Laporan Tubes 1 Pembelajaran Mesin

1. Source Code

myID3

package classifier.ID3;

import weka.classifiers.AbstractClassifier;

import weka.core.AttributeStats;

import weka.core.Instance;

import weka.core.Instances;

import weka.core.converters.ConverterUtils.DataSource;

import weka.filters.Filter;

import weka.filters.unsupervised.instance.SubsetByExpression;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashMap;

/\*\*

\* Created by nathanjamesruntuwene on 9/20/17.

\*/

public class myID3 extends AbstractClassifier {

private myID3Node model;

@Override

public void buildClassifier(Instances instances) throws Exception {

//Init model

model = new myID3Node();

ArrayList<Integer> processedIndex = new ArrayList<>();

String addCondition = "";

recursiveIterate(instances, addCondition, processedIndex, model, instances);

// testInstance(instances);

}

public void recursiveIterate(Instances instances, String decisionCondition, ArrayList<Integer> processedIndexes, myID3Node node, Instances prevInstances) throws Exception

{

if(processedIndexes.size()<4){

double entropyS = calculateEntropy(instances);

if (entropyS>0){

Instances newInstances;

int attributeIndex = decideAttributeFactor(entropyS, instances, decisionCondition, processedIndexes);

ArrayList<Integer> copyList = new ArrayList<>(processedIndexes);

copyList.add(attributeIndex);

node.setKey(attributeIndex);

for (int i=0; i<instances.attribute(attributeIndex).numValues(); i++){

String condition = addStringCondition(decisionCondition,attributeIndex,instances.attribute(attributeIndex).value(i));

node.addChildren(instances.attribute(attributeIndex).value(i));

newInstances = filterInstances(instances,condition);

recursiveIterate(newInstances, insertAnd(condition), copyList, node.getChildren(instances.attribute(attributeIndex).value(i)),instances);

}

}else{

if(instances.numInstances()>0){

node.setLeaf(instances.instance(0).value(instances.classIndex()));

}else{

AttributeStats stats = prevInstances.attributeStats(prevInstances.classIndex());

int[] countResults = (stats.nominalCounts);

int max = -999;

int id = -999;

for(int i=0;i<countResults.length;i++){

if(countResults[i]>max){

max = countResults[i];

id = i;

}

}

System.out.println(instances.attribute(instances.classIndex()).value(id));

node.setLeaf(id);

}

}

}else{

System.out.println("Ada yang sampai sini ga?");

}

}

@Override

public double classifyInstance(Instance instance) throws Exception {

myID3Node node = model;

while(!node.isLeaf || !node.hasChildren()){

node = node.getChildren(instance.stringValue(instance.attribute(node.getKey())));

}

System.out.println(node.getValue());

return node.getValue();

}

public void testInstance(Instances instances) throws Exception{

for(int i=0; i < instances.numInstances(); i++){

classifyInstance(instances.instance(i));

}

}

public int decideAttributeFactor(double entropyS, Instances instances, String addCondition, ArrayList<Integer> processedIndexes) throws Exception{

double maxIG = calculateGain(entropyS, instances, 0, addCondition);

int id = 0;

for(int i=1; i<instances.numAttributes()-1; i++){

if (!processedIndexes.contains(i)){

double gain;

if ((gain = calculateGain(entropyS, instances, i, addCondition)) > maxIG){

maxIG = gain;

id = i;

}

}

}

System.out.println(id);

System.out.println("");

return id;

}

public static double calculateEntropy(Instances instances){

AttributeStats stats = instances.attributeStats(instances.classIndex());

int[] countResults = (stats.nominalCounts);

int totalCount = stats.totalCount;

double ret= 0.0;

for(int i=0; i < countResults.length; i++){

if(countResults[i]>0){

double distribution = ((double)countResults[i]/totalCount);

ret -= distribution\*(Math.log(distribution)/Math.log(2));

}

}

return ret;

}

public static double calculateGain(double entropy, Instances instances, int attributeIndex, String addCondition) throws Exception{

double ret = entropy;

for(int i=0; i<instances.attribute(attributeIndex).numValues(); i++){

// System.out.println(instances.attribute(attributeIndex).value(i));

String val = instances.attribute(attributeIndex).value(i);

if (val.charAt(0) != '\'') {

val = "'" + val + "'";

