-> ConnectedComponentsComputation.java

import org.apache.giraph.graph.BasicComputation;

import org.apache.giraph.edge.Edge;

import org.apache.giraph.graph.Vertex;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.NullWritable;

import java.io.IOException;

/\*\*

\* Implementation of the connected component algorithm that identifies

\* connected components and assigns each vertex its "component

\* identifier" (the smallest vertex id in the component).

\*/

public class ConnectedComponentsComputation extends

BasicComputation<IntWritable, IntWritable, NullWritable, IntWritable> {

/\*\*

\* Propagates the smallest vertex id to all neighbors. Will always choose to

\* halt and only reactivate if a smaller id has been sent to it.

\*

\* @param vertex Vertex

\* @param messages Iterator of messages from the previous superstep.

\* @throws IOException

\*/

@Override

public void compute(

Vertex<IntWritable, IntWritable, NullWritable> vertex,

Iterable<IntWritable> messages) throws IOException {

//TODO

}

}

-> KMeansMP.java

import java.util.regex.Pattern;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.Function;

import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.api.java.function.VoidFunction;

import org.apache.spark.mllib.clustering.KMeans;

import org.apache.spark.mllib.clustering.KMeansModel;

import org.apache.spark.mllib.linalg.Vector;

import org.apache.spark.mllib.linalg.Vectors;

import scala.Tuple2;

public final class KMeansMP {

// TODO

public static void main(String[] args) {

if (args.length < 2) {

System.err.println(

"Usage: KMeansMP <input\_file> <results>");

System.exit(1);

}

String inputFile = args[0];

String results\_path = args[1];

JavaPairRDD<Integer, Iterable<String>> results;

int k = 4;

int iterations = 100;

int runs = 1;

long seed = 0;

final KMeansModel model;

SparkConf sparkConf = new SparkConf().setAppName("KMeans MP");

JavaSparkContext sc = new JavaSparkContext(sparkConf);

//TODO

results.saveAsTextFile(results\_path);

sc.stop();

}

}

-> ShortestPathsComputation.java

import org.apache.giraph.graph.BasicComputation;

import org.apache.giraph.conf.LongConfOption;

import org.apache.giraph.edge.Edge;

import org.apache.giraph.graph.Vertex;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.NullWritable;

import java.io.IOException;

/\*\*

\* Compute shortest paths from a given source.

\*/

public class ShortestPathsComputation extends BasicComputation<

IntWritable, IntWritable, NullWritable, IntWritable> {

/\*\* The shortest paths id \*/

public static final LongConfOption SOURCE\_ID =

new LongConfOption("SimpleShortestPathsVertex.sourceId", 1,

"The shortest paths id");

/\*\*

\* Is this vertex the source id?

\*

\* @param vertex Vertex

\* @return True if the source id

\*/

private boolean isSource(Vertex<IntWritable, ?, ?> vertex) {

return vertex.getId().get() == SOURCE\_ID.get(getConf());

}

@Override

public void compute(

Vertex<IntWritable, IntWritable, NullWritable> vertex,

Iterable<IntWritable> messages) throws IOException {

}

}

-> RandomForestMP.java

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.Function;

import org.apache.spark.mllib.classification.SVMModel;

import org.apache.spark.mllib.classification.SVMWithSGD;

import org.apache.spark.mllib.linalg.Vector;

import org.apache.spark.mllib.linalg.Vectors;

import org.apache.spark.mllib.regression.LabeledPoint;

import org.apache.spark.mllib.tree.model.RandomForestModel;

import org.apache.spark.mllib.tree.RandomForest;

import java.util.HashMap;

import java.util.regex.Pattern;

public final class RandomForestMP {

public static void main(String[] args) {

if (args.length < 3) {

System.err.println(

"Usage: RandomForestMP <training\_data> <test\_data> <results>");

System.exit(1);

}

String training\_data\_path = args[0];

String test\_data\_path = args[1];

String results\_path = args[2];

SparkConf sparkConf = new SparkConf().setAppName("RandomForestMP");

JavaSparkContext sc = new JavaSparkContext(sparkConf);

final RandomForestModel model;

Integer numClasses = 2;

HashMap<Integer, Integer> categoricalFeaturesInfo = new HashMap<Integer, Integer>();

Integer numTrees = 3;

String featureSubsetStrategy = "auto";

String impurity = "gini";

Integer maxDepth = 5;

Integer maxBins = 32;

Integer seed = 12345;

// TODO

JavaRDD<LabeledPoint> results = test.map(new Function<Vector, LabeledPoint>() {

public LabeledPoint call(Vector points) {

return new LabeledPoint(model.predict(points), points);

}

});

results.saveAsTextFile(results\_path);

sc.stop();

}

}