Machine Learning Lab 6-8

By the end of this lab sheet you will have:

1. Become familiar with the inner workings of nearest neighbour Classifier IBk.
2. Learnt about linear classifiers in Weka.
3. Explored the multi-layer perceptron (MLP) classifier in Weka.
4. Explored the support vector machine (SVM) classifier in Weka.
5. Used the Weka Evaluation class to assess a classifier on a train set.
6. Performed a simple parameter grid search for MLP and SVM on the Football problem from lab 5 and/or the mosquito problem from lab 2.

Do not feel you have to complete this all in one lab – it's for weeks 6-8. People work at different paces, and I would rather you understand what you are doing than try race through. You will explore a lot of the inner code in Weka.

Prerequisite (you should have already done it for previous labs):

1. Download the tsml project from github
https://github.com/uea-machine-learning/tsml (please feel free to star it)
2. Move the download to somewhere sensible and unzip
3. In Intellij open the project by locating to the directory of the unzipped folder
4. Weka source code is in src/main/java/weka

Useful Resources

Weka API = http://weka.sourceforge.net/doc.dev/

1) IBk

We will now take a closer look at IBk classifier. It is more complex than you might imagine.

```
public class IBk
  extends AbstractClassifier
  implements OptionHandler, UpdateableClassifier, WeightedInstancesHandler,
             TechnicalInformationHandler, AdditionalMeasureProducer {
```

Find out what behaviour *UpdateableClassifier* enforces on a classifier. This capability is one of the reasons so much data cloning goes on in Weka. Look at the class attributes. Can you determine what they all mean? IBk contains a mechanism for searching.

```
/** for nearest-neighbor search. */
protected NearestNeighbourSearch m_NNSearch = new LinearNNSearch();
```

It is hard to figure out without seeing the execution of the algorithm. Go to *buildClassifier* to make it clearer.

```
// Throw away initial instances until within the specified window size
if ((m_WindowSize > 0) && (instances.numInstances() > m_WindowSize)) {
  m_Train = new Instances(m_Train,
                          m_Train.numInstances()-m_WindowSize,
                          m_WindowSize);
}
```

m_WindowSize is the max number of train cases allowed. It defaults to 0 (use all train cases), although never explicitly set to zero that I can see.

```
m_NumAttributesUsed = 0.0;
for (int i = 0; i < m_Train.numAttributes(); i++) {
  if ((i != m_Train.classIndex()) &&
      (m_Train.attribute(i).isNominal() ||
       m_Train.attribute(i).isNumeric())) {
    m_NumAttributesUsed += 1.0;
  }
}
```

This is saying only use nominal or numeric attributes. Basically ignores Date, String and Relational Attributes. This is only used in a method called *makeDistribution*, so the checks are repeated elsewhere no doubt. kNN is a lazy classifier, so setting the data for the search algorithm is all that happens in *buildClassifier*. Note IBk only implements *distributionForInstance*. If you do this, you do not have to implement *classifyInstance*, the default behaviour from *AbstractClassifier* is to call *distributionForInstance* and return the max probability. Have a look at *distributionForInstance*, it seems overly complex to me!

2) Linear Classifiers in Weka

Weka does not contain an implementation of Linear Discriminant Analysis. The closest it gets to a linear classifier are *SimpleLinearRegression*. However, this only works with numeric class attributes (i.e. regression problems)

```
run:
Exception in thread "main" weka.core.UnsupportedAttributeTypeException: weka.classifiers.functions.SimpleLinearRegression: Cannot handl
        at weka.core.Capabilities.test(Capabilities.java:936)
        at weka.core.Capabilities.test(Capabilities.java:1109)
        at weka.core.Capabilities.test(Capabilities.java:1022)
        at weka.core.Capabilities.testWithFail(Capabilities.java:1301)
```

This is set in *getCapabilities()*

```
// class
result.enable(Capability.NUMERIC_CLASS);
result.enable(Capability.DATE_CLASS);
result.enable(Capability.MISSING_CLASS_VALUES);
```

You can either manually change the arff file (change class attribute to numeric and change string labels of 0 and 1) or in code, but it is difficult and not really sensible. Linear regression is generally not that good an approach for classification and for many problems it is nonsensical. If, for example,

the classes are RED, GREEN,BLUE, converting them to 0,1,2 and fitting a linear regression is not very appropriate. For classification, stats folk would tend to use Logistic Regression for two class problems (class Logistic). Try it out.

