Improving Probability Estimation Trees for Ranking

Rivka Levitan, Haimonti Dutta

Center for Computational Learning Systems at Columbia University, New York

rlevitan@cs.columbia.edu, haimonti@ccls.columbia.edu

Abstract

We suggest an algorithm to improve the performance of probability estimation trees in producing class probability predictions for ranking, specifically by reducing the occurrence of ties. To more accurately reflect the probability of class predictions, we weight each leaf by the proportion of training instances that fall within it. To eliminate the problem of ties, we implement a kernel density estimate at the leaf.

1. Introduction

Decision tree learning algorithms are commonly used in machine learning for classification problems. A tree is defined as a set of logical conditions on attributes; a leaf represents the subset of instances corresponding to the conjunction of conditions along its branch, or path back to the root. An instance being classified is passed along the tree to a leaf and is assigned the majority class label of that leaf.

For many applications, it is useful to order instances, which involves assigning them a rank, rather than a class label. For example, a web-page recommender may want to order web pages by the likelihood of their being of interest to the user, instead of classifying them as "of interest" and "not of interest." A simple approach to ranking is to estimate the probability of an instance's membership in a class, and assign that probability as the instance's rank. A decision tree can easily be used to estimate these probabilities. If a leaf node contains class frequencies $n_1, n_2, \ldots, n_c$, the probability that an instance falling in that leaf belongs to class i can be defined as $\frac{n_i}{\sum n_i}$.

Decision trees acting as probability estimators, however, are often observed to produce bad probability estimates. Specifically, every instance in a node is assigned the same probability, resulting in a proliferation of ties, which reduces the results' usefulness for ranking. Sparse training sets may also lead to skewed estimates. Most developments in decision tree learning algorithms have aimed at improving classification accuracy rather than probability estimates. We explore two techniques for producing improved probability estimates.

2. Prior Work

Cohen et al. [1] discuss learning to order based on feedback in the form of preference judgments, or statements that one instance should be ranked above another, rather than probabilities of class membership, because preference information may be more natural and easier to obtain than the information necessary for classification. Cao et al. [2] present the problem of ranking as a listwise rather than pairwise problem, and employ Neural Network and Gradient Descent as model and algorithm in their learning method.

Provost and Domingos [3], however, defend the use of probability estimation trees, noting that the inaccurate probabilities are at least partially the result of algorithms that maximize

classification accuracy and minimize tree size, while larger trees are better for calculating probabilities. They show that two simple methods, Laplace correction and bagging, substantially improve probability-based ranking, and that pruning degrades it.

Ferri et al. [4] show that standard splitting criteria aimed at increasing classification accuracy do not necessarily produce good probability estimates. They suggest a new splitting criterion based on probability ranking, the AUC-splitting criterion.

Smyth et al. [5] suggest a new smoothing method that considers all the frequencies from root to leaf, a new splitting criterion based on the minimum squared error of the probability estimates, and a pruning criterion that can reduce the size of a tree without degrading the quality of the probability estimates.

Existing work has focused on improving the efficiency and accuracy of probability estimates produced by decision trees. However, little has been done to address the problem of dealing with ties among instances, making probability estimation trees difficult for use as rankers. Our work focuses on this issue.

3. Weighted Splits

To increase the accuracy of the predictions, we weight each node by the proportion of instances it contains. The intuition is that if a large number of instances fall in a node, the classification rule it represents is likely to be significant, and the probability estimates should definitely be higher. The weight of a node i with cardinality c_i is defined as

$$w_i = w_{parent(i)} * \frac{c_i}{c_{parent(i)}}$$

Greater significance is therefore given to splits involving large proportions of instances, while splits involving small proportions are devalued accordingly. An example is given in Figure 1.