}

String condition = addCondition + "(ATT"+(attributeIndex+1)+" is " + val +")";

Instances filteredInstances = filterInstances(instances,condition);

// System.out.println(filteredInstances.numInstances());

ret -= calculateEntropy(filteredInstances)\*((double)filteredInstances.numInstances()/instances.numInstances());

}

System.out.println("Information Gain = "+ret);

// System.out.println("");

return ret;

}

public void printInstances(Instances instances){

for(int j=0; j < instances.numInstances(); j++){

Instance curInstance = instances.instance(j);

System.out.println(curInstance);

}

System.out.println("");

}

public static Instances filterInstances(Instances instances, String condition) throws Exception{

SubsetByExpression filter = new SubsetByExpression();

String[] options = new String[2];

options[0] = "-E";

options[1] = condition;

System.out.println(condition);

filter.setOptions(options);

filter.setInputFormat(instances);

Instances filteredInstances = Filter.useFilter(instances, filter);

return filteredInstances;

}

public String addStringCondition(String addOption, int attributeIndex, String value){

String val = value;

if (val.charAt(0) != '\'') {

val = "'" + val + "'";

}

return addOption+"(ATT"+(attributeIndex+1)+" is "+val+")";

}

public String insertAnd(String addOption){

return addOption+" and ";

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String filename = "D:\\Programming\\\_Project\\WekaTest\\test\\weather.nominal.arff";

DataSource source = new DataSource(filename);

Instances data = source.getDataSet();

if (data.classIndex() == -1){

data.setClassIndex(data.numAttributes()-1);

}

myID3 classifier = new myID3();

classifier.buildClassifier(data);

}

}

class myID3Node implements Serializable {

public boolean isLeaf = false;

private int key;

private double value;

private HashMap<String, myID3Node> children;

myID3Node(){

children = new HashMap<>();

value = -999.0;

}

public void addChildren(String value){

myID3Node childNode = new myID3Node();

children.put(value,childNode);

}

public void setLeaf(double value){

this.value = value;

isLeaf = true;

}

public void setKey(int key){

this.key = key;

}

public int getKey(){

return key;

}

public double getValue(){

return value;

}

public myID3Node getChildren(String value){

return children.get(value);

}

public void printChildren(){

System.out.println("Key = "+key);

System.out.println("Value = "+value);

System.out.println(Arrays.asList(children));

}

public boolean hasChildren(){

return children.isEmpty();

}

}

myC45

package classifier.C45;

import classifier.ID3.myID3;

import weka.classifiers.AbstractClassifier;

import weka.core.Attribute;

import weka.core.Capabilities;

import weka.core.Instance;

import weka.core.Instances;

import weka.core.converters.ConverterUtils;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.Serializable;

import java.util.\*;

import java.util.concurrent.ThreadLocalRandom;

/\*\*

\* Created by nathanjamesruntuwene on 9/20/17.

\*/

public class myC45 extends AbstractClassifier {

private DTLNode root;

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String filename = "/Users/anthony/ML/ML-TuBes1/test/weather.numeric.arff";

ConverterUtils.DataSource source = new ConverterUtils.DataSource(filename);

Instances data = source.getDataSet();

if (data.classIndex() == -1) {

data.setClassIndex(data.numAttributes() - 1);

}

myC45 classifier = new myC45();

classifier.buildClassifier(data);

Enumeration<Instance> instanceEnumeration = data.enumerateInstances();

while (instanceEnumeration.hasMoreElements()) {

Instance instance = instanceEnumeration.nextElement();

classifier.classifyInstance(instance);

}

}

@Override

public Capabilities getCapabilities() {

Capabilities result = super.getCapabilities(); // returns the object from weka.classifiers.Classifier

// attributes

result.enable(Capabilities.Capability.NOMINAL\_ATTRIBUTES);

result.enable(Capabilities.Capability.NUMERIC\_ATTRIBUTES);

result.enable(Capabilities.Capability.MISSING\_VALUES);

result.enable(Capabilities.Capability.DATE\_ATTRIBUTES);

// class

result.enable(Capabilities.Capability.NOMINAL\_CLASS);

return result;

}

@Override

public void buildClassifier(Instances instances) throws Exception {

root = new DTLNode();

Instances instancesCopy = new Instances(instances);

fillMissingValue(instancesCopy);

root.buildTree(instancesCopy);