3) MLP Classifier in Weka

The standard MLP weka is

weka.classifiers.functions.MultilayerPerceptron;

open the source code up and take a look at buildClassifier

```
 */
public void buildClassifier(Instances i) throws Exception {

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

This tests capabilities, i.e. tests whether the classifier works with this kind of data. What sort of data does MLP work with? Take a look. Pretty much all possible inputs, but not string attributes. It can be used for classification and regression. Now some weirdness, back to buildClassifier

```
  // remove instances with missing class
  i = new Instances(i);
  i.deleteWithMissingClass();

  m_ZeroR = new weka.classifiers.rules.ZeroR();
  m_ZeroR.buildClassifier(i);
  // only class? -> use ZeroR model
  if (i.numAttributes() == 1) {
    System.err.println(
        "Cannot build model (only class attribute present in data!), "
        + "using ZeroR model instead!");
    m_useDefaultModel = true;
    return;
  } else {
    m_useDefaultModel = false;
  }
```

Again, the data is cloned, removing cases with missing classes. The Capabilities say it can handle them, I guess this is how. The cloning of data can cause problems for large data, particularly if you are performing parameter searches (See later). It now, quite bizarrely, builds a ZeroR classifier (see previous lab sheet), then uses it if all the data is of one class. This could be simply handled by actually storing the single class value. As you saw previously, ZeroR also clones the data, so we now have three copies in memory! Anyway, onto the actual model ….

```
m_epoch = 0;
m_error = 0;
m_instances = null;
m_currentInstance = null;
m_controlPanel = null;
m_nodePanel = null;


m_outputs = new NeuralEnd[0];
m_inputs = new NeuralEnd[0];
m_numAttributes = 0;
m_numClasses = 0;
m_neuralNodes = new NeuralConnection[0];

m_selected = new FastVector(4);
m_graphers = new FastVector(2);
m_nextId = 0;
m_stopIt = true;
m_stopped = true;
m_accepted = false;
m_instances = new Instances(i);
m_random = new Random(m_randomSeed);
m_instances.randomize(m_random);
```

The underscore m indicates it is an attribute of the MLP object (rather than a local variable). The Netbeans font also indicates this. These are all the variables that control the structure. An MLP has an input layer and output layer of NeuralEnd objects and one or more connected layers of type NeuralConnection. These are set up a little below

```
//////////

setupInputs();

setupOutputs();
if (m_autoBuild) {
  setupHiddenLayer();
}
```

This sets up the correct network structure prior to learning the weights. There is an input node for each attribute and an output node for each class. The hidden layer structure is determined by the parameters and set in setupHiddenLayer.

So here is the input layer set up. The business with the variable now is to handle the situation where the class variable is not at the end of the attribute list (which it is for all our problems)

```
/**
 * This creates the required input units.
 */
private void setupInputs() throws Exception {
  m_inputs = new NeuralEnd[m_numAttributes];
  int now = 0;
  for (int noa = 0; noa < m_numAttributes+1; noa++) {
    if (m_instances.classIndex() != noa) {
      m_inputs[noa - now] = new NeuralEnd(m_instances.attribute(noa).name());

      m_inputs[noa - now].setX(.1);
      m_inputs[noa - now].setY((noa - now + 1.0) / (m_numAttributes + 1));
      m_inputs[noa - now].setLink(true, noa);
    }
    else {
      now = 1;
    }
  }
```

Ignore all the GUI code, it should not be here anyway in my opinion. Scrolling down buildClassifier we get to the bit where the training of network occurs. Without going into too much detail, this should seem vaguely familiar from the lectures.

```
  for (int noa = 1; noa < m_numEpochs + 1; noa++) {
    right = 0;
    for (int nob = numInVal; nob < m_instances.numInstances(); nob++) {
      m_currentInstance = m_instances.instance(nob);

      if (!m_currentInstance.classIsMissing()) {

        //this is where the network updating (and training occurs, for the
        //training set
        resetNetwork();
        calculateOutputs();
        tempRate = m_learningRate * m_currentInstance.weight();
        if (m_decay) {
          tempRate /= noa;
        }

        right += (calculateErrors() / m_instances.numClasses()) *
          m_currentInstance.weight();
        updateNetworkWeights(tempRate, m_momentum);

      }

    }
```

The rest is mostly filler, although the validation stage can be important. In terms of new instances, MLP only implements distributionForInstance. This is fine, because the AbstractClassifier classifyInstance is by default built on top of distributionForInstance (i.e. it calls it then returns the max value).

