Homework 6

Decision Trees

The different parts of this assignment all do the same thing: ask you to separate two different samples from each other. You will be given two *training samples*, which are categorized as *signal* and *background*. The signal is the thing you want; the background is the stuff you don't want. Thus it is important to find the difference.

In most real cases, it is not possible to perfectly categorize each event. However, you can separate two events statistically, so that you can get most of the signal separate from most of the background. The approach here is to use a machine-learning algorithm. You are going to *train* an algorithm to separate signal from background, and then see how well it performs.

The algorithm we use is called a *decision tree*. It is perhaps the simplest machine-learning algorithm in common use, and in its boosted form it is also one of the most powerful.

Level I

You can do all your work in the DecisionTree project provided on GitHub. It is already set up with the files that you need and you should be able to run it "out of the box" without any problems.

A DataSet is a collection of DataPoints. Each DataPoint contains a set of variables that are recorded for a given particle physics event. The actual identity of the variables is recorded in the DataSet in the Names field; however, for this assignment, the actual meaning of the variables is unimportant. These variables have been selected as likely to help distinguish signal events from background events.

Your job is to choose a variable that best separates signal from background. You are trying to find a value of this variable such that if you make a cut at that point, most of the signal is on one side and most of the background is the other side. Write some code that chooses one variable and finds one value that does the best job of separating the events.

There is already code that will produce an output file. It is formatted like this:

Event	Purity
0	1
1	0
2	1

Where 0 is signal and 1 is background. Turn this in as a text file.

Level II

A true decision tree does this exercise not just once, but many times. For each resulting sample, you should repeat the same exercise, testing each variable and choosing the one that does the best job of separating signal from background. When a sample cannot be separated any further, then stop. Each final leaf can be assigned a value equal to the purity of the sample; that is, it is the fraction of the stars which have habitable planets.

I have created a `Tree` class for training and running a decision tree. Most of the framework is already set up for you (you're welcome), but you need to write the crucial `Leaf.ChooseVariable()` method, which actually makes the decision about the cuts. It is designed to run recursively, so once you write the method the rest of it should work automatically. The code is already set up to run everything else.

You should stop separating when you are not able to improve the purity, or if any of the leaves will be too small. At the very least, make sure each leaf has at least one event of both signal and background.

The code will make another text file, but instead of just putting 0 or 1, it will put the purity of the leaf that the data point ended up in. Your score will partly depend on how successful your algorithm is.

Level III

Decision trees have many shortcomings, but many of them can be solved by using a process called *boosting*. You can read about boosting at http://arxiv.org/abs/1004.1181 (an old paper of mine) on page 43. The idea is that you train many decision trees (called a *forest*), where each event gets a weight, or a *boost*, when put into the next tree. The final value for each event is a weighted sum of the values that the event gets in all the different trees in the forest.

Write an algorithm to implement a boosted decision tree, using a forest of at least 10 trees. Use this on the data file I gave you make a text file with the weighted sum as your output. Make sure to give it a different name.

Challenge

The same paper mentions a method of reducing extraneous leaves to reduce the possibility of overfitting using the cost complexity algorithm. Learn how the cost complexity pruning algorithm works, then implement it in your (boosted) tree so there is no need for an arbitrary minimum leaf size. Give me a new output based on this tree. It should ideally have the same sensitivity as your previous tree but have fewer leaves.

Homework 7 Answer Sheet

Level I

Level II

Level III

For all the levels, please just attach the output in a text file. Label each file Level1.txt and so forth for clarity. It is okay if you skip a level, especially Level I, but you should be very sure your new file is better – it is probably safer to submit lower levels just for safety.

Challenge

How many leaves and branches did your tree have before and after implementing the cost complexity algorithm?

Include another text file for the challenge.

