Machine Learning Lab 3

Weka Decision Trees

By the end of lab sheet 2 you:

1. Have some basic tools to help train and evaluate classifiers;
2. Be able to form a contingency table for test predictions;
3. Have implemented a basic majority class classifier;
4. become familiar with the inner workings of a basic Classifier in Weka, ZeroR;

Evaluated some classifiers on the Mosquito problemBy the end of lab sheet 3 you will have:

1. explored the J48 classifier in Weka (C4.5 decision tree);
2. be able to calculate information gain and gain ratio;
3. have an idea as to the different tree implementations in Weka;
4. practice building trees by hand.

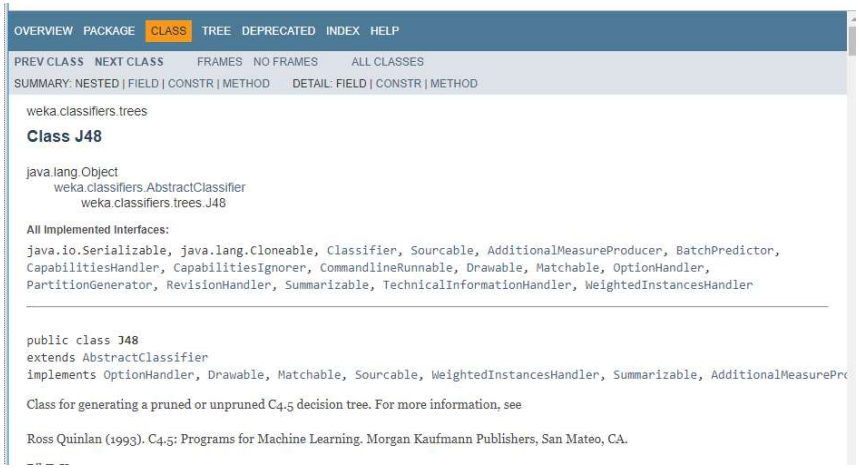
Prerequisite:

1. You should use the same project as the last lab for the Weka exercises

1) Getting started with J48 Classifier in Weka

The C4.5 implementation in Weka is called J48. J48 is a re-implementation of C4.5 release 8 in Java (hence the name J48).

http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html



OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

weka.classifiers.trees

Class J48

java.lang.Object
weka.classifiers.AbstractClassifier
weka.classifiers.trees.J48

All Implemented Interfaces:

java.io.Serializable, java.lang.Cloneable, Classifier, Sourcable, AdditionalMeasureProducer, BatchPredictor, CapabilitiesHandler, CapabilitiesIgnorer, CommandlineRunnable, Drawable, Matchable, OptionHandler, PartitionGenerator, RevisionHandler, Summarizable, TechnicalInformationHandler, WeightedInstancesHandler

public class J48
extends AbstractClassifier
implements OptionHandler, Drawable, Matchable, Sourcable, WeightedInstancesHandler, Summarizable, AdditionalMeasurePro

Class for generating a pruned or unpruned C4.5 decision tree. For more information, see

Ross Quinlan (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Mateo, CA.

Its capabilities are inherited from AbstractClassifier, which by default allows all data types

```
/**
 * Returns the Capabilities of this classifier. Maximally permissive
 * capabilities are allowed by default. Derived classifiers should
 * override this method and first disable all capabilities and then
 * enable just those capabilities that make sense for the scheme.
 *
 * @return              the capabilities of this object
 * @see                 Capabilities
 */
public Capabilities getCapabilities() {
  Capabilities result = new Capabilities(this);
  result.enableAll();

  return result;
}
```

Overriding this method allows you to control what sort of data a classifier can handle. So we can assume J48 can handle all sorts of input, including missing values