```
  */
public double[] distributionForInstance(Instance i) throws Exception {
```

The important part of distributionForInstance is below. ResetNetwork sets the output layer to zero. It is not immediately obvious froIn a fairly ugly way, MLP stores the instance in a global variable m_currentInstance. So the network is actually applied with the call to outputValue.

```
resetNetwork();

//since all the output values are needed.
//They are calculated manually here and the values collected.
double[] theArray = new double[m_numClasses];
for (int noa = 0; noa < m_numClasses; noa++) {
  theArray[noa] = m_outputs[noa].outputValue(true);
}
if (m_instances.classAttribute().isNumeric()) {
  return theArray;
}
```

```
public double outputValue(boolean calculate) {
```

With this recursive call being the main bit.

```
    //node is an output.
    m_unitValue = 0;
    for (int noa = 0; noa < m_numInputs; noa++) {
      m_unitValue += m_inputList[noa].outputValue(true);
```

MLP Parameters

So that at least gives an idea of the inner workings. How is it set up. The important parameters for MLP are:

1. The number of hidden layers (default to 1)

2. The number of hidden nodes per layer: This can be set to four values which are data dependent: ‘a’,’I’,’o’,’t’.

The number of hidden layers and nodes per layer is set through setting a single string m_hiddenLayers

```
/**
 * This will set what the hidden layers are made up of when auto build is
 * enabled. Note to have no hidden units, just put a single 0, Any more
 * 0's will indicate that the string is badly formed and make it unaccepted.
 * Negative numbers, and floats will do the same. There are also some
 * wildcards. These are 'a' = (number of attributes + number of classes) / 2,
 * 'i' = number of attributes, 'o' = number of classes, and 't' = number of
 * attributes + number of classes.
 * @param h A string with a comma seperated list of numbers. Each number is
 * the number of nodes to be on a hidden layer.
 */
public void setHiddenLayers(String h) {
```

Note the typos ☺ So, for example, if you want two hidden layers with “a” and “t” nodes respectivelty, you call this method with the string “a,t”

3. Learning rate: between 0 and 1, default is m_learningRate=0.3

4. Momentum: between 0 and 1, default is m_momentum = .2

5. Decay: true or false, default is false

6. Number of epochs: default is 500

Other MLP Implementations:

By far the most popular neural network softwares are TensorFlow (by google) and Theano, usually manipulated through Keras. You may have met these in the AI module. All are usually used in Python, although there seem to be some Java plugins.

https://en.wikipedia.org/wiki/Theano_(software)

https://en.wikipedia.org/wiki/Keras

https://www.tensorflow.org/

4) SVM Classifier in Weka

Support Vector Machines are implemented in the Weka class SMO

weka.classifiers.functions.SMO;

SMO stands for sequential minimal optimization, and describes the search technique for the support vectors. There is often a shed load of maths involved, but essentially an SVM has two components: a search technique for the support vectors (instances), and a kernel to measure the distance between two instances.

The Kernel

The kernels available all inherit from

weka.classifiers.functions.supportVector.Kernel;

In SMO , the kernel is stored in m_kernel

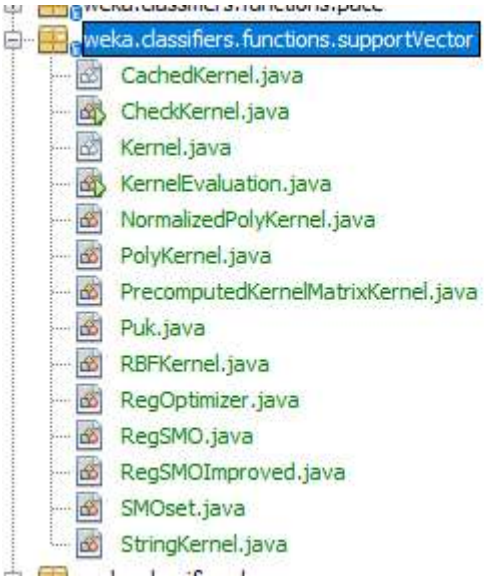
```
/** Kernel to use **/
protected Kernel m_kernel;
```

The kernel stores a reference to the dataset. The key operation for a kernel is the evaluation of the kernel between two instances.