Code

Copy and paste all the code that you used below. Please put a page break whenever there is a new file. If you did not modify one of my given files, you do not need to include it. However, include the complete code of any file you modified.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using System.Text;

namespace DecisionTree
{
    internal static class Program
    {
        private static readonly string path =
Directory.GetParent(Directory.GetCurrentDirectory()).Parent.Parent.Parent.FullName +
'\\';
        static void Main()
        {
            //LevelI();
            //LevelIIAndBeyond();
            LevelIII();
        }

        static void LevelI()
        {
            // Load training samples
            var signal = DataSet.ReadDataSet(path + "signal.dat");
            var background = DataSet.ReadDataSet(path + "background.dat");

            // Load data sample
            var data = DataSet.ReadDataSet(path + "decisionTreeData.dat");
```

```
            int bestVariableIndex = -1;
            double bestSplitValue = 0;
            double bestPurity = 1;

            // TODO: Insert code here that calculates the proper values of
bestVariableIndex and bestSplitValue

            double right = 0;
            double left = 0;


            for (int var = 0; var < signal.Points[0].Variables.Length; var++)
            {
                int i = 0;

                double sLeft = 0;
                double sRight = 0;
                double bLeft = 0;
                double bRight = 0;

                double[] splits = new double[signal.Points.Count +
background.Points.Count];
                double[] label = new double[signal.Points.Count +
background.Points.Count];
                foreach (DataPoint point in signal.Points) //loops through points
                {
                    splits[i] = point.Variables[var];
                    label[i] = 1;
                    sRight++;
                    i++;
                }
                foreach (DataPoint point in background.Points) //loops through
points
                {
                    splits[i] = point.Variables[var];
                    label[i] = 0;
                    bRight++;
                    i++;
                }
                Array.Sort(splits, label);

                double purityLeft = 0;
                double purityRight = 0;
                for (int ind = 0; ind < splits.GetLength(0) - 1; ind++)
                {
                    if (label[ind] == 0)
                    {
                        bRight--;
                        bLeft++;
                    }
                    else
                    {
                        sRight--;
                        sLeft++;
                    }
                    left = sLeft + bLeft;
                    right = sRight + bRight;
```

```
                    purityLeft = sLeft / left;
                    purityRight = sRight / right;

                    if (purityLeft > 0.5)
                    {
                        purityLeft = 1 - purityLeft;
                    }
                    else if (purityRight > 0.5)
                    {
                        purityRight = 1 - purityRight;
                    }
                    else
                    {
                        purityLeft = 2;
                    }
                    if (purityLeft + purityRight < bestPurity && left >= 50 && right
>= 50)
                    {
                        bestVariableIndex = var;
                        bestPurity = purityLeft + purityRight;
                        bestSplitValue = (splits[ind] + splits[ind + 1]) / 2;
                    }
                }
            }
            Console.WriteLine(bestVariableIndex + " " + bestSplitValue);
            using var file = File.CreateText(path +
"decisionTreeResultsLevelI.txt");
            file.WriteLine("Event\tPurity");

            for (int i = 0; i < data.Points.Count; ++i)
            {
                // Note that you may have to change the order of the 1 and 0 here,
depending on which one matches signal.
                // 1 means signal and 0 means background
                double output = data.Points[i].Variables[bestVariableIndex] >
bestSplitValue ? 1 : 0;
                file.WriteLine(i + "\t" + output);
            }
        }

        static void LevelIIAndBeyond()
        {
            // Load training samples
            var signal = DataSet.ReadDataSet(path + "signal.dat");
            var background = DataSet.ReadDataSet(path + "background.dat");

            // Load data sample
            var data = DataSet.ReadDataSet(path + "decisionTreeData.dat");

            var tree = new Tree();

            // Train the tree
            tree.Train(signal, background);

            // Calculate output value for each event and write to file
            tree.MakeTextFile(path + "decisionTreeResults.txt", data);
        }
```

```
        static void LevelIII()
        {
            // Load training samples
            var signal = DataSet.ReadDataSet(path + "signal.dat");
            var background = DataSet.ReadDataSet(path + "background.dat");

            // Load data sample
            var data = DataSet.ReadDataSet(path + "decisionTreeData.dat");

            double[] results = new double[data.Points.Count];
            double treeweight = 0;

            for(int treeNum = 0; treeNum < 3; treeNum++)
            {
                var tree = new Tree();