The buildClassifier method itself is pretty short

```
public void buildClassifier(Instances instances)
    throws Exception {

  ModelSelection modSelection;

  if (m_binarySplits)
    modSelection = new BinC45ModelSelection(m_minNumObj, instances, m_useMDLcorrection);
  else
    modSelection = new C45ModelSelection(m_minNumObj, instances, m_useMDLcorrection);
  if (!m_reducedErrorPruning)
    m_root = new C45PruneableClassifierTree(modSelection, !m_unpruned, m_CF,
                                            m_subtreeRaising, !m_noCleanup, m_collapseTree);
  else
    m_root = new PruneableClassifierTree(modSelection, !m_unpruned, m_numFolds,
                                         !m_noCleanup, m_Seed);
  m_root.buildClassifier(instances);
  if (m_binarySplits) {
    ((BinC45ModelSelection)modSelection).cleanup();
  } else {
    ((C45ModelSelection)modSelection).cleanup();
  }
}
```

If this parameter is false, a node is created for every value of nominal attributes. It defaults to false.

```
/** Binary splits on nominal attributes? */
private boolean m_binarySplits = false;
```

It basically builds one of two types of tree, based on whether reduced error pruning is used. The default is to not use it

```
/** Use reduced error pruning? */
private boolean m_reducedErrorPruning = false;
```

Before we go further, it is worth just trying building a classifier and looking at the tree.

1. **Go to the UCI archive, get the balloons and breast cancer data**

http://mlr.cs.umass.edu/ml/datasets/Balloons

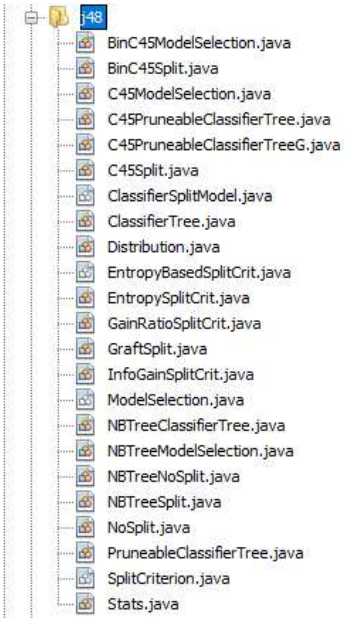
http://mlr.cs.umass.edu/ml/datasets/Breast+Cancer

and edit the file to make it into an ARFF.

2. **Load the data into a single Instances, build a tree on the whole data, then print out the whole tree.**
3. **Set binary splits to true, and repeat. Notice the difference in tree (if any!).**
4. **Do the same with reduced error pruning.**

2. Digging Deeper into J48

J48 has a whole package of helper classes.



We will look closer at the default behaviour:

```
if (!m_reducedErrorPruning)
  m_root = new C45PruneableClassifierTree(modSelection, !m_unpruned, m_CF,
                                          m_subtreeRaising, !m_noCleanup, m_collapseTree);
```

```
public class C45PruneableClassifierTree
  extends ClassifierTree {

/**
 * Constructor for pruneable tree structure. Stores reference
 * to associated training data at each node.
 *
 * @param toSelectLocModel selection method for local splitting model
 * @param pruneTree true if the tree is to be pruned
 * @param cf the confidence factor for pruning
 * @param raiseTree
 * @param cleanup
 * @throws Exception if something goes wrong
 */
```

So the ModelSelection object controls the attribute selection. We will come back to that.

```
public void buildClassifier(Instances data) throws Exception {

  // can classifier tree handle the data?
  getCapabilities().testWithFail(data);
```

Tests capabilities here, not in J48. What sort of data does it work for? Have a look. Moving on: the most important method is buildTree.

```
/**
 * Builds the tree structure.
 *
 * @param data the data for which the tree structure is to be
 * generated.
 * @param keepData is training data to be kept?
 * @throws Exception if something goes wrong
 */
public void buildTree(Instances data, boolean keepData) throws Exception {
```

The core of this method is this:

```
m_localModel = m_toSelectModel.selectModel(data);
if (m_localModel.numSubsets() > 1) {
  localInstances = m_localModel.split(data);
  data = null;
  m_sons = new ClassifierTree [m_localModel.numSubsets()];
  for (int i = 0; i < m_sons.length; i++) {
    m_sons[i] = getNewTree(localInstances[i]);
    localInstances[i] = null;
  }
}else{
  m_isLeaf = true;
  if (Utils.eq(data.sumOfWeights(), 0))
    m_isEmpty = true;
  data = null;
}
```

On a high level, the m_localModel splits the data into an array of instances called localInstances, creates an array of ClassifierTree offspring (m_sons), then calls **`getNewTree`** for each offspring. This is recursive.

```
protected ClassifierTree getNewTree(Instances data) throws Exception {

  ClassifierTree newTree = new ClassifierTree(m_toSelectModel);
  newTree.buildTree(data, false);

  return newTree;
}
```


Compare this to the lecture notes.

