Machine Learning Lab 4

Weka Ensembles

By the end of lab sheet 3 you:

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

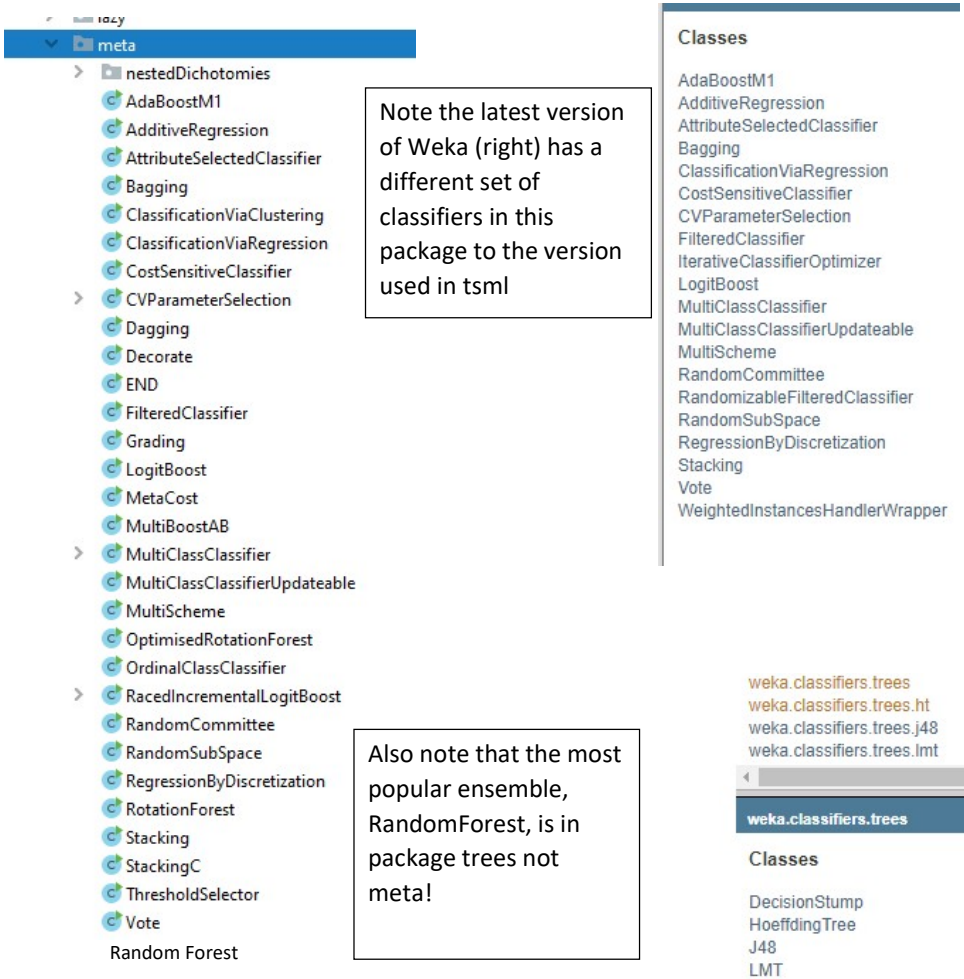
In lab sheet 4 you will:

1. Explore the ensembles available in Weka and tsml
2. (informally) experimentally compare default bagging with random forest
3. (informally) experimentally compare boosting algorithms
4. Build a simple ensemble of J48 decision trees

Overview of Ensembles in Weka

The ensembles in Weka are in weka.classifiers.meta package

https://weka.sourceforge.io/doc.dev/



Note the latest version of Weka (right) has a different set of classifiers in this package to the version used in tsml

Also note that the most popular ensemble, RandomForest, is in package trees not meta!

