Istanbul Technical University
Faculty of Computer and Informatics

BLG440E Computer Project2

Machine Learning for Network Intrusion Detection

Group No: 17

İrem Ertürk 150140725

Cem Yusuf Aydoğdu 150120251

1. Implementation Details

We implement project with Scala programming language by using Apache Spark machine learning library 'mllib'. We use Eclipse Scala IDE as implementation paltform.We aim to create accurate decision tree models by identifying data with three different classification approaches which clasify the network records in 2, 5 and 23 classes for either unbalanced and balanced data sets.

Preprocessing and Feature Extraction

At the beginning, we have KDD99 data which is in comma seperated value (csv) format. However, we cannot use csv format of data in decision tree classification. That's why need to preprocess the data before starting training and testing stages.

1. We take data from the file as RDD[String] format

```
val textRDD = sc.textFile("kddcup.data_10_percent_corrected")
```

2. Use parseNetworkRecord to model the data as RDD[[NetworkRecord] and then model as DataFrame , we describe the csv format and the type of each feature in Network Record.

```
val parseNetworkRecord = (str: String) => {
       var line = str.split(",")
NetworkRecord(line(0).toInt, line(1), line(2), line(3), line(4).toInt,
            line(5).toInt, line(6).toInt, line(7).toInt, line(8).toInt,
            line(9).toInt, line(10).toInt, line(11).toInt, line(12).toInt,
            line(13).toInt, line(14).toInt, line(15).toInt, line(16).toInt,
            line(17).toInt, line(18).toInt,line(19).toInt, line(20).toInt,
            line(21).toInt, line(22).toInt, line(23).toInt,line(24).toDouble,
            line(25).toDouble, line(26).toDouble, line(27).toDouble,
            line(28).toDouble,line(29).toDouble, line(30).toDouble, line(31).toInt,
            line(32).toInt,line(33).toDouble, line(34).toDouble, line(35).toDouble,
            line(36).toDouble, line(37).toDouble, line(38).toDouble,
            line(39).toDouble, line(40).toDouble, line(41)) }
val textRDD = sc.textFile("kddcup.data_10_percent_corrected") //("deneme.csv")
NetworkRecordRDD = textRDD.map(parseNetworkRecord).cache()
NetworkRecordDF = NetworkRecordRDD.toDF()
}
```

3. In that point our data frame contains features in different types; String, Int and Double . We need to apply feature extraction on String type features. We have four features which are protocol type, service, flag and label. However, rather than mapping label here, we write three functions for mapping label (`to2classes`, `to5classes`, `to23classes`). Because label mapping changes depending on the classification type and must be dynamic.

```
var protocol_typeMap: Map[String, Int] = Map()
var index: Int = 0
NetworkRecordRDD.map(NetworkRecord =>
NetworkRecord.protocol_type).distinct.collect.foreach(x => { protocol_typeMap += (x
                                        -> index); index += 1 })
var serviceMap: Map[String, Int] = Map()
var index1: Int = 0
NetworkRecordRDD.map(NetworkRecord =>
NetworkRecord.service).distinct.collect.foreach(x => { serviceMap += (x -> index1);
                                        index1 += 1 })
var flagMap: Map[String, Int] = Map()
var index2: Int = 0
NetworkRecordRDD.map(NetworkRecord =>
NetworkRecord.flag).distinct.collect.foreach(x => { flagMap += (x -> index2);
                                        index2 += 1 })
```

Above code maps String type features to Int , that mapping aims to use our decision treeclassification.

