27 ID3: Learning from Examples

Chapter Objectives

Review of supervised learning and decision tree representation
- Representing decision trees as recursive structures
- A general decision tree induction algorithm

Information theoretic decision tree test selection heuristic

Chapter Contents

27.1 Introduction to Supervised Learning

In machine learning, *inductive learning* refers to training a learner through use of examples. The simplest case of this is rote learning, whereby the learner simply memorizes the training examples and reuses them in the same situations. Because they do not generalize from training data, rote learners can only classify exact matches of previous examples. A further limitation of rote learning is that the learned examples might contain conflicting information, and without some form of generalization, the learner cannot effectively deal with this noise. To be effective, a learner must apply heuristics to induce reliable generalizations from multiple training examples that can handle unseen situations with some degree of confidence.

A common inductive learning task is learning to classify specific instances into general categories. In *supervised learning*, a teacher provides the system with categorized training examples. This contrasts with clustering and similar unsupervised learning tasks where the learner forms its own categories from training data. See (Luger 2009) for a discussion of these different learning tasks. An example of a supervised inductive learning problem, which we will develop throughout the chapter is a bank wanting to train a computer learning system categorize new borrowers according to `credit risk` on the basis of properties such as their `credit history`, current `debt`, their `collateral`, and current `income`. One approach would be to look at the `credit risk`, as determined over time by the actual debt payoff history of data from previous borrowers to provide categorized examples. In this chapter we do exactly that, using the ID3 algorithm.

27.2 Representing Knowledge as Decision Trees

A decision tree is a simple form of knowledge representation that is widely used in both advisors and machine learning systems. Decision trees are recursive structures in which each node examines a property of a collection

of data, and then delegates further decision making to child nodes based on the value of that particular property (Luger 2009, Section 10.3). The leaf nodes of the decision tree are terminal states that return a class for the given data collection. We can illustrate decision trees through the example of a simple credit history evaluator that was used in (Luger 2009) in its discussion of the ID3 learning algorithm. We refer the reader to this book for a more detailed discussion, but will review the basic concepts of decision trees and decision tree induction in this section.

Assume we wish to assign a credit risk of high, moderate, or low to people based on the following properties of their credit rating:

Collateral, with possible values {adequate, none}

Income, with possible values {"0$ to $15K", "$15K to $35K",

"over $35K"}

Debt, with possible values {high, low}

Credit History, with possible values {good, bad, unknown}

We could represent risk criteria as a set of rules, such as "If debt is low, and credit history is good, then risk is moderate." Alternatively, we can summarize a set of rules as a decision tree, as in figure 27.1. We can perform a credit evaluation by walking the tree, using the values of the person's credit history properties to select a branch. For example, using the decision tree of figure 27.1, an individual with credit history = unknown, debt = low, collateral = adequate, and income = $15K to $35K would be categorized as having low risk. Also note that this particular categorization does not use the income property. This is a form of generalization, where people with these values for credit history, debt, and collateral qualify as having low risk, regardless of income.