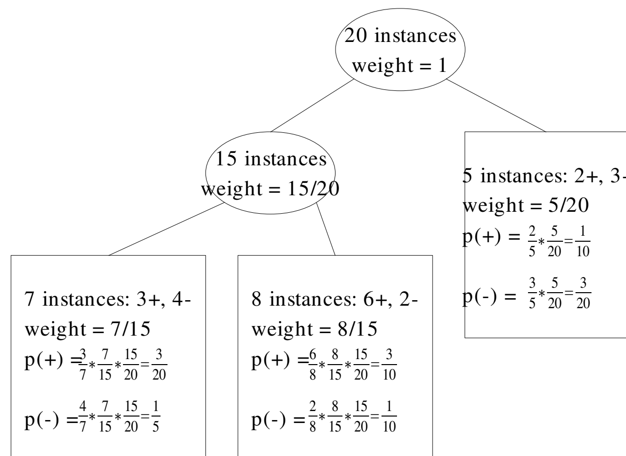

Figure 1

4. Kernel Density Estimation

One problem with calculating the estimated probability as *n/N*, where n is the number of instances of the specified class at a leaf and N is the total number of instances, is that pure nodes with a small number of instances will be assigned the same probability as pure nodes with a large number of instances. This is especially problematic for ranking, since it will result in many ties, and instances with the same probability can not be ordered.

The problem is usually addressed by smoothing the probability estimates to less extreme values. The most commonly used smoothing method is the Laplace estimate, which calculates the expected probability as

$$\frac{n+1}{N+C}$$

where C is the total number of classes. In effect, it incorporates a prior probability of $1/C$ for each class. The use of the Laplace estimate is widespread and has been found to be extremely effective in improving classification performance.

Another smoothing method is *m*-estimate, which is defined as

$$\frac{n+m \times p}{N+m}$$

The probability p is the expected probability without prior knowledge, and is either

calculated as *1/C* or estimated from the training data.

While the Laplace and *m* corrections have been shown to significantly improve classification performance, we found that they do little to resolve ties in the probability estimates. We implemented the *m*-estimate in Weka [6]. We examined the results of the J48 decision tree on the Spambase dataset from the UCI Machine Learning Repository [7], using Laplace, *m*-estimate, and no smoothing.

We found that while smoothing did reduce the occurrence of ties, the presence of ties was still significant. For 1380 instances, no smoothing resulted in only 22 distinct probability estimates. Applying the Laplace estimation resulted in 37 distinct values, while the *m*-estimate yielded 38. These results, although they do represent a slight improvement, clearly demonstrate the need for a better mechanism to resolve ties. To address this issue, we apply a kernel density estimation at the leaf, as described by Smyth et al. [5].

First, a kernel density bandwidth estimation method is run on the training data to select bandwidths h for each of the attributes k and for each of the classes w_j. A decision tree is then generated from the training data using a standard decision tree learning algorithm.

This decision tree can then be used to calculate the class probability prediction for a test instance in the following way: The test instance is passed down the tree to a leaf. At the leaf, a local density estimate is generated for each class:

$$\hat{f}(\underline{x} \mid w_j) = \frac{1}{N_j} \sum_{i=1}^{N_j} \prod_{k \in \text{path}} \frac{1}{h_k} K\left(\frac{x^k - x_i^k}{h_k}\right)$$

N_j is the number of training points belonging to class w_j. For each of these training points, the product is taken only over the attributes tested in the path from the root to the leaf.

The class probabilities are then estimated using Bayes' rule and the density estimates:

$$\hat{p}(w_j \mid x) = \frac{\hat{f}_j(\underline{x})\, p(w_j)}{\sum_1^m \hat{f}_i(\underline{x})\, p(w_i)}, 1 \leqslant j \leqslant m$$

$p(w_j)$ represents the prior probabilities of each class, estimated from the data in the usual way.

5. Conclusion

We present an algorithm for improving the probability estimates generated by decision trees. Our method reflects the probability of each class prediction more accurately by weighting each leaf with the proportions of the instances that fall in it. Furthermore, we reduce the occurrence of ties by implementing a kernel density estimation at the leaf. This is extremely important for ranking.

Our algorithm learns a standard decision tree, storing information about the proportions of each attribute split and weighting the leaves recursively with these proportions. When an instance is passed to a leaf, the instance's attributes are used to generate a kernel density estimation, which is used to calculate its probabilities of class membership. The algorithm should produce probability estimates that are more suitable for ranking, because they reflect the data more accurately and are not susceptible to ties.

We implemented our algorithm in Weka. The code is attached as Appendix A. We are continuing to implement the code so that we can test the algorithm on datasets from the UCI Machine Learning Repository, in order to obtain empirical results for the performance of the algorithm.