4. To get much more aggreable format we convert all fourty two features as an array of double elements in RDD[Array[Double]] format. In that format label representation is changed depending on the classification type by using `to2classes, to5classes, to23classes functions.`

5. As the last step of feature extrastion and preprocessing we map whole featues to label. 41Th array element represents the label and all others are the features of rows.

```
mldata = mlprep.map(x => LabeledPoint(x(41),
Vectors.dense(x(0), x(1), x(2), x(3), x(4), x(5), x(6), x(7), x(8), x(9), x(10),
x(11), x(12), x(13), x(14), x(15), x(16), x(17), x(18), x(19), x(20), x(21), x(22),
x(23), x(24), x(25), x(26), x(27), x(28), x(29), x(30), x(31), x(32), x(33), x(34),
x(35), x(36), x(37), x(38), x(39), x(40))))
```

Training the Decision Tree

After preprocessing and feature extraction stage our data becomes suitable for training decision tree and creating tree models . In that point we split data with two different approaches by using balanceData and unbalanceData functions. First one seperates the data balancedly by considering labels of each record. In that approach, data is seperated by selecting ninety percent of each label as traning split and ten percent of each label as testing split. In that approach our decision tree balancely learn the attack and non-attack conditions. On the otherhand, the second approach realize the unbalanced data splits which is derived from balance split. It is unbalance because, it takes all non-attack records in traning tree without any testing split and split the attack type records ninety percent for traning and ten percent for testing again. In the second approach it is obvious that the traning split consist of much more non-attack records and that cause misprediction in testing phase due to lack of training on attack types.

```
  def balanceData(sc: SparkContext, mldata: RDD[LabeledPoint], classNumber: Int):
Array[RDD[LabeledPoint]] = {
    var mlBalanced: RDD[LabeledPoint] = sc.emptyRDD
    var mlBalancedTest: RDD[LabeledPoint] = sc.emptyRDD
    for (i <- 0 to classNumber) {
      mlBalanced = mlBalanced ++ mldata.filter(x => x.label == i).randomSplit(Array(0.9,
0.10))(1)
      mlBalancedTest = mlBalancedTest ++ mldata.filter(x => x.label ==
i).randomSplit(Array(0.9, 0.10))(0)
    }
    return Array(mlBalanced, mlBalancedTest)
  }


  def unbalanceData(sc: SparkContext, mldata: RDD[LabeledPoint], classNumber: Int):
Array[RDD[LabeledPoint]] = {
    var mlUnBalanced: RDD[LabeledPoint] = sc.emptyRDD
    var mlUnBalancedTest: RDD[LabeledPoint] = sc.emptyRDD
    for (i <- 0 to classNumber) {
      if (i == 0)
        mlUnBalanced = mlUnBalanced ++ mldata.filter(x => x.label == i)
      else {
        mlUnBalanced = mlUnBalanced ++ mldata.filter(x => x.label ==
i).randomSplit(Array(0.9, 0.10))(1)
        mlUnBalancedTest = mlUnBalancedTest ++ mldata.filter(x => x.label ==
i).randomSplit(Array(0.9, 0.10))(0)
      }
    }
    return Array(mlUnBalanced, mlUnBalancedTest)
  }


  def trainTree(sc: SparkContext) = {
```

```
    var mldata1 = balanceData(sc, mldata, numClasses)
    if (train_type.equalsIgnoreCase("balanced")) {
      trainingData = mldata1(0)
      testData = mldata1(1)
    } else {
      var mldata2 = mldata1(0) ++ mldata1(1)
      val splits = unbalanceData(sc, mldata2, numClasses)
      //val splits = mldata.randomSplit(Array(0.9, 0.1))
      trainingData = splits(0)
      testData = splits(1)
    }

    treeModel = DecisionTree.trainClassifier(trainingData, numClasses,
categoricalFeaturesInfo, impurity, maxDepth, maxBins)
    println(treeModel.toDebugString)
    //Save treeModel as parquet
    val pathname = "C" + numClasses + "T" + train_type + "D" + maxDepth + "B" + maxBins
    treeModel.save(sc, pathname)
  }
```

Above function calls balanceData or unbalanceData depending on selection and creates the decision tree model. In visualization phase we will use that model to sketch the decision tree by D3.js

Testing

In the testing phase, we test the test split and compare the final classification result with original label then we count the wrong predictions to calculate accuracy of decision tree . And also by the help of MultiClassMetrics object of Spark we check the confusion matrix and interpret the matrix interms of true-positive, tru-negative, false-negative, false-positive ratios. Addition to these we also measeure the trainig and testing time for comparing the accuracy and complexity. All threse measurements are done for both unbalanced and balanced datasets and for each 2,5 and 23 class for different max bin, max depth values.