}

private void fillMissingValue(Instances instances) {

Vector<HashMap<Double, Integer>> counter = new Vector<>();

Double[] popularAttribute = new Double[instances.numAttributes()];

Integer[] maxCounter = new Integer[instances.numAttributes()];

for (int i = 0; i < instances.numAttributes(); i++) {

counter.add(new HashMap<>());

popularAttribute[i] = null;

maxCounter[i] = 0;

}

Enumeration<Instance> instanceEnumeration = instances.enumerateInstances();

while (instanceEnumeration.hasMoreElements()) {

Instance instance = instanceEnumeration.nextElement();

Enumeration<Attribute> attributeEnumeration = instance.enumerateAttributes();

while (attributeEnumeration.hasMoreElements()) {

Attribute attribute = attributeEnumeration.nextElement();

if (instance.isMissing(attribute))

continue;

counter.get(attribute.index()).put(instance.value(attribute), counter.get(attribute.index()).getOrDefault(instance.value(attribute), 0) + 1);

if (counter.get(attribute.index()).get(instance.value(attribute)) > maxCounter[attribute.index()]) {

maxCounter[attribute.index()] = counter.get(attribute.index()).get(instance.value(attribute));

popularAttribute[attribute.index()] = instance.value(attribute);

}

}

}

instanceEnumeration = instances.enumerateInstances();

while (instanceEnumeration.hasMoreElements()) {

Instance instance = instanceEnumeration.nextElement();

Enumeration<Attribute> attributeEnumeration = instance.enumerateAttributes();

while (attributeEnumeration.hasMoreElements()) {

Attribute attribute = attributeEnumeration.nextElement();

if (instance.isMissing(attribute)) {

instance.setValue(attribute, popularAttribute[attribute.index()]);

}

}

}

}

@Override

public double classifyInstance(Instance instance) throws Exception {

return root.classify(instance);

}

}

class DTLNode implements Serializable {

private boolean isLeaf = false;

private Double classifiedClass;

private Attribute attributeToClassify;

private HashMap<Double, DTLNode> children;

private DTLNode popularChild;

private Double threshold;

private DTLNode parent = null;

private boolean pruneChecked = false;

DTLNode() {

children = new HashMap<>();

}

public DTLNode getParent() {

return parent;

}

// hackish method to swap, see https://stackoverflow.com/a/16826296

private double returnFirst(double x, double y) {

return x;

}

void buildTree(Instances instances) {

if (instances.isEmpty()) {

throw new Error("EMPTY INSTANCES");

}

if (instances.numDistinctValues(instances.classAttribute()) == 1) {

this.isLeaf = true;

classifiedClass = instances.firstInstance().classValue();

return;

}

if (instances.numAttributes() <= 1) {

makeLeaf(instances);

return;

}

if (calcInformationGainMax(instances) == 0) {

makeLeaf(instances);

return;

}

makeChildren(instances);

}

private void makeChildren(Instances instances) {

HashMap<Double, Instances> childInstances = new HashMap<>();

HashMap<Double, Integer> counter = new HashMap<>();

Enumeration<Instance> instanceEnumeration = instances.enumerateInstances();

Integer maxCount = 0;

Double favValue = null;

while (instanceEnumeration.hasMoreElements()) {

Instance instance = instanceEnumeration.nextElement();

if (enableThreshold(this.attributeToClassify)) {

if (instance.value(this.attributeToClassify) <= this.threshold) {

childInstances.putIfAbsent(0.0, new Instances(instances, 0));

childInstances.get(0.0).add(instance);

} else {

childInstances.putIfAbsent(1.0, new Instances(instances, 0));

childInstances.get(1.0).add(instance);

}

continue;

}

childInstances.putIfAbsent(instance.value(this.attributeToClassify), new Instances(instances, 0));

childInstances.get(instance.value(this.attributeToClassify)).add(instance);

counter.put(instance.value(this.attributeToClassify), counter.getOrDefault(instance.value(this.attributeToClassify), 0) + 1);

if (counter.get(instance.value(this.attributeToClassify)) > maxCount) {

maxCount = counter.get(instance.value(this.attributeToClassify));

favValue = instance.value(this.attributeToClassify);

}

}

Double finalFavValue = favValue;

childInstances.forEach((val, ci) -> {

DTLNode node = new DTLNode();

ci.deleteAttributeAt(this.attributeToClassify.index());

node.buildTree(ci);

node.parent = this;

if (Objects.equals(val, finalFavValue)) {

this.popularChild = node;