```
/**
 * Computes the result of the kernel function for two instances.
 * If id1 == -1, eval use inst1 instead of an instance in the dataset.
 *
 * @param id1 the index of the first instance in the dataset
 * @param id2 the index of the second instance in the dataset
 * @param inst1 the instance corresponding to id1 (used if id1 == -1)
 * @return the result of the kernel function
 * @throws Exception if something goes wrong
 */
public abstract double eval(int id1, int id2, Instance inst1)
  throws Exception;
```

Unlike NN classifier distances, we pass IDs in the data rather than Instances. This is because the calculations may be repeated multiple times, and it is beneficial to cache them to improve efficiency.

The kernel package is



But the only ones of real interest are PolyKernel and RBFKernel. There is a lot of code in these classes, but to cut through a lot of it, the key method is evaluate. This is the PolyKernel evaluate

```
protected double evaluate(int id1, int id2, Instance inst1)
  throws Exception {

  double result;
  if (id1 == id2) {
    result = dotProd(inst1, inst1);
  } else {
    result = dotProd(inst1, m_data.instance(id2));
  }
  // Use lower order terms?
  if (m_lowerOrder) {
    result += 1.0;
  }
  if (m_exponent != 1.0) {
    result = Math.pow(result, m_exponent);
  }
  return result;
}
```

The most important parameter is the m_exponent. If it is set to 1 you get a linear SVM. Set to 2 or higher it is a polynomial SVM

The dot product is simply the sum of the attributes multiplied together, e.g.

```
protected final double dotProd(Instance inst1, Instance inst2)
  throws Exception {

  double result = 0;

  // we can do a fast dot product
  int n1 = inst1.numValues();
  int n2 = inst2.numValues();
  int classIndex = m_data.classIndex();
  for (int p1 = 0, p2 = 0; p1 < n1 && p2 < n2;) {
    int ind1 = inst1.index(p1);
    int ind2 = inst2.index(p2);
    if (ind1 == ind2) {
      if (ind1 != classIndex) {
        result += inst1.valueSparse(p1) * inst2.valueSparse(p2);
      }
      p1++;
      p2++;
    } else if (ind1 > ind2) {
      p2++;
    } else {
      p1++;
    }
  }
  return (result);
}
```

RBFKernel also uses dotproduct, but then does a different calculation

```
    double result = Math.exp(m_gamma
                             * (2. * dotProd(inst1, inst2) - precalc1 - m_kernelPrecalc[id2]));
```

So these values are used by the SMO problem to find the best support vectors by solving an optimisation problem. Without the maths, the problem is to find a set of cases that minimise some function of the evaluation of the support vectors. So to SMO we go, into buildClassifier …

SMO

Normal sort of preamble, then the first interesting thing to note is this

```
// Build the binary classifiers
Random rand = new Random(m_randomSeed);
m_classifiers = new BinarySMO[insts.numClasses()][insts.numClasses()];
for (int i = 0; i < insts.numClasses(); i++) {
  for (int j = i + 1; j < insts.numClasses(); j++) {
    m_classifiers[i][j] = new BinarySMO();
    m_classifiers[i][j].setKernel(Kernel.makeCopy(getKernel()));
    Instances data = new Instances(insts, insts.numInstances());
    for (int k = 0; k < subsets[i].numInstances(); k++) {
      data.add(subsets[i].instance(k));
    }
    for (int k = 0; k < subsets[j].numInstances(); k++) {
      data.add(subsets[j].instance(k));
    }
    data.compactify();
    data.randomize(rand);
    m_classifiers[i][j].buildClassifier(data, i, j,
                                        m_fitLogisticModels,
                                        m_numFolds, m_randomSeed);
  }
}
```

So SMO builds a classifier for each class vs each other classes. i.e. it implements a one-against-one wrapper for multiclass problems. This means it constructs numClasses*(numClasses-1)/2 classifiers.

This may be slow for problems with many classes. But whats BinarySMO? Lets hope it doesn't clone the data …..