References

[1] Cohen, W.; Schapire, R.; Singer, Y.: "Learning to Order Things," *Journal of Artificial Intelligence Research 10 (1999) 243-270*, 1999.

[2] Cao, Z. et al.: "Learning to Rank: From Pairwise Approach to Listwise Approach," Microsoft technique report, 2007.

[3] Provost, F.; Domingos, P.: "Tree Induction for Probability-based Ranking," *Machine Learning* 52:3, pp. 199-215, 2003.

[4] Ferri, C., Flach, P., & Hernández-Orallo, J. : "Learning Decision Trees using the Area Under the ROC Curve," in C. Sammut; A. Hoffman (eds.) "The 2002 International Conference on Machine Learning" (ICML2002), Morgan Kaufmann, pp. 139-146, 2002.

[5] Smyth, P.; Gray, A.; Gray, E.; Fayyad, U. "Retrofitting Decision Tree Classifiers using Kernel Density Estimation." Morgan Kaufmann, San Francisco, CA 1995.

Appendix A:

Code:

We modified WEKA's ClassifierTree to apply an M-estimate and to implement our algorithm. Our changes and insertions are marked with "Rivka" in the comments.

```
/*
 *    This program is free software; you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation; either version 2 of the License, or
 *    (at your option) any later version.
 *
 *    This program is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *
 *    You should have received a copy of the GNU General Public License
 *    along with this program; if not, write to the Free Software
 *    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 *    ClassifierSplitModel.java
 *    Copyright (C) 1999 University of Waikato, Hamilton, New Zealand
 *
 */

package weka.classifiers.trees.j48;

import weka.core.Instance;
import weka.core.Instances;
import weka.core.RevisionHandler;
import weka.core.Utils;

import java.io.Serializable;

/**
 * Abstract class for classification models that can be used
 * recursively to split the data.
 *
 * @author Eibe Frank (eibe@cs.waikato.ac.nz)
 * @version $Revision: 1.11 $
 */
public abstract class ClassifierSplitModel
implements Cloneable, Serializable, RevisionHandler {

      /** for serialization */
      private static final long serialVersionUID = 4280730118393457457L;

      /** Distribution of class values. */
      protected Distribution m_distribution;
```

```
/** Number of created subsets. */
protected int m_numSubsets;

/**
 * Allows to clone a model (shallow copy).
 */
public Object clone() {

 Object clone = null;

 try {
       clone = super.clone();
 } catch (CloneNotSupportedException e) {
 }
 return clone;
}

/**
 * Builds the classifier split model for the given set of instances.
 *
 * @exception Exception if something goes wrong
 */
public abstract void buildClassifier(Instances instances) throws Exception;

/**
 * Checks if generated model is valid.
 */
public final boolean checkModel() {

 if (m_numSubsets > 0)
       return true;
 else
       return false;
}

/**
 * Classifies a given instance.
 *
 * @exception Exception if something goes wrong
 */
public final double classifyInstance(Instance instance)
throws Exception {

 int theSubset;

 theSubset = whichSubset(instance);
 if (theSubset > -1)
       return (double)m_distribution.maxClass(theSubset);
 else
       return (double)m_distribution.maxClass();
}

/**
 * Gets class probability for instance.
 *
 * @exception Exception if something goes wrong
```

```
 */
public double classProb(int classIndex, Instance instance, int theSubset)
throws Exception {

 if (theSubset > -1) {
       return m_distribution.prob(classIndex,theSubset);
 } else {
       double [] weights = weights(instance);
       if (weights == null) {
             return m_distribution.prob(classIndex);
       } else {
             double prob = 0;
             for (int i = 0; i < weights.length; i++) {
                   prob += weights[i] * m_distribution.prob(classIndex, i);
             }
             return prob;
       }
 }
}

/**
 * Gets class probability for instance.
 *
 * @exception Exception if something goes wrong
 */
public double classProbLaplace(int classIndex, Instance instance,
       int theSubset) throws Exception {
//     System.out.println("classProbLaplace called in ClassifierSplitModel");
 if (theSubset > -1) {
       return m_distribution.laplaceProb(classIndex, theSubset);
 } else {
       double [] weights = weights(instance);
       if (weights == null) {
             return m_distribution.laplaceProb(classIndex);
       } else {
             double prob = 0;
             for (int i = 0; i < weights.length; i++) {
                   prob += weights[i] * m_distribution.laplaceProb(classIndex,