}

this.children.put(val, node);

});

}

// Calculate the maximum information gain

private double calcInformationGainMax(Instances instances) {

double informationGainMax = 0;

Enumeration<Attribute> attributeEnumeration = instances.enumerateAttributes();

while (attributeEnumeration.hasMoreElements()) {

Attribute attribute = attributeEnumeration.nextElement();

if (attribute == instances.classAttribute())

continue;

double informationGain;

Double thresholdMax = null;

if (enableThreshold(attribute)) {

double[] attributeValues = instances.attributeToDoubleArray(attribute.index());

Arrays.sort(attributeValues);

double[] thresholdCandidate = new double[attributeValues.length - 1];

for (int i = 0; i < attributeValues.length - 1; i++) {

thresholdCandidate[i] = (attributeValues[i + 1] + attributeValues[i]) / 2;

}

if (thresholdCandidate.length > 10) {

// Fisher–Yates shuffle

for (int i = 0; i < thresholdCandidate.length - 1; i++) {

int j = ThreadLocalRandom.current().nextInt(i, thresholdCandidate.length);

thresholdCandidate[i] = returnFirst(thresholdCandidate[j], thresholdCandidate[j] = thresholdCandidate[i]);

}

thresholdCandidate = Arrays.copyOf(thresholdCandidate, 10);

}

thresholdMax = thresholdCandidate[0];

double maxGain = 0;

for (double candidate : thresholdCandidate) {

double gain = calcThresholdGain(candidate, attribute, instances);

if (gain > maxGain) {

maxGain = gain;

thresholdMax = candidate;

}

}

informationGain = calcInformationGain(attribute, instances, thresholdMax);

} else {

informationGain = calcInformationGain(attribute, instances);

}

if (informationGain > informationGainMax) {

informationGainMax = informationGain;

this.attributeToClassify = attribute;

this.threshold = thresholdMax;

}

}

return informationGainMax;

}

// Make leaf with classified class as the most frequent class in instances

private void makeLeaf(Instances instances) {

double[] instancesClassValues = instances.attributeToDoubleArray(instances.classIndex());

HashMap<Double, Integer> counter = new HashMap<>();

Integer maxCount = 0;

Double maxCountValue = null;

for (double val : instancesClassValues) {

Integer count = counter.getOrDefault(val, 0) + 1;

counter.put(val, count);

if (maxCount < count) {

maxCount = count;

maxCountValue = val;

}

}

this.isLeaf = true;

this.classifiedClass = maxCountValue;

}

private boolean enableThreshold(Attribute attribute) {

return attribute.isNumeric();

}

private double calcThresholdGain(double candidate, Attribute attribute, Instances instances) {

// TODO(ParadiseCatz): Is this correct?

return calcInformationGain(attribute, instances, candidate);

}

private double calcInformationGain(Attribute attribute, Instances instances) {

try {

return myID3.calculateGain(myID3.calculateEntropy(instances), instances, attribute.index(), "");

} catch (Exception e) {

throw new Error(e);

}

}

private double calcInformationGain(Attribute attribute, Instances instances, double threshold) {

double informationGain;

double[] attributeValues = instances.attributeToDoubleArray(attribute.index());

for (int i = 0; i < instances.numInstances(); i++) {

if (instances.instance(i).value(attribute) <= threshold) {

instances.instance(i).setValue(attribute, 0.0);

} else {

instances.instance(i).setValue(attribute, 1.0);

}

}

try {

informationGain = myID3.calculateGain(myID3.calculateEntropy(instances), instances, attribute.index(), "");

} catch (Exception e) {

throw new Error(e);

}

for (int i = 0; i < instances.numInstances(); i++) {

instances.instance(i).setValue(attribute, attributeValues[i]);

}

return informationGain;

}

Double classify(Instance instance) {

if (this.isLeaf) {

return this.classifiedClass;

}

if (enableThreshold(this.attributeToClassify)) {

if (instance.value(this.attributeToClassify) <= this.threshold) {

return this.children.get(0.0).classify(instance);

} else {

return this.children.get(1.0).classify(instance);

}

}

if (instance.isMissing(this.attributeToClassify) || this.children.get(instance.value(this.attributeToClassify)) == null) {

return this.popularChild.classify(instance);