                // Train the tree
                tree.Train(signal, background);
                int wrongClassification = 0;
                for (int i = 0; i < signal.Points.Count; ++i)
                {
                    double output = tree.RunDataPoint(signal.Points[i]);
                    if (output < 0.5)
                    {
                        wrongClassification++;
                        signal.Points[i].weight =treeweight;
                    }
                }
                for (int i = 0; i < background.Points.Count; ++i)
                {
                    double output = tree.RunDataPoint(background.Points[i]);
                    if (output < 0.5)
                    {
                        wrongClassification++;
                        background.Points[i].weight = treeweight;
                    }
                }
                for (int i = 0; i < data.Points.Count; ++i)
                {
                    double output = tree.RunDataPoint(data.Points[i]);
                    if(treeNum ==0)
                    {
                        results[i] = output;
                    }
                    else
                    {
                        results[i] += Math.Abs(output *
Math.Log(Math.Abs(treeweight)));
                    }
                }
                treeweight = (1.0 - wrongClassification) / wrongClassification;
            }
            using var file = File.CreateText(path + "decisionTreeResults3.txt");
            // Calculate output value for each event and write to file
            for (int i = 0; i < results.Length; ++i)
            {
                if(results[i]>1)
                {
```

```
                    results[i] = 1;
                }
                file.WriteLine(i + "\t" + results[i]);
            }
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;

namespace DecisionTree
{
    /// <summary>
    /// Class holding information for a single event
    /// </summary>
    public class DataPoint
    {
        // An array of doubles, representing the relevant variables for the event
        public double[] Variables { get; set; }
        public double weight { get; set; }

        /// <param name="nvar">The number of different variables in the
point</param>
        public DataPoint(int nvar)
        {
            Variables = new double[nvar];
            weight = 1;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DecisionTree
{
    /// <summary>
    /// A leaf or branch of a tree
    /// Does most of the work of decision trees
    /// </summary>
    internal class Leaf
    {
        /// <summary>
        /// A pointer to the next leaves, if this is a branch
        /// </summary>
        private Leaf output1 = null;
        /// <summary>
        /// A pointer to the next leaves, if this is a branch
        /// </summary>
        private Leaf output2 = null;
        /// <summary>
        /// The value of the cut that is applied at this branch (unneeded if it is a leaf)
```

```
/// </summary>
private double split;
/// <summary>
/// The index of the variable which is used to make the cut (unneeded if this is a leaf)
/// </summary>
private int variable;

/// <summary>
/// The number of background training events in this leaf
/// </summary>
private int nBackground = 0;
/// <summary>
/// The number of signal training events in this leaf
/// </summary>
private int nSignal = 0;

/// <summary>
/// A default constructor is needed for some applications, but it is generally not sensible
/// </summary>
internal Leaf() :
   this(-1, 0) // Default values will generate an error if used
{ }

/// <param name="variable">The index of the variable used to make the cut</param>
/// <param name="split">The value of the cut used for the branch</param>
```

```
public Leaf(int variable, double split)
{
   this.variable = variable;
   this.split = split;
}

/// <summary>
/// Write the leaf to a binary file
/// </summary>
internal void Write(BinaryWriter bw)
{
   bw.Write(variable);
   bw.Write(split);
   bw.Write(nSignal);
   bw.Write(nBackground);

   bw.Write(IsFinal);
   if (!IsFinal)
   {
      output1.Write(bw);
      output2.Write(bw);
   }
}

/// <summary>
```

```
/// Construct a leaf from a binary file
/// </summary>
internal Leaf(BinaryReader br)
{
   variable = br.ReadInt32();
   split = br.ReadDouble();
   nSignal = br.ReadInt32();
   nBackground = br.ReadInt32();

   bool fin = br.ReadBoolean();
   if (!fin)
   {
      output1 = new Leaf(br);
      output2 = new Leaf(br);
   }
}

/// <summary>
/// Determines if it is a leaf or a branch (true for leaves)
/// </summary>
public bool IsFinal => output1 == null || output2 == null;

/// <summary>
/// The purity of the leaf
/// </summary>
```

```
public double Purity => (double)nSignal / (nSignal + nBackground);