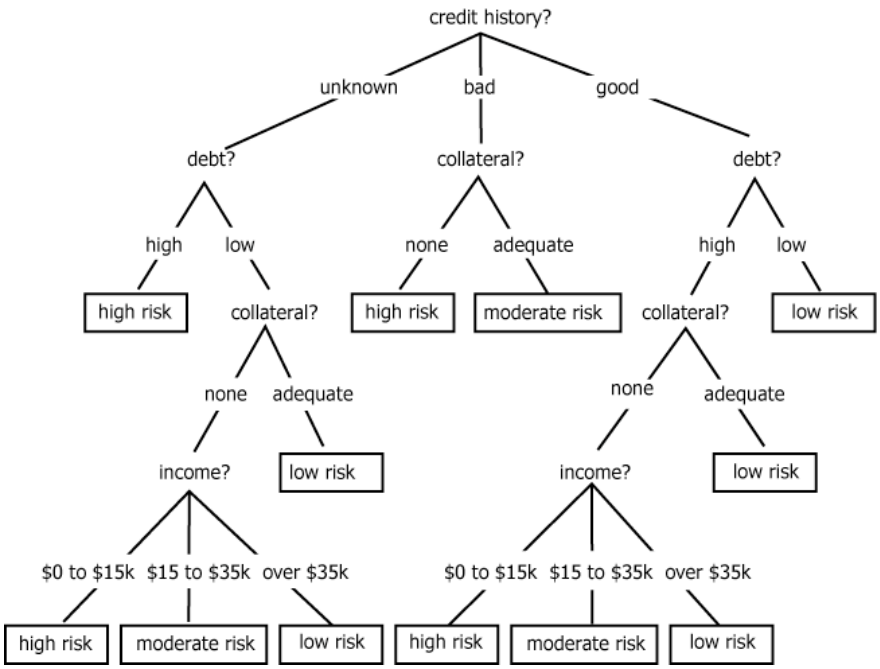

Figure 27.1 A Decision Tree for the Credit Risk Problem (Luger 2009)

Now, assume the following set of 14 training examples. Although this does not cover all possible instances, it is large enough to define a number of meaningful decision trees, including the tree of figure 27.1 (the reader may want to construct several such trees. See exercise 1). The challenge facing any inductive learning algorithm is to produce a tree that both covers all the training examples correctly, and has the highest probability of being correct on new instances.

risk	**collateral**	**income**	**debt**	**credit history**
high	none	$0 to $15K	high	bad
high	none	$15K to $35K	high	unknown
moderate	none	$15K to $35K	low	unknown
high	none	$0 to $15K	low	unknown
low	none	over $35K	low	unknown
low	adequate	over $35K	low	unknown
high	none	$0 to $15K	low	bad
moderate	adequate	over $35K	low	bad
low	none	over $35K	low	good
low	adequate	over $35K	high	good
high	none	$0 to $15K	high	good
moderate	none	$15K to $35K	high	good
low	none	over $35K	high	good
high	none	$15K to $35K	high	bad

A valuable heuristic for producing such decision trees comes from the time-honored logical principle of Occam's Razor. This principle, first articulated by the medieval logician, William of Occam, holds that we should always prefer the simplest correct solution to any problem. In our case, this would favor decision trees that not only classify all training examples, but also that do so, on average, by examining the fewest properties possible. The reason for this is straightforward: the simplest decision tree that correctly handles the known examples is the tree that makes the fewest assumptions about unknown instances. Stating it simply, the fewer assumptions made, the less likely we are to make an erroneous one.

Because omitting properties is a way of generalizing decision trees, and because the order in which the properties are tested determines the ability of the tree to omit properties while still matching all the test data, the order of tests from root down to leaf nodes is the major factor in inducing decision trees. This is captured in the following pseudo code for a recursive algorithm for inducing trees:

```
function induce_tree (example_set, Properties)
begin
  if all entries in example_set are the same class
  then return a leaf node labeled with that class
```

```
    else if Properties is empty
       then return a leaf node with default class
    else
      begin
          select a property, P, and
       make it the root of the current tree
        delete P from Properties
        for each value V of P
          begin
            create a branch of the tree labeled with V
            let partition_V be elements of
             example_set with values V of P
            let branch_V =
           induce_tree (partition_V, Properties)
            attach branch_V to root for value V of P
          end
        endfor
        return current root
      end
    endif
  end
```

This algorithm builds trees in a top-down fashion. It stops when all examples have the same categorization, thereby pruning extraneous branches of the tree. Using this algorithm, production of a simple (i.e., generalized) tree depends upon the order in which properties are selected. This, in turn, depends upon the selection function used to select the property to check in the current node of the tree.

For the decision tree induction, we use the original approach from the ID3 algorithm of (Quinlan 1986) elaborated by Luger (2009, Section 10.3). This approach uses information theory to select the property that gains the most information about the example set. Intuitively, this heuristic should minimize the number of properties the tree checks. We will explain it in detail later. We should note, however, that there are several important extensions of the early ID3 paradigm, differing only in a few operations. For example, C4.5 and C5.0 are Quinlan's (1996) own extensions that overcome a number of the original ID3 weaknesses. We will not implement C4.5/C5.0 here, but we should remember that more sophisticated or domain-specific modifications to the core decision tree induction algorithm may be desired by future developers using this code.

27.3 A Decision Tree Induction Program

Implementing this in Java raises at least two interesting problems. Managing trees, lists of examples, partitioning examples on various properties, and so forth is a challenge for designing data structures. Our example code will not be optimally efficient, but is intended to give the

student opportunities to improve performance by using table lookup and other techniques to reduce time spent scanning lists of examples. The other challenge will be in maintaining the quality of training data. We take a simplified approach of requiring all examples contain legitimate values for all desired properties. Although the machine learning literature is filled with techniques for managing missing or noisy data, this simple assumption will let us investigate a number of interesting Java techniques, such as immutable objects, error checks in constructors, etc.

