Assignment 2: Data Mining and Machine Learning

2ID90/2IDC0 Q3, 2013-2014

March 14, 2014

1 The Data

In this assignment you are going to build classifiers for recognizing handwritten digits. These digits are taken from the online MNIST database at http://yann.lecun.com/exdb/mnist; some example digits are shown in Figure 1.

The database consists of 60.000 training patterns and 10.000 test patterns. More information on the data can be found at the given URL. Sample java code for reading the datasets as well as the data itself will be provided.

Figure 1: some labeled handwritten digits in the MNIST database.

2 K-nearest neighbour Classifier

1. As a first classifier you are asked to construct a k-NN classifier. To do so you are asked to implement the `Classifier` interface, as specified below. This interface is part of a java package[1] that will be provided. Initially, the `getConfusionMatrix` method may simply return null.

```
public interface Classifier<V extends Features, Target> {
  /** returns the classification of feature vector v. */
  Target classify(V v);

  /**
   * @return errorRate of this classifier for the testData.
   */
  double errorRate(Dataset<V, Target> testData);

  /** returns a map m such that m.get(a).get(b) is the
   * number of times target a has been recognized as b.
   * @param testData dataset for which matrix is computed
   * @return confusion matrix
   */
  Map<Target,Map<Target,Integer>>
  getConfusionMatrix(Dataset<V,Target> testData);
}
```

[1] Class `KNN` in package `nl.tue.s2id90.classification.knn` has an empty implementation of this interface. You may adapt this class to fit your needs.

2. Run the k-NN classifier with the given training and test patterns[2]. Note down the resulting errorRate for different values of k. Analyze the behaviour of this classifier by looking at the confusion matrix.[3]

[2] See section 4.1 for instructions on how to read the MNIST database.

[3] If you implement the getConfusionMatrix method of Classifier, you can visualize the confusion matrix using method showIt of ConfusionMatrixPanel.

3 Decision tree classifier

1. Next you are going to build a decision tree classifier.

   (a) As a preparation step implement and test the following class.

```java
public class Information {
  /**
   * @param p probability distribution
   * @return the entropy of p.
   **/
  public static double entropy(double ... p){
    // YOUR IMPLEMENTATION
  }
  /**
   * @param x argument
   * @return  2log(x)
   **/
    public static double log2(double x){
    // YOUR IMPLEMENTATION
  }
  /**
   * @param x argument
   * @return  x 2log(x)
   **/
  public static double xlog2x(double x) {
    // YOUR IMPLEMENTATION
  }
}
```

(b) The next step is to write a program to build a decision tree for discrete variables only. See section**??** for a description of helpful provided software. Eventually,

i. test on toy data[4];

ii. test on digit database; discuss your results.

[4] There are two toy datasets; see section 4.2 on how to access them.

(c) Implement pruning of a decision tree and apply to the MNIST tree.

2. Build decision tree with continuous variables

   (a) decide on granularity of data (speed vs accuracy?)

3. Build several decision trees from (randomly?) chosen subsets of the training data. Combine their classification results to build a boosted classifier. Optimize your results by choosing proper parameter settings.

4. Compute additional features to further improve the error rate.

4 *Software*

4.1 *Read MNIST database*

For reading the MNIST database (static) methods in the class `HandWrittenDigits` are used. The data read from the database consists of labeled images, which have image data encoded in unsigned bytes and a classification label encoded in a byte.

```
List<LabeledImage> trainingImages, testImages;
//read 15000 training records and shuffle them
trainingImages = HandWrittenDigits.getTrainingData(15000,true);

//read test data
testImages = HandWrittenDigits.getTestData();
```

The resulting lists of images may be viewed using an auxiliary method in DigitsUtil. The next line of code shows the first 25 test images in a 5 column grid of images; see Figure 2.

```
DigitsUtil.showImages("test images",testImages.subList(0,25),5)
```

Figure 2: The first 25 test images in the MNIST database.