Virsualization of Decision Tree

At the last stage we visualize our decision tree models by using 'spay-json' library and D3.js which is browser based decision tree visualization tool.

```
  def treeModelToJson(sc: SparkContext, model_path: String, out_json_path: String) = {
    val model = DecisionTreeModel.load(sc, model_path)
    println(model.toDebugString)
    println(model.topNode.toString())
    println(model.topNode.leftNode.toString())
    val node = model.topNode.split
    println(node.toString())

    object NodeJsonProtocol extends DefaultJsonProtocol {
      implicit object NodeJsonFormat extends RootJsonFormat[Node] {
        def write(c: Node) = {
          var parent_name = ""
          c.split match {
            case Some(split) =>
              JsObject("name" -> JsString("feature: " + split.feature.toString() + " " +
               split.threshold.toString()),"parent" -> JsString(parent_name),"children" ->
               JsArray(c.leftNode.toJson, c.rightNode.toJson))
            case None =>
              JsObject("name" -> JsString("class " + c.predict.predict))
          }
        }
        def read(value: JsValue) = {
          throw new DeserializationException("Expected")
        }}}
    import NodeJsonProtocol._
    val writer = new PrintWriter(new File(out_json_path))
    writer.write(model.topNode.toJson.toString())
    writer.close()
    model  }
```

2. Results

Results are measured and compared by considering :

- Impurity Type{Gini, Entropy}
- Max Bin {100,1000,1000}
- # Class {2, 5, 23}
- Max Depth{5,10,30}