a

Bagging and Random Forest

Bagging Classifier in Weka

The Bagging algorithm is the pregenitor of all modern ensembles. It works with any base classifier. It is very simple: to build, resample the training data for each base classifier and build independently. To classify, make a prediction with each classifier, then sums predicted probabilities. Open the source code for class **`Bagging`** and have a look. The base classifier is stored in **`m_Classifier`** and set by **`setClassifier`** (both inherited from a tortuous inheritance hierarchy)

```
/**
 * Set the base learner.
 *
 * @param newClassifier the classifier to use.
 */
public void setClassifier(Classifier newClassifier) {

  m_Classifier = newClassifier;
}
```

The number of classifiers is stored in **`m_numberOfIterations.`**

When built, **`m_numberOfIterations`** copies of **`m_Classifier`** are made and stored in **`m_Classifiers`** (again, this is higher in the inheritance hierarchy, this in IteratedSingleClassifierEnhancer)

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

  if (m_Classifier == null) {
    throw new Exception("A base classifier has not been specified!");
  }
  m_Classifiers = AbstractClassifier.makeCopies(m_Classifier, m_NumIterations);
}
```

The key parameter in Bagging is **`m_BagSizePercent`**

```
/** The size of each bag sample, as a percentage of the training size */
protected int m_BagSizePercent = 100;
```

This is defined as **"The size of each bag sample, as a percentage of the training size"**. For each classifier, the train data is resampled with replacement (bootstrap sampling) to get the same number of elements as in the train data, meaning each train set size is the same as the original, but almost certainly will contain duplicates. However, if **`m_BagSizePercent`** is less than 100, only a proportion are retained.

Bagging can store the out of bag error. This is revisited in the evaluation lecture.

To classify new instances, it does not hold a vote with classifyInstance (as I thought). Rather, it sums the probability estimates from distributionForInstance, then normalises

```
public double[] distributionForInstance(Instance instance) throws Exception {

  double [] sums = new double [instance.numClasses()], newProbs;
  for (int i = 0; i < m_NumIterations; i++) {
    if (instance.classAttribute().isNumeric() == true) {
  sums[0] += m_Classifiers[i].classifyInstance(instance);
    } else {
  newProbs = m_Classifiers[i].distributionForInstance(instance);
  for (int j = 0; j < newProbs.length; j++)
    sums[j] += newProbs[j];
    }
```

Random Forest Classifier in Weka

For parameter info, see the lecture. This is more about the structure of the classifier. Random forest contains an instance of **`Bagging`** (the bagger) and a base classifier, hard coded to **`RandomTree`**, that is passed to the Bagger.

```
m_bagger = new Bagging();
RandomTree rTree = new RandomTree();
```

A RandomTree is a simple decision tree classifier. At each node it assesses **`m_KValue`** randomly selected attributes using information gain

```
protected double gain(double[][] dist, double priorVal) {

  return priorVal - ContingencyTables.entropyConditionedOnRows(dist);
}
```

The value **m_KValue** is dependent on the number of attributes in the data. For Weka, this is hard coded to be LOG_2(numAttributes)+1 (note difference to other implementations).