/// <summary>
/// Calculates the return value for a single data point, forwarding it to other leaves as needed
/// </summary>
public double RunDataPoint(DataPoint dataPoint)
{
   if (IsFinal)
   {
      return Purity;
   }


   if (DoSplit(dataPoint))
   {
      return output1.RunDataPoint(dataPoint);
   }
   else
   {
      return output2.RunDataPoint(dataPoint);
   }
}


/// <summary>
/// Checks to see whether the DataPoint fails or passes the cut
/// </summary>
```

```
private bool DoSplit(DataPoint dataPoint)
{
   return dataPoint.Variables[variable] <= split;
}

/// <summary>
/// Trains this leaf based on input DataSets for signal and background
/// </summary>
public void Train(DataSet signal, DataSet background)
{
   nSignal = signal.Points.Count;
   nBackground = background.Points.Count;

   // Determines whether this is a final leaf or if it branches
   bool branch = ChooseVariable(signal, background);

   if (branch)
   {
      // Creates a branch
      output1 = new Leaf();
      output2 = new Leaf();

      DataSet signalLeft = new DataSet(signal.Names);
      DataSet signalRight = new DataSet(signal.Names);
      DataSet backgroundLeft = new DataSet(background.Names);
```

```
DataSet backgroundRight = new DataSet(background.Names);


foreach (var dataPoint in signal.Points)
{
   if (DoSplit(dataPoint))
   {
      signalLeft.AddDataPoint(dataPoint);
   }
   else
   {
      signalRight.AddDataPoint(dataPoint);
   }
}


foreach (var dataPoint in background.Points)
{
   if (DoSplit(dataPoint))
   {
      backgroundLeft.AddDataPoint(dataPoint);
   }
   else
   {
      backgroundRight.AddDataPoint(dataPoint);
   }
}
```

```
        // Trains each of the resulting leaves
        output1.Train(signalLeft, backgroundLeft);
        output2.Train(signalRight, backgroundRight);
    }
    else
    {
        //nsole.WriteLine(Purity);
    }
    // Do nothing more if it is not a branch
}

/// <summary>
/// Chooses which variable and cut value to use
/// </summary>
/// <returns>True if a branch was created, false if this is a final leaf</returns>

private bool ChooseVariable(DataSet signal, DataSet background)
{
    double bestPurity = 1;
    double pLeft = 1;
    double pRight = 1;
    double right = 0;
    double left = 0;
    double tRight = 0;
```

```
double tLeft = 0;


if (signal.Points.Count < 50 || signal.Points.Count < 50)
{
   return false;
}


for (int var = 0; var < signal.Points[0].Variables.Length; var++)
{
   int i = 0;


   double sLeft = 0;
   double sRight = 0;
   double bLeft = 0;
   double bRight = 0;


   double[] splits = new double[signal.Points.Count + background.Points.Count];
   int[] label = new int[signal.Points.Count + background.Points.Count];
   foreach (DataPoint point in signal.Points) //loops through points
   {
      splits[i] = point.Variables[var];
      label[i] = i;
      sRight+= point.weight;
      i++;
   }
```

```
foreach (DataPoint point in background.Points) //loops through points
{
   splits[i] = point.Variables[var];
   label[i] = i;
   bRight+= point.weight;
   i++;
}
Array.Sort(splits, label);

double purityLeft = 0;
double purityRight = 0;
for (int ind = 0; ind < splits.GetLength(0) - 1; ind++)
{
   if (label[ind] <signal.Points.Count)
   {
      sRight-=signal.Points[label[ind]].weight;
      sLeft+= signal.Points[label[ind]].weight;
   }
   else
   {
      bRight -= background.Points[label[ind] - signal.Points.Count].weight;
      bLeft += background.Points[label[ind] - signal.Points.Count].weight;
   }
   left = sLeft + bLeft;
   right = sRight + bRight;
```