Figure 27.2 shows the five classes that form the basis of our implementation. `AbstractDecisionTreeNode` defines the basic behaviors of a decision tree. It is a recursive structure, as shown by the use of an assembly link back to itself. `AbstractDecisionTreeNode` will define methods to solve a new instance by walking the tree, and the basic tree induction algorithm mentioned above. The method to evaluate a test property's partition of the example space into subproblems into will be abstract in this class, allowing definition of multiple alternative evaluation heuristics. The class, `InformationTheoreticDecisionTreeNode`, will implement the basic ID3 evaluation heuristic, which uses information theory to select a property that gives the greatest information gain on the set of training examples.

The remaining classes define and manage training examples. An `AbstractProperty` defines properties as <name, value> pairs. It is an abstract class, requiring subclasses define a method to test for legal <name, value> definitions. An `AbstractExample` defines examples as a set of properties and a categorization of those properties: i.e. a single row in the example table given above. Like `AbstractProperty`, it requires subclasses define domain specific checks for the validity of examples. Finally, `ExampleSet` maintains a set of training examples, such as is given in the table above. It enforces checks that all examples are of the same type, provides basic accessors, and also methods to partition an example set on specific properties.

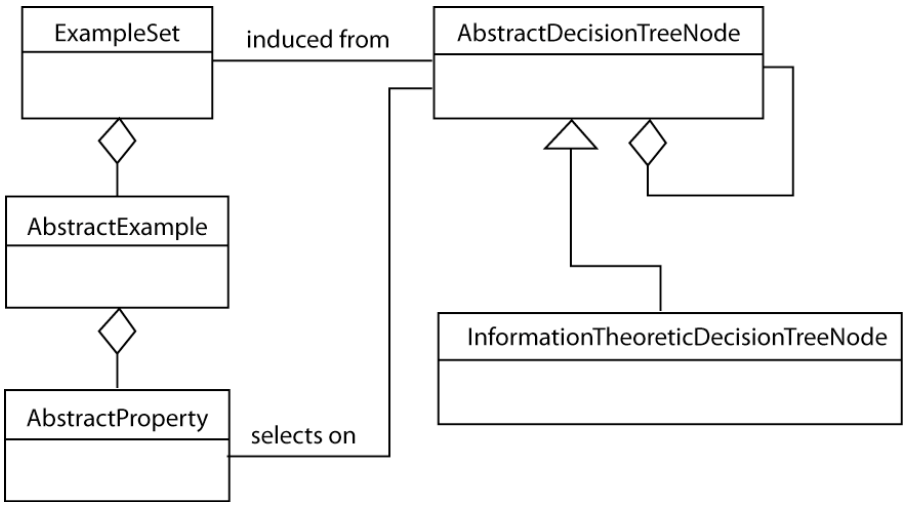

Figure 27.2 Class structure of decision tree nodes and examples

Properties as Immutable Objects

The basic definition of a property is straightforward: it consists of two strings, defining the name and value respectively. A simple initial implementation might be:

```
public class Property
{
  private String name = null;
  private String value = null;
  public Property(String name, String value)
  {
     this.name = name;
     this.value = value;
  }
  public String getName()
  {
     return name;
  }
  public String getValue()
  {
     return value;
  }
}
```

Although this gives the basic structure of the class, and would work in the program, it fails to perform any correctness checks on data values. The first of these the opportunity to perform type checks on property values. Referring to the credit evaluation example, the only values for debt are "high" and "low," and a robust program should check for them.