2.1. Balanced

IMPURITY TYPE							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"			87252.0 51.0 15.0 37.0 33.0 49.0 352473.0 0.0 4.0 31.0 19.0 7.0 15.0 4.0 0.0 144.0 8.0 7.0 842.0 2.0				
Max Depth "10"							
Max Bin "1000"							
# Class "5"	0.9982391753133728	0.0017608246866271775	359.0 12.0 0.0 1.0 3313.0	15636	12739	73	10
Impurity Type "Entropy"			87216.0 51.0 18.0 53.0 24.0 25.0 352329.0 0.0 2.0 33.0 24.0 2.0 21.0 2.0 1.0 73.0 0.0 3.0 925.0 5.0				
Max Depth "10"							
Max Bin "1000"							
# Class "5"	0.9991406596224751	8.593403775248747E-4	36.0 30.0 0.0 0.0 3654.0	15529	12280	119	10
MAX BIN							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"			87221.0 50.0 8.0 41.0 140.0 74.0 352449.0 0.0 20.0 11.0 28.0 1.0 14.0 1.0 0.0 182.0 0.0 2.0 833.0 1.0				
Max Depth "10"							
Max Bin "100"							
# Class "5"	0.9981811069	0.0018188931057431157	246.0 2.0 0.0 2.0 3450.0	10974	12733	83	10
Impurity Type "Gini"			87252.0 51.0 15.0 37.0 33.0 49.0 352473.0 0.0 4.0 31.0 19.0 7.0 15.0 4.0 0.0 144.0 8.0 7.0 842.0 2.0				
Max Depth "10"							
Max Bin "1000"							
# Class "5"	0.9982391753133728	0.0017608246866271775	359.0 12.0 0.0 1.0 3313.0	15636	12739	73	10
Impurity Type "Gini"			87385.0 17.0 0.0 28.0 135.0 105.0 352175.0 0.0 4.0 23.0 41.0 0.0 0.0 9.0 1.0 161.0 5.0 0.0 837.0 0.0				
Max Depth "10"							
Max Bin "10000"							
# Class "5"	0.998335612494602	0.0016643875053980136	204.0 3.0 0.0 4.0 3471.0	15817	13911	67	10
# CLASS							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"							
Max Depth "10"							
Max Bin "1000"			87608.0 130.0				
# Class "2"	0.9989388322	0.0010611678	342.0 356713.0	11509	5793	79	10
Impurity Type "Gini"			87252.0 51.0 15.0 37.0 33.0 49.0 352473.0 0.0 4.0 31.0 19.0 7.0 15.0 4.0 0.0 144.0 8.0 7.0 842.0 2.0				
Max Depth "10"							
Max Bin "1000"							
# Class "5"	0.9982391753133728	0.0017608246866271775	359.0 12.0 0.0 1.0 3313.0	15636	12739	73	10
Impurity Type "Gini"			31.0 0.0 0.0 0.0 0.0 0.0 0.0 2.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 91.0 0.0 0.0 0.0 1.0 0.0 0.0				
Max Depth "10"							
Max Bin "1000"							
# Class "23"	0.9979393873951697	0.002060612604830292	19.0 0.0 0.0 0.0 0.0 0.0 0.0	42780	49507	91	10
MAX DEPTH							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"			87428.0 65.0 16.0 0.0 63.0 98.0 352239.0 0.0 0.0 3.0 36.0 1.0 9.0 0.0 2.0 1016.0 14.0 1.0 0.0 1.0				
Max Depth "5"							
Max Bin "1000"							
# Class "5"	0.9957248042	0.0042751958	559.0 26.0 0.0 0.0 3081.0	12838	13060	33	5
Impurity Type "Gini"			87252.0 51.0 15.0 37.0 33.0 49.0 352473.0 0.0 4.0 31.0 19.0 7.0 15.0 4.0 0.0 144.0 8.0 7.0 842.0 2.0				
Max Depth "10"							