}

return this.children.get(instance.value(this.attributeToClassify)).classify(instance);

}

public boolean hasChildren() {

return children.isEmpty();

}

void prunReducedError(Instances evaluationSet) {

ArrayList<DTLNode> leafNodes = this.getAllLeaf();

int prevErr = countError(evaluationSet);

for (DTLNode leafNode : leafNodes) {

DTLNode nodeParent = leafNode.getParent();

Double temp = nodeParent.classifiedClass;

nodeParent.isLeaf = true;

nodeParent.classifiedClass = leafNode.classifiedClass;

int afterErr = countError(evaluationSet);

if (afterErr > prevErr) {

nodeParent.isLeaf = false;

nodeParent.classifiedClass = temp;

} else {

prevErr = afterErr;

}

}

}

ArrayList<DTLNode> getAllLeaf() {

ArrayList<DTLNode> list = new ArrayList<>();

getLeaf(list, this);

return list;

}

void getLeaf(ArrayList<DTLNode> list, DTLNode node) {

if (node.isLeaf) {

list.add(node);

} else {

node.children.forEach((aDouble, dtlNode) -> getLeaf(list, dtlNode));

}

}

int countError(Instances evaluationSet) {

int error = 0;

for (int i = 0; i < evaluationSet.numInstances(); i++) {

Instance instance = evaluationSet.instance(i);

if (instance.value(instance.classIndex()) != classify(instance)) {

error++;

}

}

return error;

}

}

main

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.util.Random;

import classifier.C45.myC45;

import classifier.ID3.myID3;

import weka.classifiers.AbstractClassifier;

import weka.classifiers.Evaluation;

import weka.classifiers.trees.J48;

import weka.core.Instance;

import weka.core.Instances;

import weka.core.SerializationHelper;

import weka.core.Utils;

import weka.core.converters.ConverterUtils.DataSource;

import weka.filters.Filter;

import weka.filters.supervised.attribute.Discretize;

import weka.filters.supervised.instance.Resample;

import weka.filters.unsupervised.attribute.Remove;

/\*\*

\* Created by nathanjamesruntuwene on 9/30/17.

\*/

public class Main {

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String loadLocation = Utils.getOption("load\_location", args);

String saveLocation = Utils.getOption("save\_location", args);

String trainingLocation = Utils.getOption("training\_location", args);

String testLocation = Utils.getOption("test\_location", args);

String removedAttributes = Utils.getOption("remove", args);

boolean useResample = Utils.getFlag("resample", args);

boolean useCrossValidation = Utils.getFlag("use\_cv", args);

String splitPercentage = Utils.getOption("split",args);

AbstractClassifier classifier;

if (loadLocation.length() > 0) { // LOAD MODEL

classifier = (AbstractClassifier) SerializationHelper.read(loadLocation);

System.out.println("Classifier loaded");

if (testLocation.length() > 0) {

DataSource testSource = new DataSource(testLocation);

Instances testData = testSource.getDataSet();

int countCorrect = 0;

for (int i = 0; i < testData.size(); i++) {

Instance instance = testData.get(i);

if (classifier.classifyInstance(instance) == instance.classValue()) {

countCorrect++;

}

}

double accuracy = ((double) (countCorrect \* 100) / testData.size());

System.out.println("Test Data Accuracy: " + accuracy + " %");

}

} else { // BUILD MODEL

String classifierName = Utils.getOption("classifier", args);

DataSource trainingSource = new DataSource(trainingLocation);

Instances trainingDataFull = trainingSource.getDataSet();

// Remove attributes

Remove remove = new Remove();

remove.setAttributeIndices(removedAttributes);

remove.setInputFormat(trainingDataFull);

Instances removedTrainingData = Filter.useFilter(trainingDataFull, remove);

// Apply resample

Instances resampledTrainingData;

if (useResample) {

Resample resample = new Resample();

System.out.println("Input resample size: ");

double resampleSize = Double.parseDouble(br.readLine());

System.out.println("Input bias value: ");

double biasValue = Double.parseDouble(br.readLine());

resample.setSampleSizePercent(resampleSize);

resample.setBiasToUniformClass(biasValue);

resample.setInputFormat(removedTrainingData);

resampledTrainingData = Filter.useFilter(removedTrainingData, resample);

} else {

resampledTrainingData = removedTrainingData;