Top-Down Induction of Decision Trees

```
buildTree(DataSet D)
        TreeNode t= new TreeNode(D)
//Base case: stop building a tree
        if(stoppingCriteria(D))
            t.setAsLeaf()
            return t
//Recursive cases
//1. Choose an attribute
        Attribute A=chooseAttribute(D)
//2. Split data by attribute
        DataSet[] s= splitData(D,A)
        t.offspring= new TreeNode[size(s)]
//3. Recursively call for each split (depth first)
        for i:=1 to size(s)
                t.offspring[i]=buildTree(s[i])
        return t
```

Step 3 is handled above. Step 1 and 2 are handled by the ModelSelection object. ModelSelection is an abstract class with two subclasses, C45ModelSelection, and BinC45ModelSelection. Lets look at the former.

```
/**
 * Class for selecting a C4.5-type split for a given dataset.
 *
 * @author Eibe Frank (eibe@cs.waikato.ac.nz)
 * @version $Revision: 8034 $
 */
public class C45ModelSelection
  extends ModelSelection {
```

The key method is SelectModel. It is fairly long, but these are the distilled operations

1. Attribute Selection

```
123
124         // For each attribute.
125         for (i = 0; i < data.numAttributes(); i++){
```

```
130           // Get models for current attribute.
131           currentModel[i] = new C45Split(i,m_minNoObj,sumOfWeights,m_useMDLcorrection);
132           currentModel[i].buildClassifier(data);
```

```
158         // Find "best" attribute to split on.
159         minResult = 0;
160         for (i=0;i<data.numAttributes();i++){
```

2. Stopping Criteria:

```
// Check if all Instances belong to one class or if not
// enough Instances to split.
checkDistribution = new Distribution(data);
noSplitModel = new NoSplit(checkDistribution);
if (Utils.sm(checkDistribution.total(),2*m_minNoObj) ||
    Utils.eq(checkDistribution.total(),
             checkDistribution.perClass(checkDistribution.maxClass())))
  return noSplitModel;
```

And

```
// Check if any useful split was found.
if (validModels == 0)
  return noSplitModel;
```

I would like you to use this code to calculate the information gain for the play golf example.

```
/**
 * This method is a straightforward implementation of the information
 * gain criterion for the given distribution.
 */
public final double splitCritValue(Distribution bags) {
```

1. **Look at the class `Distribution` and `InfoGainSplitCrit`, write some code to create and initialise a `Distribution` object with some of the values used in the lecture examples. Use an `InfoGainSplitCrit` object with `splitCritVal` to find the information gain. Do they match the ones we did by hand? If not, why not? Add (temporarily) print statements to `splitCritVal` to figure out the calculations.**
2. **Repeat with `EntropySplitCrit`**
3. **Load the play golf example and build a J48 Tree, registering the printed values statements. Use the Instances to find out the distribution for the outlook attribute, then work out the information gain.**
4. **(if you want!) Write a new `SplitCriterion` and test it as above. Try implement ChiSquared.**

J48 Parameters

Generally, J48 is used off the shelf without tuning. The only parameter I have played with is the minimum number of instances per tree.

J48 has the following parameters that can be adjusted.

- **binarySplits** This specifies whether to use binary splits on nominal data. This is a process by which the tree is grown by considering one nominal value versus all other nominal values instead of considering a split on each nominal value individually. This results in a tree where there are only two branches from any node.

- **confidenceFactor** This determines how aggressive the pruning process will be. The higher this value, the more 'confident' you are that the data you are learning from is a good representation of all possible events, and therefore the less pruning that will occur. Smaller values induce more pruning. This significantly affects classifier performance.

- **minNumObj** This determines what the minimum number of observations are allowed at each leaf of the tree. This is another way to control overfitting.