Max Bin "1000"							
# Class "5"	0.9982391753133728	0.0017608246866271775	359.0 12.0 0.0 1.0 3313.0	15636	12739	73	10
Impurity Type "Gini"			87231.0 76.0 9.0 76.0 65.0 15.0 352271.0 10.0 0.0 25.0 22.0 1.0 24.0 2.0 0.0 90.0 2.0 2.0 926.0 2.0				
Max Depth "30=max"							
Max Bin "1000"							
# Class "5"	0.9989585231273098	0.0010414768726901371	56.0 9.0 1.0 0.0 3646.0	17860	12676	169	21

2.2. Unbalanced

IMPURITY TYPE							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"							
Max Depth "10"			352199.0 0.0 0.0 5.0 0.0 10.0 0.0 0.0				
Max Bin "1000"			3.0 0.0 289.0 2.0				
# Class "5"	0.9960625842	0.0039374158	79.0 0.0 0.0 3183.0	106725	122543	77	10
Impurity Type "Entropy"							
Max Depth "10"			352452.0 0.0 0.0 13.0 0.0 13.0 0.0 0.0				
Max Bin "1000"			1.0 2.0 779.0 3.0				
# Class "5"	0.9982392888	0.0017607112	20.0 0.0 2.0 3369.0	115663	123598	103	10
MAX BIN							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"			352312.0 0.0 0.0 15.0				
Max Depth "10"			0.0 0.0 0.0 0.0				
Max Bin "100"			0.0 0.0 298.0 2.0				
# Class "5"	0.9959503179	0.0040496821	26.0 0.0 0.0 3255.0	93042	122543	73	10
Impurity Type "Gini"			352199.0 0.0 0.0 5.0				
Max Depth "10"			0.0 10.0 0.0 0.0				
Max Bin "1000"			3.0 0.0 289.0 2.0				
# Class "5"	0.9960625842	0.0039374158	79.0 0.0 0.0 3183.0	106725	122543	77	10
Impurity Type "Gini"	0.9958069106						
Max Depth "10"			351978.0 0.0 50.0 24.0 0.0 5.0 0.0 0.0				
Max Bin "10000"			0.0 3.0 273.0 3.0				
# Class "5"		0.0041930894	59.0 0.0 0.0 3263.0	164719	122046	73	10
# CLASS							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"							
Max Depth "10"							
Max Bin "1000"							
# Class "2"	0.9956172225	0.0043827775	356196	45446	24315	61	10
Impurity Type "Gini"			352199.0 0.0 0.0 5.0				
Max Depth "10"			0.0 10.0 0.0 0.0				
Max Bin "1000"			3.0 0.0 289.0 2.0				
# Class "5"	0.9960625842	0.0039374158	79.0 0.0 0.0 3183.0	106725	122543	77	10
MAX DEPTH							
	Precision	**Wrong Prediction Ratio**	**Confusion Matrix**	**Training Time**	**Testing Time**	**# Nodes**	**Depth**
Impurity Type "Gini"			351503.0 0.0 0.0 0.0				
Max Depth "5"			0.0 0.0 0.0 0.0				
Max Bin "1000"			1.0 0.0 0.0 2.0				
# Class "5"	0.9904133121	0.0095866879	23.0 0.0 0.0 1925.0	103180	135376	29	5
Impurity Type "Gini"			352199.0 0.0 0.0 5.0				
Max Depth "10"			0.0 10.0 0.0 0.0				
Max Bin "1000"			3.0 0.0 289.0 2.0				
# Class "5"	0.9960625842	0.0039374158	79.0 0.0 0.0 3183.0	106725	122543	77	10
Impurity Type "Gini"			352456.0 0.0 0.0 47.0				
Max Depth "30=max"			0.0 14.0 1.0 0.0				
Max Bin "1000"			1.0 1.0 868.0 2.0				
# Class "5"	0.9989616249	0.0010383751	5.0 0.0 0.0 3580.0	167898	128568	197	20