We can implement this by making `Property` an abstract class that uses an abstract method to test for legal property values. Each property type will be a subclass that defines this method. Our definition then becomes:

```
public abstract class AbstractProperty
{
  private String value = null;
  public AbstractProperty(String name,
        String value)
        throws IllegalArgumentException
  {
     if(isLegalValue(value) == false)
        throw
           new IllegalArgumentException(value +
              "is an illegal Value for Property " +
              getName());
     this.value = value;
  }
```

```
    public final String getValue()
    {
       return value;
    }
    public abstract boolean isLegalValue(String
       value);
    public abstract String getName();
  }
```

This version uses the `islegalValue(…)` method to check for bad values in the constructor, throwing an `IllegalArgumentException` if one is found. Since property is now an abstract class, any property type must define its own subclass defining the abstract methods. Also note that, since the name of a property is the same for all instances of a type, we have made `getName()` an abstract method as well. An example of how a property can implement this is given by this implementation of the debt property:

```
public class DebtProperty extends AbstractProperty {
  public static final String DEBT = "Debt";
  public static final String HIGH = "high";
  public static final String LOW = "low";
  public DebtProperty(String value)
  {
     super(value);
  }
  public boolean isLegalValue(String value)
  {
     return(value.equals(HIGH) ||
        value.equals(LOW));
  }
  public final String getName()
  {
     return DEBT;
  }
}
```

Although simple, the implementation of `AbstractProperty` has another interesting quality. Note that the member variable `value` is private, and we have not provided a set method other than through the constructor. This means that, once an instance of property is created, its value cannot change. This pattern is called an immutable object. Because immutable objects avoid many types of bugs (imagine the effect on the learning algorithm of changing a property value during execution), this should be used where it matches our intent. To reduce the chance that a well-intentioned programmer will change this, we should write code so as to make our intention clear. We can do this by making our get method `final`, to prevent subclasses from violating the immutability pattern, and

also by defining set methods that throw an exception if called. This completes the definition of AbstractProperty as:

```
public abstract class AbstractProperty
{
  private String value = null;
  public AbstractProperty(String value)
     throws IllegalArgumentException
  {
     if(isLegalValue(value) == false)
        throw
        new IllegalArgumentException(value +
        "is an illegal Property Value for " +
        getName());
     this.value = value;
  }
  public abstract boolean isLegalValue(String
     value);
  public abstract String getName();
  public final String getValue()
  {
     return value;
  }
  //Enforcing Immutable object pattern
  public final void setValue(String v)
     throws UnsupportedOperationException
  {
     throw new UnsupportedOperationException();
  }
  //Enforcing Immutable object pattern
  public final void setName(String n)
     throws UnsupportedOperationException
  {
     throw new UnsupportedOperationException();
  }
 }
```

Implementing Examples

Like a property, an example is conceptually simple: it is a collection of properties describing a problem instance and a categorization of that instance. In our credit example, the properties that form an example are debt, collateral, credit history, and income. The example category is a risk assessment. Each row of the example table in section 27.1 would be represented as an example. Like the property class, however, it also presents opportunities for insuring the validity of examples. In this case, we will require that an example consist only of specified properties, and that a

legal example include all properties. Examples also offer an opportunity to use an immutable object pattern, since it makes little sense to allow examples to change during the course of a learning session.

The structure of an example is similar to that of an `AbstractProperty`: it is an abstract class that requires subclasses define methods to support validity checks. We will follow the immutable object pattern, providing access methods but no "add," "set," or other modification methods, and requiring all properties be defined in the constructor.

The class has two member variables. A `category` is a `String` defining the classification of the example. In our credit example, this would be the risk level of high, moderate, or low. The `properties` member variable is a Map that indexes different properties by their name. We define two constructors. The primary constructor does error checks to require that each example contains all legal properties and only legal properties. The single argument constructor allows us to define uncategorized examples. Both of these call the private method, `addProperties` to add the elements of the `propertyList` argument to the properties member variable. This method also checks that the `propertyList` argument contains only legal values and all legal values. The implementation of `AbstractExample` is:

```
public abstract class AbstractExample
{
  private String category = null;
  private Map<String, AbstractProperty> properties
     = new HashMap <String, AbstractProperty> ();
  // Constructor for classified examples
  public AbstractExample(String category,
     AbstractProperty... propertyList)
     throws IllegalArgumentException
  {
     if(isLegalCategory(category) == false)
        throw
        new IllegalArgumentException(category +
        "is an illegal category for example.");
     this.category = category;
     addProperties(propertyList);
  }
  // Constructor for unclassified examples
  public AbstractExample(AbstractProperty...
        propertyList)
        throws IllegalArgumentException
  {
     addProperties(propertyList);
  }
```

```
private void addProperties(AbstractProperty[]
      propertyList)
      throws IllegalArgumentException
{
   Set<String> requiredProps =
      getPropertyNames();
   // check that all properties are legal
   for(int i = 0; i < propertyList.length;
      i++)
   {
      AbstractProperty prop =
         propertyList[i];
   if(requiredProps.contains(
            prop.getName()) == false)
         throw
         new IllegalArgumentException(
         prop.getName() +
         "illegal Property for example.");
      properties.put(prop.getName(), prop);
      requiredProps.remove(prop.getName());
   }
   // Check that all legal properties were used
   if(requiredProps.isEmpty() == false)
   {
      Object[] p = requiredProps.toArray();
      String props = "";
      for (int i = 0; i < p.length; i++)
         props += (String)p[i] + " ";
      throw
         new IllegalArgumentException(
         "Missing Properties in example: " +
         props);
   }
}
public AbstractProperty getProperty(
   String name)
{
   return properties.get(name);
}
public String getCategory()
{
   return category;
```

```
    }
    public String toString()
    {
        // to be defined by reader
    }
    public abstract Set<String> getPropertyNames();
}
```