- **numFolds** This determines how much of the data will be used to prune the tree. One of the folds is held out for pruning while the rest grow the tree. The default value of J48 Classifier Parameters 2 three means one third of the data is used for pruning, while two thirds are used for growing the tree. Setting this number too low will increase overfitting.

- **unpruned** This specifies if the tree should not be pruned.

- **reducedErrorPruning** Reduced error pruning is an alternative algorithm for pruning that focuses on minimizing the statistical error of the tree, instead of the misclassification rate. **This is not the default pruning mechanism.**

- **subtreeRaising** This is a specific method of pruning whereby a whole set of branches further down the tree are moved up to replace branches that were grown above it. **This is the default pruning mechanism**

- **useLaplace** This applies laplace smoothing to counts at the leaves. This is also sometimes called additive smoothing, and is a method by which a certain number is added to all instances in order to eliminate circumstances that are statistically undesirable, such as encountering the number zero. This is most useful when predicting probabilities.

3. Other decision trees in Weka

Explore some of the other decision trees available in Weka experiment and compare to J48.

4. Examples to do by Hand

The following exercises are to do by hand, but feel free to verify your answers in code

1. Construct a classification tree for the data in Table 1 using the following rules:

1. Stopping criteria

"stop if all the data are one class or there are two or less data or all attributes have been used as parents"

2. Greedy selection rule:

"Select the attribute that, after splitting, has the most of Class 1 in one node"

3. Branching rule

"create a new node for each attribute value for the attribute selected"

Evaluate the accuracy on the training data. Now split the data 50/50 and rebuild the tree. Assess the accuracy on the test data.

2. Calculate the information, the information gain and the information gain ratio for each attribute on the data in Table 2.

3. Calculate the Chi-squared statistic for each attribute on the data in Table 2.

4. Construct an ID3 type of tree using the data in Table 3.

5. a). Prune the tree you constructed in Question 4 to remove as many branches as possible while keeping the accuracy above 95%.

b). For the attribute marital status, form binary splits by grouping

together the class values and assess information gain. Does this

form better trees?

6 CART (classification and regression trees) uses selection techniques called Gini and Twoing, both of which are similar to information gain and Chi-squared.

Calculate the Gini Index and the Gini Split index for the data in Tables 1, 2, and 3.

Table 1: Fraud Detection

Large Purchase	Electronics	Card Type	Referred	Fraud
0	0	0	0	0
1	0	0	0	0
0	1	1	0	0
0	1	1	1	0
0	0	1	0	0
0	0	1	1	0
1	0	1	0	1
0	1	1	0	0
0	1	1	1	0
1	1	0	0	1
1	1	1	0	1
1	1	0	0	1
1	1	0	0	1
1	1	0	0	1
1	1	1	0	1
1	1	0	0	0
1	1	1	0	1
0	1	1	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	0	0
0	1	0	1	0
0	1	0	0	0
0	1	1	1	0
1	1	0	0	1
0	1	1	0	0
0	0	0	0	0
0	1	0	0	0
0	1	1	0	0
0	0	1	1	0

Machine Learning Lab 3

University of East Anglia

Table 2: Marketing. Information on whether a company purchases a product

Industry	Size	Quoted		BUY
1	0	Y	A	TRUE
1	1	N	A	FALSE
2	0	N	B	FALSE
2	1	N	C	FALSE
3	0	N	C	TRUE
3	1	Y	A	FALSE
3	0	Y	B	TRUE
3	1	Y	A	FALSE
1	0	N	A	FALSE
1	1	Y	B	TRUE
1	0	Y	B	TRUE
1	1	N	B	FALSE
2	0	Y	C	FALSE

Table 3: Tax Cheating

Refund	Marital Status	Salary (k)	Cheat
Y	Single	125	N
N	Married	100	N
N	Single	70	N
Y	Divorced	120	N
N	Married	95	Y
N	Married	60	N
Y	Divorced	220	N
N	Single	85	Y
N	Married	75	N
N	Single	90	Y