```
purityLeft = sLeft / left;
purityRight = sRight / right;

if (purityLeft > 0.5)
{
   purityLeft = 1 - purityLeft;
}
else if (purityRight > 0.5)
{
   purityRight = 1 - purityRight;
}
else
{
   purityLeft = 2;
}
if (purityLeft + purityRight < bestPurity && left >= 50 && right >= 50)
{
   variable = var;
   bestPurity = purityLeft + purityRight;
   split = (splits[ind] + splits[ind + 1]) / 2;
   pLeft = purityLeft;
   pRight = purityRight;
   tLeft = left;
   tRight = right;
```

```
            }
        }
    }
    // Console.WriteLine(variable + " split: " + split + " " + right + " " + left + " " + right2 + " " + left2 + "
" + bestPurity);


    //nsole.WriteLine(variable + " split: " + split);
    if (tLeft < 50 || tRight < 50)
    {
        return false;
    }
    if (Purity > .5)
    {
        if (1 - Purity < pLeft && 1 - Purity < pRight)
        {


            return false;
        }
    }
    else
    {
        if (Purity < pLeft && Purity < pRight)
        {
            return false;
        }
    }
```

```
  //Console.WriteLine(variable + " split: " + split);
    return true;

}

//private bool ChooseVariable(DataSet signal, DataSet background)
//{
//    double bestPurity = 1;
//    double pLeft = 1;
//    double pRight = 1;
//    double right = 0;
//    double left = 0;
//    double tRight = 0;
//    double tLeft = 0;

//    if(signal.Points.Count < 50 || signal.Points.Count < 50)
//    {
//        return false;
//    }

//    for (int var = 0; var < signal.Points[0].Variables.Length; var++)
//    {
//        int i = 0;
```

```
//      double sLeft = 0;
//      double sRight = 0;
//      double bLeft = 0;
//      double bRight = 0;

//      double[] splits = new double[signal.Points.Count + background.Points.Count];
//      double[] label = new double[signal.Points.Count + background.Points.Count];
//      foreach (DataPoint point in signal.Points) //loops through points
//      {
//          splits[i] = point.Variables[var];
//          label[i] = 1;
//          sRight++;
//          i++;
//      }
//      foreach (DataPoint point in background.Points) //loops through points
//      {
//          splits[i] = point.Variables[var];
//          label[i] = 0;
//          bRight++;
//          i++;
//      }
//      Array.Sort(splits, label);

//      double purityLeft = 0;
//      double purityRight = 0;
```

```
//      for (int ind = 0; ind < splits.GetLength(0) - 1; ind++)
//      {
//          if (label[ind] == 0)
//          {
//              bRight--;
//              bLeft++;
//          }
//          else
//          {
//              sRight--;
//              sLeft++;
//          }
//          left = sLeft + bLeft;
//          right = sRight + bRight;

//          purityLeft = sLeft / left;
//          purityRight = sRight / right;

//          if (purityLeft > 0.5)
//          {
//              purityLeft = 1 - purityLeft;
//          }
//          else if (purityRight > 0.5)
//          {
//              purityRight = 1 - purityRight;
```

```
//          }
//          else
//          {
//              purityLeft = 2;
//          }
//          if (purityLeft + purityRight < bestPurity && left >=50 && right >= 50)
//          {
//              variable = var;
//              bestPurity = purityLeft + purityRight;
//              split = (splits[ind] + splits[ind + 1]) / 2;
//              pLeft = purityLeft;
//              pRight = purityRight;
//              tLeft = left;
//              tRight = right;
//          }
//      }
//  }
    //  // Console.WriteLine(variable + " split: " + split + " " + right + " " + left + " " + right2 + " " + left2 +
" " + bestPurity);


    //  Console.WriteLine(variable + " split: " + split);
    //  if(tLeft < 50 || tRight < 50)
    //  {
    //      return false;
    //  }
    //  if (Purity>.5)
```

```
        //    {
        //        if(1-Purity<pLeft && 1-Purity<pRight )
        //        {

        //            return false;
        //        }
        //    }
        //    else
        //    {
        //        if(Purity<pLeft && Purity < pRight)
        //        {
        //            return false;
        //        }
        //    }

        //    Console.WriteLine(variable + " split: " + split);
        //    return true;

        //}
    }
}
```