i);
             }
             return prob;
       }
 }
}

public double classProbM(int classIndex, Instance instance, int theSubset)
throws Exception {

 if (theSubset > -1) {
       return m_distribution.mProb(classIndex, theSubset, 4);
 } else {
       double [] weights = weights(instance);
       if (weights == null) {
             return m_distribution.mProb(classIndex, 4);
       } else {
             double prob = 0;
```

```
                for(int i=0; i<weights.length; i++) {
                     prob += weights[i] * m_distribution.mProb(classIndex, i,
4);
                }
                return prob;
          }
     }

     }






     /**
      * Returns coding costs of model. Returns 0 if not overwritten.
      */
     public double codingCost() {

     return 0;
     }

     /**
      * Returns the distribution of class values induced by the model.
      */
     public final Distribution distribution() {

     return m_distribution;
     }

     /**
      * Prints left side of condition satisfied by instances.
      *
      * @param data the data.
      */
     public abstract String leftSide(Instances data);

     /**
      * Prints left side of condition satisfied by instances in subset index.
      */
     public abstract String rightSide(int index,Instances data);

     /**
      * Prints label for subset index of instances (eg class).
      *
      * @exception Exception if something goes wrong
      */
     public final String dumpLabel(int index,Instances data) throws Exception {

     StringBuffer text;

     text = new StringBuffer();
     text.append(((Instances)data).classAttribute().
                value(m_distribution.maxClass(index)));
     text.append(" ("+Utils.roundDouble(m_distribution.perBag(index),2));
```

```
 if (Utils.gr(m_distribution.numIncorrect(index),0))

text.append("/"+Utils.roundDouble(m_distribution.numIncorrect(index),2));
 text.append(")");

 return text.toString();
}

public final String sourceClass(int index, Instances data) throws Exception {

 System.err.println("sourceClass");
 return (new StringBuffer(m_distribution.maxClass(index))).toString();
}

public abstract String sourceExpression(int index, Instances data);

/**
 * Prints the split model.
 *
 * @exception Exception if something goes wrong
 */
public final String dumpModel(Instances data) throws Exception {

 StringBuffer text;
 int i;

 text = new StringBuffer();
 for (i=0;i<m_numSubsets;i++) {
       text.append(leftSide(data)+rightSide(i,data)+": ");
       text.append(dumpLabel(i,data)+"\n");
 }
 return text.toString();
}

/**
 * Returns the number of created subsets for the split.
 */
public final int numSubsets() {

 return m_numSubsets;
}

/**
 * Sets distribution associated with model.
 */
public void resetDistribution(Instances data) throws Exception {

 m_distribution = new Distribution(data, this);
}

/**
 * Splits the given set of instances into subsets.
 *
 * @exception Exception if something goes wrong
 */
public final Instances [] split(Instances data)
```

```
    throws Exception {

    Instances [] instances = new Instances [m_numSubsets];
    double [] weights;
    double newWeight;
    Instance instance;
    int subset, i, j;

    for (j=0;j<m_numSubsets;j++)
          instances[j] = new Instances((Instances)data,
                      data.numInstances());
    for (i = 0; i < data.numInstances(); i++) {
          instance = ((Instances) data).instance(i);
          weights = weights(instance);
          subset = whichSubset(instance);
          if (subset > -1)
                instances[subset].add(instance);
          else
                for (j = 0; j < m_numSubsets; j++)
                      if (Utils.gr(weights[j],0)) {
                            newWeight = weights[j]*instance.weight();
                            instances[j].add(instance);
                            instances[j].lastInstance().setWeight(newWeight);
                      }
    }
    for (j = 0; j < m_numSubsets; j++)
          instances[j].compactify();

    return instances;
    }

    /**
     * Returns weights if instance is assigned to more than one subset.
     * Returns null if instance is only assigned to one subset.
     */
    public abstract double [] weights(Instance instance);

    /**
     * Returns index of subset instance is assigned to.
     * Returns -1 if instance is assigned to more than one subset.
     *
     * @exception Exception if something goes wrong
     */
    public abstract int whichSubset(Instance instance) throws Exception;
}
```