Machine Learning Lab 3

University of East Anglia

Optional Topic

Decision Trees in sci-kit learn

A good top level description is here

https://scikit-learn.org/stable/modules/tree.html

```
class DecisionTreeClassifier(ClassifierMixin, BaseDecisionTree):
    """A decision tree classifier.

    Read more in the :ref:`User Guide <tree>`.

    Parameters
    ----------
    criterion : {"gini", "entropy"}, default="gini"
        The function to measure the quality of a split. Supported criteria are
        "gini" for the Gini impurity and "entropy" for the information gain.

    splitter : {"best", "random"}, default="best"
        The strategy used to choose the split at each node. Supported
        strategies are "best" to choose the best split and "random" to choose
        the best random split.

    max_depth : int, default=None
        The maximum depth of the tree. If None, then nodes are expanded until
        all leaves are pure or until all leaves contain less than
        min_samples_split samples.

    min_samples_split : int or float, default=2
        The minimum number of samples required to split an internal node:

        - If int, then consider `min_samples_split` as the minimum number.
        - If float, then `min_samples_split` is a fraction and
          `ceil(min_samples_split * n_samples)` are the minimum
          number of samples for each split.

        .. versionchanged:: 0.18
           Added float values for fractions.

    min_samples_leaf : int or float, default=1
        The minimum number of samples required to be at a leaf node.
        A split point at any depth will only be considered if it leaves at
        least ``min_samples_leaf`` training samples in each of the left and
        right branches.  This may have the effect of smoothing the model,
        especially in regression.

        - If int, then consider `min_samples_leaf` as the minimum number.
        - If float, then `min_samples_leaf` is a fraction and
          `ceil(min_samples_leaf * n_samples)` are the minimum
          number of samples for each node.

        .. versionchanged:: 0.18
           Added float values for fractions.

    min_weight_fraction_leaf : float, default=0.0
        The minimum weighted fraction of the sum total of weights (of all
        the input samples) required to be at a leaf node. Samples have
        equal weight when sample_weight is not provided.

    max_features : int, float or {"auto", "sqrt", "log2"}, default=None
        The number of features to consider when looking for the best split:
```

Entropy or gini, I don't think it uses gain ratio

Random splits are useful for ensembles

The default values for the min and max parameters will lead to a very large tree, since no pruning is performed. You can control the size either by setting these parameters or by using pruning, through the ccp_alpha parameter

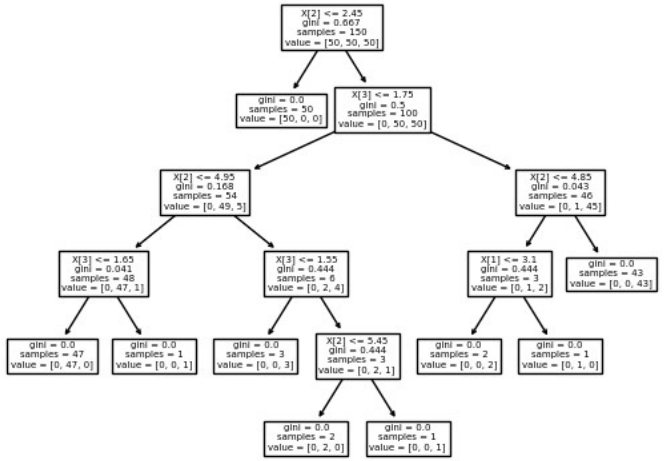
```
ccp_alpha : non-negative
float, default=0.0
    Complexity parameter
used for Minimal Cost-
Complexity Pruning. The
    subtree with the
largest cost complexity
that is smaller than
    ``ccp_alpha`` will be
chosen. By default, no
pruning is performed. See

:ref:`minimal_cost_complexi
ty_pruning` for details.
```


There are some nice visualisation tools. Try this

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

X, y = load_iris(return_X_y=True)
tr = tree.DecisionTreeClassifier()
tr.fit(X, y)
tree.plot_tree(tr)
plt.show()
```



Then try the same but with one of the problems we used with Weka, and explore what happens when you change the tree parameters by visualising the tree and measureing tesat accuracy. Try making it very shallow trees. Some links if you want to explore further

https://www.datacamp.com/community/tutorials/decision-tree-classification-python

https://stackabuse.com/decision-trees-in-python-with-scikit-learn/

https://towardsdatascience.com/scikit-learn-decision-trees-explained-803f3812290d