3. Discussion of Results and Conclusions

In the light of above results we can see that even the parameters are antipodal to each other the precision of the tree model is changed really milimetric. However these milimetric changes can be dramatically higher for bigger data sets.Thats why we have to know how these parameters effect the tree model.

Firstly, we measure the results in two different impurity type ; Gini and Entropy. Impurity type determines the calculation of features inforrmation gain and select the feature which gives higher information gain in each iteration until all data set classified in one class.

Their formulas are given below;

$$Gini\ Index = 1 - \sum_j p_j^2 \qquad Entropy = \sum_j -p_j log_2 p_j$$

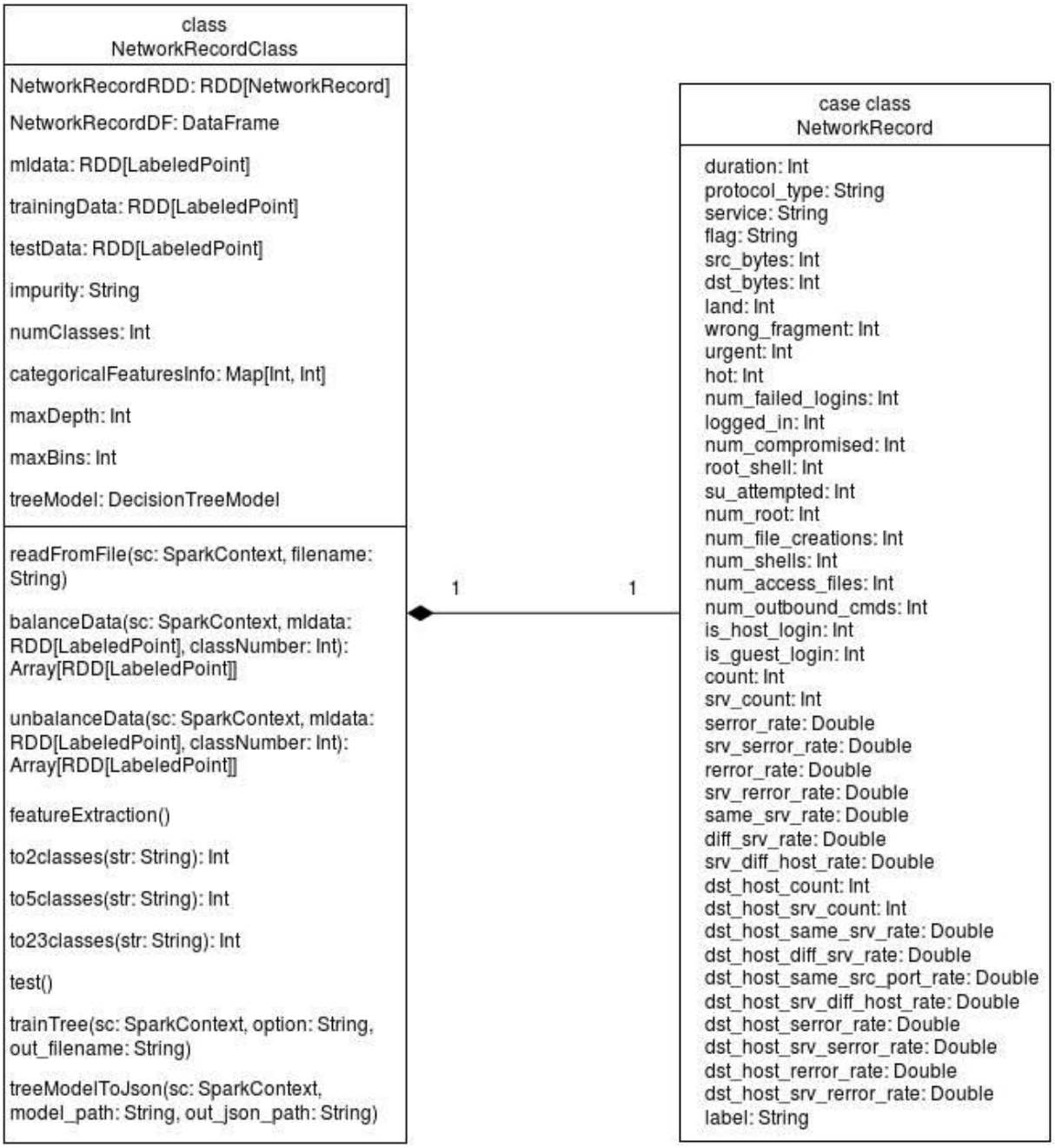
Due to the given formula computation with Entrophy takes little more time. But what is the main difference between Gini and Entropy? Gini is mostly used for continious attributes and Entropy is used for features that occur in classes.

Secondly, increasing number of classes also increase the decision tree precision. Because that more detail classification. We can understand it in such a scenerio if we have 500.000 and we want to classify them in 500.000 number of classes, each class contains only one row and the precision is hundred percent.(actually it is only substitution). On the other hand all features must be used in such a case and all branches travelled like brute force and that increase both the training time and testing time of the data set. In that point this is the trade off between required accuracy level and calculation time.

Third parameter is Max Depth, as results points that increasing max depth returns better decision tree model.This is really logical because that is the restriction of classification, giving max depth cause higher pressure on classification and limit the accuracy.

The last paramer we use is Max Bin which is directly related with the Spark distributed file sytem. Bins are used for handling continuous data and increasing number of bin increase the performance.

4. Class Structure/UML



Above UML diagram represents the class structure of our project. As you can see each NetworkRecordClass has one NetworkRecord. We use NetworkRecord class for representing the features of records.

5. User Guide

First, install the Scala IDE plugin for Eclipse IDE from official website of Scala IDE. Then, create a new “Maven Project” from File->New->Project. Select “Create a simple project option” while creating the project, fill the necessary information for project. After that, add dependencies “spark-mllib_2.11”, “spark-core_2.11”, “spark-sql_2.11” artifacts from “org.apache.spark” group, and necessary maven tools to “pom.xml” file in the project. Refactor project folders and rename “java” to “scala”. Then, add a scala object to “src/main/scala” folder in the project from New->Scala Object. Optionally, scala classes can be added to this scala object. Compilation configurations for the necessary objects are automatically created. To build and run the project, click “Run” button or execute relevant keyboard shortcut.