```
// set up the bagger and build the forest
m_bagger.setClassifier(rTree);
m_bagger.setSeed(m_randomSeed);
m_bagger.setNumIterations(m_numTrees);
m_bagger.setCalcOutOfBag(true);
m_bagger.setNumExecutionSlots(m_numExecutionSlots);
m_bagger.buildClassifier(data);
```

The bagger can be threaded by changing the number of execution slots to be > 1.

Bagging and Random Forest Exercises

Download the zip file UCIContinuous. This contains 33 problems from the UCR machine learning archive

https://archive.ics.uci.edu/ml/index.php

we used these in the experiments for CAWPE

https://link.springer.com/article/10.1007/s10618-019-00638-y

A list of these problems and simple code to load them (using tsml tools) is in the class ensembles_lab4.

Exercise 1: Compare default bagging to default random forest

Construct a Bagging and a RandomForest classifier using default settings on the train data, and save the accuracies on the test data for each problem to file. Compare them. Calculate the average accuracy of each classifier over all problems, the average difference in accuracy and the win/lose/draw count. Is one of them always better? Which seems to be better on average?

Exercise 2: Evaluate random forest for different number of trees

Repeat the experiment, but now do it for random forest with an increasing number of trees. Run the algorithms with 10, 50, 100, 200, …, 1000 trees (setNumberTrees or setNumberIterations, depending on version). Calculate the average **difference** between RF with 10 trees and with increasing numbers. Plot the average difference in excel. Does it improve? If so, does it stop improving? Try running with a stupid number (20,000 say). Does it break or take too long? Just kill it if it does.

Exercise 3: Compare random forest with 500 trees to J48

The usual default number of trees for RF is 500. Compare RF(500) with J48 (i.e. C4.5).

(I'm a little bit obsessed with classifier evaluation and comparison, only do as many of these as you think you need to in order to get a grasp of the usage of these classifiers)

Boosting: AdaBoost, LogitBoost and Gradient Boosting

Boosting involves building a sequence of classifiers, but rather than randomly sampling eah dataset, an iterative reweighting is used. The idea is that greater weight is placed on instances that were incorrectly classified at the previous stage.

In Weka, this can be done in one of two ways: by using the attribute in Instance called `m_Weight` or by weighted resampling. It all depends on the base classifier. Some Classifiers can use the weights for each classifier internally when building the model. These implement the (empty) interface **WeightedInstanceHandler**

```
public class DecisionStump
  extends AbstractClassifier
  implements WeightedInstancesHandler, Sourcable {
```

If the classifier is a **`WeightedInstanceHandler`**, the boosting algorithm simply adjusts the weights for the instances and relies on the classifier to use them. If not, it resamples the instances based on the weights.

```
 */
protected void buildClassifierWithWeights(Instances data)
        throws Exception {
```

This is the heart of the build algorithm for AdaBoostM1. It uses the Weka in **`Evaluation`** class.

```
    // Do boostrap iterations
    for (m_NumIterationsPerformed = 0; m_NumIterationsPerformed < m_Classifiers.length;
         m_NumIterationsPerformed++) {
        m_Classifiers[m_NumIterationsPerformed].buildClassifier(trainData);
        // Evaluate the classifier
        evaluation = new Evaluation(data);
        evaluation.evaluateModel(m_Classifiers[m_NumIterationsPerformed], training);
        epsilon = evaluation.errorRate();
        // Stop if error too small or error too big and ignore this model
        if (Utils.grOrEq(epsilon, b: 0.5) || Utils.eq(epsilon, b: 0)) {
            if (m_NumIterationsPerformed == 0) {
                m_NumIterationsPerformed = 1; // If we're the first we have to to use it
            }
            break;
        }
        // Determine the weight to assign to this model
        m_Betas[m_NumIterationsPerformed] = Math.log((1 - epsilon) / epsilon);
        reweight = (1 - epsilon) / epsilon;
        // Update instance weights
        setWeights(training, reweight);
    }
```

The weight setting is slightly different from the algorithm given in the lecture. It only adjusts the weights of the cases misclassified by the previous classifier. Note the use of Enumeration, a very old version of Iterator (which came in Java 1.2).

```
protected void setWeights(Instances training, double reweight)
        throws Exception {

    double oldSumOfWeights, newSumOfWeights;

    oldSumOfWeights = training.sumOfWeights();
    Enumeration enu = training.enumerateInstances();
    while (enu.hasMoreElements()) {
        Instance instance = (Instance) enu.nextElement();
        if (!Utils.eq(m_Classifiers[m_NumIterationsPerformed].classifyInstance(instance),
                instance.classValue()))
            instance.setWeight(instance.weight() * reweight);
    }

