesInfo = Map[Int, Int]()
   val numTrees = nTree // Use more in practice.
   val featureSubsetStrategy = "auto" // Let the algorithm choose.
   val impurity = "gini"
   val maxDepth = mDepth
   val maxBins = mBins

   val model = RandomForest.trainClassifier(train, numClasses, categoricalFeaturesInfo,
     numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

   val predictions = test.map { point =>
     val prediction = model.predict(point.features)
     (prediction, point.label)
   }

   val metrics = new MulticlassMetrics(predictions)
   val precision = metrics.precision
   val cm = metrics.confusionMatrix
   println("\nRamdonForest numTree = " + nTree + " maxDepth = " + mDepth + " maxBins = " + mBins)
   println(cm.toString())

   val tpr = cm(0,0)/(cm(0,0) + cm(0,1))
   val tnr = cm(1,1)/(cm(1,0) + cm(1,1))

   println("\nTPR = " + tpr + " -- " + "TNR = " + tnr)
   println("TPR*TNR = " + tpr*tnr)
   println("----------------------------------------------------------------")
 }
}
```

Una vez expuestas las secciones de código relevantes, se procederá a explicar la naturaleza los experimentos.

En primer lugar se pretendió analizar como afecta las distintas técnicas de preprocesamiento a la clasificación de los dos árboles. Para ello se ejecutó un mismo árbol sencillo pero con distintas versiones de la base de datos.

La primera prueba seria entrenar los árboles con los datos en bruto sin preprocesar. Seguidamente entrenamos el mismo árboles con distintas configuraciones de balanceo RUS y ROS, y por último se realizó las mismas ejecuciones que las anteriores pero con la base de datos discretizada y con la selección de características.

Estas pruebas como se ha especificado antes se realizaron para los algoritmos Decision Tree y para Random Forest y cuyo codigo es el siguiente:

```
// Dataset de pruebas
val dataNoPrep = trainDataLabered
val dataNoPrepRus = runRUS.apply(trainDataLabered, 1.0, 0.0)
```

```
val dataNoPrepRos50 = runROS.apply(trainDataLabered, 1.0, 0.0, 50)
val dataNoPrepRos100 = runROS.apply(trainDataLabered, 1.0, 0.0, 100)
val trainDataDisSel = trainSelection
val trainDataDisSelRus = runRUS.apply(trainSelection, 1.0, 0.0)
val trainDataDisSelRos50 = runROS.apply(trainSelection, 1.0, 0.0, 50)
val trainDataDisSelRos100 = runROS.apply(trainSelection, 1.0, 0.0, 100)

println("DecisionTree - Sin preprocesamiento")
runDecisionTree(dataNoPrep, testDataLabered, 10, 32)
println("DecisionTree - Sin preprocesamiento balanceo RUS")
runDecisionTree(dataNoPrepRus, testDataLabered, 10, 32)
println("DecisionTree - Sin preprocesamiento balanceo ROS 50%")
runDecisionTree(dataNoPrepRos50, testDataLabered, 10, 32)
println("DecisionTree - Sin preprocesamiento balanceo ROS 100%")
runDecisionTree(dataNoPrepRos100, testDataLabered, 10, 32)
println("DecisionTree - Discretización y Selec. Caract 50")
runDecisionTree(trainDataDisSel, testSelection, 10, 32)
println("DecisionTree - Discretización y Selec. Caract 50 balanceo RUS")
runDecisionTree(trainDataDisSelRus, testSelection, 10, 32)
println("DecisionTree - Discretización y Selec. Caract 50 balanceo ROS 50%")
runDecisionTree(trainDataDisSelRos50, testSelection, 10, 32)
println("DecisionTree - Discretización y Selec. Caract 50 balanceo ROS 100%")
runDecisionTree(trainDataDisSelRos100, testSelection, 10, 32)