```
/**
 * Class for building a binary support vector machine.
 */
public class BinarySMO
  implements Serializable {
```

It doesn't! It is an inner class of SMO, so can access all the attributes of the outer class (remember programming 2?). Also note it does not implement the Classifier interface. How does it work? Well this is where it gets hairy. This is the start of buildClassifier

```
protected void buildClassifier(Instances insts, int cl1, int cl2,
                               boolean fitLogistic, int numFolds,
                               int randomSeed) throws Exception {

 // Initialize some variables
 m_bUp = -1; m_bLow = 1; m_b = 0;
 m_alpha = null; m_data = null; m_weights = null; m_errors = null;
 m_logistic = null; m_I0 = null; m_I1 = null; m_I2 = null;
 m_I3 = null; m_I4 = null; m_sparseWeights = null; m_sparseIndices = null;
```

I think we will say goodbye to SMO here, or rather, I leave it as an exercise to work out what it all means ☺

SVM Parameters

SVM are very sensitive to parameters and should be tuned for best results.

The regularisation parameter, C, is used in the SMO to control how much to penalise solutions with a large number of support vectors. Defaults to 1. Range of values 0 to very large. We normally search on an exponential range 2^{-4}, 2^{-3} …. 2^{4}. Higher C means more penalty (regularisation)

```
/** The complexity parameter. */
protected double m_C = 1.0;
```

Kernel: generally either **RBFKernel or PolyKernel**

```
public void setKernel(Kernel value) {
  m_kernel = value;
}
```

RBF Kernel (PARTICULARLY needs tuning). Parameter gamma used in the evaluation. Same range as C, defaults to 0.01.

```
double result = Math.exp(m_gamma
                         * (2. * dotProd(inst1, inst2) - precalc1 - m_kernelPrecalc[id2]));
```

PolyKernel. Parameter exponent used in the evaluation. Range 1,2, …… default 1 (linearSVM)

```
  result = Math.pow(result, m_exponent);
}
```

Other SVM Implementations:

LIBSVM is the most popular SVM library, written in C.

https://www.csie.ntu.edu.tw/~cjlin/libsvm/

It is easly wrappable to use in Weka

http://weka.sourceforge.net/doc.stable/weka/classifiers/functions/LibSVM.html

We have found little difference between it and the Weka SMO.

5) Weka Evaluation Class

We want to be able to evaluate a classifier on training data so that we can set the parameters. **Parameters should always be set using the training data only.** The basic idea is to perform a **cross validation** to estimate the accuracy/error. Read this or similar article

https://www.geeksforgeeks.org/cross-validation-machine-learning/

In Weka, you can evaluate a classifier on a training data set with the Evaluator class

weka.classifiers.evaluation

```
 */
public class Evaluation implements Summarizable, RevisionHandler, Serializable {
```

Use the mosquito data from the previous lab to test this out. The basic usage is this

```
Evaluation eval= new Evaluation(train);
eval.crossValidateModel(zero, train, 10, new Random());
double error=eval.errorRate();
System.out.println("Error rate for ZeroR on train CV ="+error);
```

Test out SMO and MultiLayerPerceptron with the default values. For SMO, also evaluate with a quadratic SMO (exponent =2) and RBFKernel. Evaluation can also be used to get summary measures on the test data.

```
knn.buildClassifier(train);
Evaluation eval= new Evaluation(train);
eval.evaluateModel(knn, test);
```

This creates a whole shed load of performance measures. Call getMetricsToDisplay to find out which.

6) Grid Search on the Train Data

Tuning a classifier is finding parameter values that we think will give better results on unseen data (i.e. the test data). The idea is we try many combinations of parameters, evaluate them on

the train data, and pick the parameters with the lowest error. Use the mosquito data to test this out.

SVM and MLP are two classifiers that particularly benefit from tuning.

1. Write a method to assess SVM with an RBF kernel on a total of 25 parameter combinations,

Gamma = {0.001, 0.01, 0.1, 1, 10}

C = {0.001, 0.01, 0.1, 1, 10}

And record the combination with the lowest error.

2. Try a wider range of parameter values, say

Gamma = {2^{-5}, 2^{-4}, …, 2^{4}, 2^{5}}

C = {0.001, 0.01, 0.1, 1, 10}

If it is taking too long, just stop and move on.

3. Now repeat the exercise for a polynomial kernel and multilayer perceptron.

4. Note that if you use a Random(), the results may not be the same each time.

5. Now compare default classifiers with tuned values on the test data. Did tuning improve performance?