}

// Assign class index

if (resampledTrainingData.classIndex() == -1)

resampledTrainingData.setClassIndex(resampledTrainingData.numAttributes() - 1);

// Assign classifiers

Instances trainingData;

if (classifierName.equals("myID3")) {

classifier = new myID3();

// Discretize attributes

Discretize discretize = new Discretize();

discretize.setInputFormat(resampledTrainingData);

trainingData = Filter.useFilter(resampledTrainingData, discretize);

System.out.println(trainingData.toString());

} else if (classifierName.equals("myC45")) {

classifier = new myC45();

trainingData = resampledTrainingData;

} else if (classifierName.equals("ID3")) {

classifier = new myID3();

// Discretize attributes

Discretize discretize = new Discretize();

discretize.setInputFormat(resampledTrainingData);

trainingData = Filter.useFilter(resampledTrainingData, discretize);

System.out.println(trainingData.toString());

} else if (classifierName.equals("J48")) {

classifier = new J48();

trainingData = resampledTrainingData;

} else {

System.out.println("Classifier needs to be either 'myID3', 'myC45', 'ID3', or 'J48");

return;

}

if (useCrossValidation) { // Use Cross Validation

System.out.println("Using 10-fold cross validation");

Evaluation eval = new Evaluation(trainingData);

eval.crossValidateModel(classifier,trainingData,10,new Random());

System.out.println(eval.toSummaryString());

} else if (splitPercentage.length() > 0) { // Use Split percentage

System.out.println("Using split percentage");

trainingData.randomize(new Random());

int threshold = (int)Math.round((double)trainingData.numInstances() \* Double.parseDouble(splitPercentage) / 100.0D);

int numTestingInstances = trainingData.numInstances() - threshold;

Instances training = new Instances(trainingData, 0, threshold);

Instances testing = new Instances(trainingData, threshold, numTestingInstances);

classifier.buildClassifier(training);

DataSource testSource = new DataSource(testLocation);

Instances testData = testSource.getDataSet();

int countCorrect = 0;

for (int i = 0; i < testData.size(); i++) {

Instance instance = testData.get(i);

if (classifier.classifyInstance(instance) == instance.classValue()) {

countCorrect++;

}

}

double accuracy = ((double) (countCorrect \* 100) / testData.size());

System.out.println("Test Data Accuracy: " + accuracy + " %");

} else { // Use training-test

System.out.println("Using training-test");

classifier.buildClassifier(trainingData);

int countCorrect = 0;

for (int i = 0; i < trainingData.size(); i++) {

Instance instance = trainingData.get(i);

if (classifier.classifyInstance(instance) == instance.classValue()) {

countCorrect++;

}

}

double accuracy = ((double) (countCorrect \* 100) / trainingData.size());

System.out.println("Test Data Accuracy: " + accuracy + " %");

}

}

// SAVE MODEL

if (saveLocation.length() > 0)

SerializationHelper.write(saveLocation, classifier);

}

}

2. Perbandingan hasil ID3 & J48 weka

ID3 vs myID3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Option | Classifier | Weather Nominal | Weather Numeric | Iris |
| Full Training | ID3 | 100% | 100% | 99.33% |
|  | myID3 | 100% | 64% | 33.33% |
| 10-folds CV | ID3 | 100% | 42.85% | 92% |
|  | myID3 | 78.5% | 64% | 33.33% |
| Percentage Split | ID3 | 100% | 66.67% | 80% |
|  | myID3 | 66.67% | 100% | 26,57% |

J48 vs myC45

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Option | Classifier | Weather Nominal | Weather Numeric | Iris |
| Full Training | J48 | 71,43% | 85,71% | 96% |
|  | myC45 | 71,43% | 71,43% | 45,33% |
| 10-folds CV | J48 | 50% | 42.86% | 94% |
|  | myC45 | 71,43% | 64,29% | 47,33% |
| Percentage Split(80%) | ID3 | 66,67% | 66.67% | 73,33% |
|  | myC45 | 33,33% | 66.67% | 43,3% |

3. Pembagian Tugas

|  |  |
| --- | --- |
| Nama/NIM | Tugas |
| Davin Prasetya – 13514003 | Pembuatan myID3 |
| Nathan J. Runtuwene – 13514083 | Pembuatan main code pengaksesan weka |
| Christian Anthony S. – 13514085 | Pembuatan myC45 |