This implementation of `AbstractExample` as an immutable object is incomplete in that it does not include the techniques demonstrated in `AbstractProperty` to enforce the immutability pattern. We leave this as an exercise.

Implementing `ExampleSet`

`ExampleSet`, along with `AbstractDecisionTreeNode`, is one of the most interesting classes in the implementation. This is because the decision tree induction algorithm requires a number of fairly complex operations for partitioning the example set on property values. The implementation presented here is simple and somewhat inefficient, storing examples as a simple vector. This requires examination of all examples to form partitions, retrieve examples with a specific value for a property, etc. We leave a more efficient implementation as an exercise.

In providing integrity checks on data, we have required that all examples be categorized, and that all examples belong to the same class.

The basic member variables and accessors are defined as:

```
public class ExampleSet
{
  private Vector<AbstractExample> examples =
     new Vector<AbstractExample>();
  private HashSet<String> categories =
     new HashSet<String>();
  private Set<String> propertyNames = null;
  public void addExample(AbstractExample e)
     throws IllegalArgumentException
  {
     if(e.getCategory() == null)
        throw new IllegalArgumentException(
           "Example missing categorization.");
     // Check that new example is of same class
     // as existing examples
     if((examples.isEmpty()) ||
        e.getClass() ==
           examples.firstElement().getClass())
     {
        examples.add(e);
        categories.add(e.getCategory());
```

```
            if(propertyNames == null)
               propertyNames =
                  new HashSet<String>(
                     e.getPropertyNames());
            }
            else
               throw new IllegalArgumentException(
                  "All examples must be same type.");
         }
         public int getSize()
         {
            return examples.size();
         }
         public boolean isEmpty()
         {
            return examples.isEmpty();
         }
         public AbstractExample getExample(int i)
         {
            return examples.get(i);
         }
         public Set<String> getCategories()
         {
            return new HashSet<String>(categories);
         }
         public Set<String> getPropertyNames()
         {
            return new HashSet<String>(propertyNames);
         }
                  // More complex methods to be defined.
         public int getExampleCountByCategory(String cat)
            throws IllegalArgumentException
         {
                                    // to be defined below.
         }
         public HashMap<String, ExampleSet> partition(
            String propertyName)
            throws IllegalArgumentException
         {
                                    // to be defined below.
         }
      }
```

As mentioned, this implementation is fairly simple. It stores examples as a `Vector`, so most retrieval or partitioning operations will require iterating through this list. The `categories` and `propertyNames` member variables are a convenience, allowing simpler access of these values. Since example sets should not change during a learning session, we could use an immutable object pattern in the `ExampleSet` implementation. This implementation does not, since it would lead to extremely complex constructor implementations. Instead, we implemented an `addExample` method. This method performs simple data integrity checks, requiring that all examples be of the same type, and prohibiting unclassified examples. Reworking this using an immutable pattern is left as an exercise. The remaining methods are straightforward accessors.