println("RandomForest - Sin preprocesamiento")
runRandomForest(dataNoPrep, testDataLabered, 10, 4, 32)
println("RandomForest - Sin preprocesamiento balanceo RUS")
runRandomForest(dataNoPrepRus, testDataLabered, 10, 4, 32)
println("RandomForest - Sin preprocesamiento balanceo ROS 50%")
runRandomForest(dataNoPrepRos50, testDataLabered, 10, 4, 32)
println("RandomForest - Sin preprocesamiento balanceo ROS 100%")
runRandomForest(dataNoPrepRos100, testDataLabered, 10, 4, 32)
println("RandomForest - Discretización y Selec. Caract 50")
runRandomForest(trainDataDisSel, testSelection, 10, 4, 32)
println("RandomForest - Discretización y Selec. Caract 50 balanceo RUS")
runRandomForest(trainDataDisSelRus, testSelection, 10, 4, 32)
println("RandomForest - Discretización y Selec. Caract 50 balanceo ROS 50%")
runRandomForest(trainDataDisSelRos50, testSelection, 10, 4, 32)
println("RandomForest - Discretización y Selec. Caract 50 balanceo ROS 100%")
runRandomForest(trainDataDisSelRos100, testSelection, 10, 4, 32)
```

Los resultados están representados en las siguientes tablas:

Prueba - Decision Tree	TPR	TNR	TPR*TNR
Datos balanceados con ROS 100%	0.7057	0.7329	0.5172
Datos balanceados con RUS	0.7158	0.7266	0.5202
Datos balanceados con ROS 50%	0.8699	0.5081	0.4420
Datos balanceados con ROS 100%	0.7057	0.7329	0.5172
Datos discretizados y con selección de características (50)	0.8816	0.4860	0.4285
Datos discretizados y con selección de	0.7396	0.6910	0.5111

características (50) con RUS			
Datos discretizados y con selección de características (50) con ROS 50%	0.8638	0.5212	0.4502
Datos discretizados y con selección de características (50) con ROS 100%	0.7138	0.7184	0.5128

Prueba - Random Forest	TPR	TNR	TPR*TNR
Sin ningún preprocesamiento	0.9799	0.1314	0.1288
Datos balanceados con RUS	0.6667	0.7181	0.4788
Datos balanceados con ROS 50%	0.9479	0.2508	0.2378
Datos balanceados con ROS 100%	0.6552	0.7269	0.4763
Datos discretizados y con selección de características (50)	0.9527	0.2315	0.2205
Datos discretizados y con selección de características (50) con RUS	0.6174	0.7631	0.4712
Datos discretizados y con selección de características (50) con ROS 50%	0.9183	0.3387	0.3110
Datos discretizados y con selección de características (50) con ROS 100%	0.6683	0.7287	0.4870

Observando los resultados, ambos algoritmos tienen comportamientos idénticos. Echando un primer vistazo observamos cómo influye la presencia de alguna técnica de preprocesamiento en la calidad de clasificación de ambos árboles. Incluso cuando no se aplica ninguna técnica de balanceo pero si una discretización y una selección de características observamos un incremento en el valor de TPR*TNR. Queda claro que los mejores resultados se obtienen cuando combinamos una técnica de balanceo, en este caso Random Oversampling con 100%, junto con las técnicas de discretización y selección de características.

Una vez determinada la combinación de técnicas de preprocesamiento adecuadas para los dos algoritmos, se procedió a determinar una configuración de sus parámetros más o menos óptima. Para ello se ejecutaron los árboles(con el preprocesamiento anteriormente identificado) probando distintos valores de sus parámetros:

```
// Prueba con los paremetros del algoritmo DecisionTree
val testmDepth = Array(5, 10, 20, 30)
val testmBins = Array(5, 20, 50, 100)
for(mD <- testmDepth; mB <- testmBins) runDecisionTree(trainDataDisSelRos100, testSelection, mD, mB
```