For usage in the Classifier interface this data has to be reordered in a Map that maps features (here of type `ImageFeatures<Double>`) on labels (here of type `Byte`).

```
Map<ImageFeatures<Double>, Byte> trainingDataset, testDataset;
trainingDataset= new HashMap<>();
for (LabeledImage image : trainingImages) {
  trainingDataset.put(new Doubles(image), image.getLabel());
}

testDataset = new HashMap<>();
for (LabeledImage image : testImages) {
  testData.put(new Doubles(image), image.getLabel());
}
```

For a subtype F of Features and a label type L, a typical usage of a `labeled` dataset is given below.

```
Map<F extends Features, L> dataset;
...
for (Entry<F,L> train : dataset.entrySet()) {
  F f = train.getKey());
  L label = train.getValue()
  ...
}

// or alternatively
for (F train : dataset.keySet()) {
  L label = trainingDataset.get(F);
  ...
}
```

Computing the distance between two ImageFeatures might be done as follows.

```
public double distance(
  ImageFeatures<Double> fo, ImageFeatures<Double> f1
){
  Double[] ao = fo.getValues(), a1 = f1.getValues();
  final int n = fo.getNumberOfAttributes(); double d = 0;
  for (int i = 0; i < n; i = i + 1)
    double h = ao[i] - a1[i]; d += h * h;
  return d;
}
```

Alternatively, you may use fo.get(i) to get the value of the i-th feature of fo.

4.2 *Toy data*

There are two toy datasets: SkiData and GolfData. They can be used as in the following code fragment; the resulting tree is shown in Figure 3.

```
// create a proper dataset for decisiontree building
public static LabeledDataset2<Features,GolfData.PLAY>
  getDataset(GolfData data)
{
 LabeledDataset2<Features,GolfData.PLAY> set;
 set = new MyLabeledDataset2();
 set.putAll(data.getClassification());
 return set;
}

// build decision tree and show it on screen
public static void main(String[] argv) {
 GolfData d = new GolfData();
 LabeledDataset2<Features,GolfData.PLAY> dataset=getDataset(d);
 DecisionTree<Features,GolfData.PLAY> dt;
 dt = new MyDecisionTree<>(dataset);
 DotUtil.showDotInFrame(dt.toDot(),"golf data");
}
```

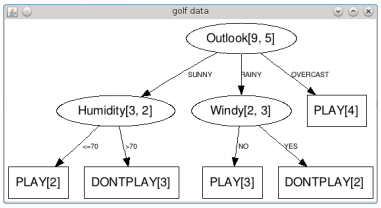

Figure 3: Golf data decision tree created using an online GraphViz server. Note that large trees are not accepted by this server.

4.3 *class LabeledDataset*

In order to program your own decision tree builder you are given several classes to give you a quick start. In this section and the following sections the most important ones are discussed.

The class LabeledDataset is a container for feature vectors and their labels. It is fully implemented. Some of its methods are:

- getLabel(f) that returns the label of feature vector f;
- getFeatureVectors() that returns a list of all feature vectors in the dataset;
- getLabels() that returns a set of all labels in the dataset.

4.4 *class LabeledDataset2*

For decision tree building it is handy to have a mapping from a label to feature vectors with that label. This functionality is supplied by LabeledDataset2 in methods like:

- getFeatures(L label) that returns all feature vectors with classification label.

The class LabeledDataset2 also contains several methods that you have to implement for building decision trees; for example:

```
/** splits dataset in subsets with a constant value
 * for the i-th feature.
 * @return result of splitting, the keys in this map are
 * the values of the i-th attribute.
 **/
public Map<Object,LabeledDataset2<V, L>> discreteSplit(int i)
{ ... }

/** returns the probabilities of the classification labels
 *  for this dataset.
 *  @return probability of the labels.
**/
public double[] classProbabilities() { ... }

/**
 * @param index index of attribute used for discrete splitting.
 * @return gain by discrete splitting on attribute index.
 */
public double gain(int index) { ... }
...
```

4.5 *class DecisionTree*

A decision tree is a tree with labels, corrresponding to attribute values, on the edges[5]. For this functionality class LabeledTree has been implemented. It contains functionality, among others, for inspecting the labels and walking the tree. The class DecisionTree inherits from LabeledTree and uses Objects as edge labels (that is , any possible attribute value). Furthermore, each node in the tree contains a labeled dataset with feature vectors (comparable to Figure 8.23 in Ertel where per node in the tree the feature vectors are mentioned). For leaf nodes the DecisionTree contains a classification label (yes/no in Ertel's example, PLAY/DON'TPLAY in the Golf example of Figure 3). The DecisionTree class has several methods that still need to be implemented, for instance:

[5] Note that this label differs from the label in LabeledDataset. The latter one is a classification label, that is, it is the name of a class as it appears in the leaf of a decision tree.

```
// implementation of interface Classifier's method
public L classify(F v) { ... }

/**
 * called from constructor of DecisionTree, this method should
 *  do the actual recursive building of the decision tree.
 */
protected void build(LabeledDataset2<F,L> dataset) { ... }

public int prune(final LabeledDataset2<F,L> testData) { ...}

/**
 *compute the errorRate of this decision tree for testData.
 */
public double errorRate(Map<F, L> testData) { ... }
```