    // Renormalize weights
    newSumOfWeights = training.sumOfWeights();
    enu = training.enumerateInstances();
    while (enu.hasMoreElements()) {
        Instance instance = (Instance) enu.nextElement();
        instance.setWeight(instance.weight() * oldSumOfWeights
                / newSumOfWeights);
    }
}
```

Decision Stump

DecisionStump is a one level decision tree based on the best attribute to split on. It uses information gain to pick an attribute. It is a **`WeightedInstanceHandler`**. Sum of weights are used in the IG calculation instead of counts. This is the function to assess a nominal attribute with a nominal class

```
protected double findSplitNominalNominal(int index) throws Exception {

  double bestVal = Double.MAX_VALUE, currVal;
  double[][] counts = new double[m_Instances.attribute(index).numValues()
              + 1][m_Instances.numClasses()];
  double[] sumCounts = new double[m_Instances.numClasses()];
  double[][] bestDist = new double[3][m_Instances.numClasses()];
  int numMissing = 0;

  // Compute counts for all the values
  for (int i = 0; i < m_Instances.numInstances(); i++) {
    Instance inst = m_Instances.instance(i);
    if (inst.isMissing(index)) {
  numMissing++;
  counts[m_Instances.attribute(index).numValues()]
    [(int)inst.classValue()] += inst.weight();
    } else {
  counts[(int)inst.value(index)][(int)inst.classValue()] += inst
    .weight();
    }
  }
```


LogitBoost

```
public class LogitBoost
        extends RandomizableIteratedSingleClassifierEnhancer
        implements Sourcable, WeightedInstancesHandler, TechnicalInformationHandler {
```

Logistic boosting is more complex. The essential difference are that:

1) The problem is converted into a regression problem (predict a function of the probabilities of each class rather than the actual class value

```
    // Make class numeric
    data.setClassIndex(-1);
    data.deleteAttributeAt(classIndex);
    data.insertAttributeAt(new Attribute( attributeName: "'pseudo class'"), classIndex);
    data.setClassIndex(classIndex);
    m_NumericClassData = new Instances(data,  capacity: 0);
```

2) The target variable is altered at each iteration, so that each model is trying to correct the errors of the previous model.

```
            // Set values for instance
            Instance current = boostData.instance(i);
            current.setValue(boostData.classIndex(), z);
            current.setWeight(current.weight() * w);
        }
```

Gradient boosting (XGBoost) is not part of the default Weka distribution. There is a wrapped version in tsml called TunedXGBoost

```
 * @author James Large (james.large@uea.ac.uk)
 */
public class TunedXGBoost extends EnhancedAbstractClassifier implements SaveParameterInfo, DebugPrinting, SaveEachParameter, ParameterSplittable {

    //data info
```

This is capable of tuning parameters, but does not do so by default.

Exercise 4: Multiclass problems

I think there is a problem with Adaboost with multiclass problems. It has this condition

```
// Stop if error too small or error too big and ignore this model
if (Utils.grOrEq(epsilon, b: 0.5) || Utils.eq(epsilon, b: 0)) {
    if (m_NumIterationsPerformed == 0) {
        m_NumIterationsPerformed = 1; // If we're the first we have to to use it
    }
    break;
}
```

For problems with a large number of classes, ignoring models with error greater 0.5 may be wrong. Download the problem Adaic.arff from here

https://timeseriesclassification.com/description.php?Dataset=Adiac

Adiac is a 37 class problem. There is a default train test split in the download. Build a range classifiers you have used on this data. Does it seem Adaboost is particularly bad? Find out how many trees it discards inserting print statements in the above code.

Logit boost builds a separate model for each class. How well does it do on Adiac?

Exercise 5: Build your own ensemble

Thisd exercise is to build an ensemble from scratch, without using any built in tools. Implement an ensemble classifier that contains an array of J48 base classifiers. Diversify your ensemble by sampling 50% of the train data for each classifier (without replacement). Classify new instances with a simple majority vote. Compare your classifier to others you have used in this lab.

Ensembles in sci-kit learn

Scikit learn has good ensemble functionality, look here

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