`ExampleSet` includes a number of methods to support the induction algorithm. The first of these counts the number of examples that belong to a given category:

```
public int getExampleCountByCategory(String cat)
  throws IllegalArgumentException
{
  Iterator<AbstractExample> iter =
     examples.iterator();
  AbstractExample example;
  int count = 0;
  while(iter.hasNext())
  {
     example = iter.next();
     if(example.getCategory().equals(cat))
           count++;
   }
     return count;
 }
```

A more complex method partitions the example set according to different examples value for a specified property. `Partition` takes as argument a property name, and returns an instance of `HashMap<String, ExampleSet>` where each key is a property value, and each value is an instance of `ExampleSet` containing examples that have that value for the chosen property. `Partition` calls to private methods, `getValues`, which returns a list of values for a property that appear in the example set, and `getExamplesByProperty`, which constructs a new instance of `ExampleSet` where each example has the same value for a property.

```
public HashMap<String, ExampleSet> partition(
  String propertyName)
  throws IllegalArgumentException
{
  HashMap<String, ExampleSet> partition =
     new HashMap<String, ExampleSet>();
```

```
  Set<String> values = getValues(propertyName);
  Iterator<String> iter = values.iterator();
  while(iter.hasNext())
  {
     String val = iter.next();
     ExampleSet examples =
        getExamplesByProperty(propertyName,
           val);
     partition.put(val, examples);
  }
  return partition;
}
private Set<String> getValues(String propName)
{
  HashSet<String>values = new HashSet<String>();
  Iterator<AbstractExample> iter =
     examples.iterator();
  while(iter.hasNext())
  {
     AbstractExample ex = iter.next();
  values.add(ex.getProperty(propName).
        getValue());
  }
  return values;
}
private ExampleSet getExamplesByProperty(
  String propName, String value)
  throws IllegalArgumentException
{
  ExampleSet result = new ExampleSet();
  Iterator<AbstractExample> iter =
     examples.iterator();
  AbstractExample example;
  while(iter.hasNext())
  {
     example = iter.next();
     if(example.getProperty(propName).getValue().
           equals(value))
        result.addExample(example);
  }
  return result;
 }
```

Placing the partitioning algorithm in a method of `ExampleSet`, rather than in the actual decision tree induction algorithm was an interesting design decision. The reason for this choice was a desire to treat `ExampleSet` as an abstract data type, including all operations on it in its class definition.

Although this implementation works, it is inefficient, performing multiple iterations through lists of examples. An alternative approach would construct more complex sets of indices of examples by property and value on construction. Trying this approach and evaluating its effectiveness is left as an exercise.

Implementing Decision Tree Nodes

A decision tree node will define methods to solve problems by walking the tree, as described in section 27.1. We have also chosen to implement the basic induction algorithm in the decision tree class. Justification for this decision was that the inherently recursive nature of the induction algorithm matched the recursive structure of trees, simplifying the implementation. Because the induction algorithm is general, and could be used with a variety of heuristics for evaluating candidate example partitions, we will make the basic implementation of decision trees an abstract class.

The basic definition of `AbstractDecisionTreeNode` appears below. Member variables include `category`, which is set to a categorization in leaf nodes; for internal nodes, its value is not defined. `DecisionPropertyName` is the property on which the node branches; it is undefined for leaf nodes. `Children` is a `HashMap` that indexes child nodes by values of `decisionPropertyName`. Each constructor calls `induceTree` to perform tree induction. Note that the two-argument constructor is protected. Its second argument is the list of unused properties for consideration by the induction algorithm, and it is only used by the `induceTree` method. The remaining methods defined below are straightforward accessors.

```
public abstract class AbstractDecisionTreeNode
{
  private String category = null;
  private String decisionPropertyName = null;
  private HashMap<String,AbstractDecisionTreeNode>
     children = new
     HashMap<String,AbstractDecisionTreeNode>();
  public AbstractDecisionTreeNode (
        ExampleSet examples)
     throws IllegalArgumentException
  {
     induceTree(examples,
        examples.getPropertyNames());
  }
  protected AbstractDecisionTreeNode(ExampleSet
     examples, Set<String> selectionProperties)
```

```
   throws IllegalArgumentException
{
   induceTree(examples, selectionProperties);
}
 public boolean isLeaf()
{
   return children.isEmpty();
 }
public String getCategory()
{
   return category;
}
public String getDecisionProperty()
{
   return decisionPropertyName;
 }
public AbstractDecisionTreeNode getChild(String
   propertyValue)
{
   return children.get(propertyValue);
}
 public void addChild(String propertyValue,
   AbstractDecisionTreeNode child)
{
   children.put(propertyValue, child);
 }
public String Categorize(AbstractExample ex)
{
   // defined below
}
public void induceTree(ExampleSet examples,
      Set<String> selectionProperties)
      throws IllegalArgumentException
{
   // defined below
}
public void printTree(int level)
{
   // implementation left as an exercise
}
protected abstract double
```

```
      evaluatePartitionQuality(HashMap<String,
         ExampleSet> part, ExampleSet examples)
         throws IllegalArgumentException;
   protected abstract AbstractDecisionTreeNode
      createChildNode(ExampleSet examples,
      Set<String> selectionProperties)
      throws IllegalArgumentException;
}
```

Note the two abstract methods for evaluating a candidate partition and creating a new child node. These will be implemented on 27.3.