```
// Prueba con los paremetros del algoritmo RandomForest
val testRFnTree = Array(5, 10, 30, 50)
val testRFmDepth = Array(4, 10, 20, 30)
for(nT <- testRFnTree; mD <- testRFmDepth) runRandomForest(trainDataDisSelRos100, testSelection,nT,
mD, 32)
```

Pruebas con los parámetros	TPR	TNR	TPR*TNR
DecisionTree maxDepth = 5 maxBins = 5	0.6854	0.7238	0.4961
DecisionTree maxDepth = 5 maxBins = 20	0.6822	0.7232	0.4934
DecisionTree maxDepth = 5 maxBins = 50	0.6890	0.7185	0.4951
DecisionTree maxDepth = 5 maxBins = 100	0.6890	0.7185	0.4951
DecisionTree maxDepth = 10 maxBins = 5	0.7252	0.7015	0.5088
DecisionTree maxDepth = 10 maxBins = 20	0.7068	0.7243	0.5120
DecisionTree maxDepth = 10 maxBins = 50	0.7138	0.7184	0.5128
DecisionTree maxDepth = 10 maxBins = 100	0.7138	0.7184	0.5128
DecisionTree maxDepth = 20 maxBins = 5	0.7364	0.5998	0.4417
DecisionTree maxDepth = 20 maxBins = 20	0.7394	0.6205	0.4588
DecisionTree maxDepth = 20 maxBins = 50	0.7359	0.6307	0.4642
DecisionTree maxDepth = 20 maxBins = 100	0.7359	0.6307	0.4642
DecisionTree maxDepth = 30 maxBins = 5	0.7785	0.4917	0.3828
DecisionTree maxDepth = 30 maxBins = 20	0.7803	0.5132	0.4004
DecisionTree maxDepth = 30 maxBins = 50	0.7794	0.5163	0.4024
DecisionTree maxDepth = 30 maxBins = 100	0.7794	0.5163	0.4024

Pruebas con los parámetros	TPR	TNR	TPR*TNR
RamdonForest numTree = 5 maxDepth = 4 maxBins = 32	0.6178	0.7390	0.4566
RamdonForest numTree = 5 maxDepth = 10 maxBins = 32	0.7130	0.7113	0.5072
RamdonForest numTree = 5 maxDepth = 20 maxBins = 32	0.7699	0.6435	0.4955
RamdonForest numTree = 5 maxDepth = 30	0.8200	0.5406	0.4433

maxBins = 32			
RamdonForest numTree = 10 maxDepth = 4 maxBins = 32	0.6427	0.7060	0.4538
RamdonForest numTree = 10 maxDepth = 10 maxBins = 32	0.6967	0.7310	0.5093
RamdonForest numTree = 10 maxDepth = 20 maxBins = 32	0.7486	0.6948	0.5201
RamdonForest numTree = 10 maxDepth = 30 maxBins = 32	0.8066	0.6082	0.4906
RamdonForest numTree = 30 maxDepth = 4 maxBins = 32	0.6822	0.7178	0.4897
RamdonForest numTree = 30 maxDepth = 10 maxBins = 32	0.7104	0.7221	0.5130
RamdonForest numTree = 30 maxDepth = 20 maxBins = 32	0.7805	0.6765	0.5280
RamdonForest numTree = 30 maxDepth = 30 maxBins = 32	0.8569	0.5772	0.4947
RamdonForest numTree = 50 maxDepth = 4 maxBins = 32	0.6834	0.7232	0.4942
RamdonForest numTree = 50 maxDepth = 10 maxBins = 32	0.7141	0.7188	0.5133
RamdonForest numTree = 50 maxDepth = 20 maxBins = 32	0.7855	0.6730	0.5286

Según los resultados, parece ser que en el caso del árbol de decisión los parámetros que mejor resultados ofrecen son aquellos con niveles más o menos bajos de máximo de profundidad(maxDepth) y un máximo de Bins elevado.

En el caso del algoritmo RandomForest, la configuración de árboles que mejor resultados ofrece es aquella donde hay un número elevado de numTree junto con una profundidad máxima también ligeramente alta.