`Categorize` categorizes a new example by performing a recursive tree walk.

```
public String categorize(AbstractExample ex)
{
   if(children.isEmpty())
      return category;
   if(decisionPropertyName == null)
      return category;

   AbstractProperty prop =
      ex.getProperty(decisionPropertyName);
     AbstractDecisionTreeNode child =
      children.get(prop.getValue());
      if(child == null)
      return null;
   return child.categorize(ex);
}
```

`InduceTree` performs the induction of decision trees. It deals with four cases. The first is a normal termination: all examples belong to the same category, so it creates a leaf node of that category. Cases two and three occur if there is insufficient information to complete a categorization; in this case, the algorithm creates a leaf node with a null category.

Case four performs the recursive step. It iterates through all properties that have not been used in the decision tree (these are passed in the parameter `selectionProperties`), using each property to partition the example set. It evaluates the example set using the abstract method, `evaluatePartitionQuality`. Once it finds the best evaluated partition, it constructs child nodes for each branch.

```
public void induceTree(ExampleSet examples,
       Set<String> selectionProperties)
   throws IllegalArgumentException
{
                        // Case 1: All instances are the same
                        // category, the node is a leaf.
```

```
if(examples.getCategories().size() == 1)
  {
     category = examples.getCategories().
        iterator().next();
     return;
  }
                    //Case 2: Empty example set. Create
                    // leaf with no classification.
    if(examples.isEmpty())
     return;
                    //Case 3: Empty property set; could not classify.
  if(selectionProperties.isEmpty())
     return;
                    // Case 4: Choose test and build subtrees.
                    // Initialize by partitioning on first
                    // untried property.
  Iterator<String> iter =
          selectionProperties.iterator();
  String bestPropertyName = iter.next();
  HashMap<String, ExampleSet> bestPartition =
        examples.partition(bestPropertyName);
  double bestPartitionEvaluation =
         evaluatePartitionQuality(bestPartition,
        examples);
                    // Iterate through remaining properties.
  while(iter.hasNext())
  {
     String nextProp = iter.next();
     HashMap<String, ExampleSet> nextPart =
             examples.partition(nextProp);
     double nextPartitionEvaluation =
             evaluatePartitionQuality(nextPart,
           examples);
                    // Better partition found. Save.
     if(nextPartitionEvaluation >
        bestPartitionEvaluation)
     {
        bestPartitionEvaluation =
           nextPartitionEvaluation;
        bestPartition = nextPart;
        bestPropertyName = nextProp;
     }
  }
                    // Create children; recursively build tree.
  this.decisionPropertyName = bestPropertyName;
```

```
Set<String> newSelectionPropSet =
      new HashSet<String>(selectionProperties);
  newSelectionPropSet.remove(decisionPropertyName);
iter = bestPartition.keySet().iterator();
while(iter.hasNext())
{
   String value = iter.next();
   ExampleSet child = bestPartition.get(value);
   children.put(value,
      createChildNode(child,
         newSelectionPropSet));
}
```

27.4 ID3: An Information Theoretic Tree Induction Algorithm

The heart of the ID3 algorithm is its use of information theory to evaluate the quality of candidate partitions of the example set by choosing properties that gain the most information about an examples categorization. Luger (2009) discusses this approach in detail, but we will review it briefly here.

Shannon (1948) developed a mathematical theory of information that allows us to measure the information content of a message. Widely used in telecommunications to determine such things as the capacity of a channel, the optimality of encoding schemes, etc., it is a general theory that we will use to measure the quality of a decision property.

Shannon's insight was that the information content of a message depended upon two factors. One was the size of the set of all possible messages, and the probability of each message occurring. Given a set of possible messages, $M = \{m_1, m_2 \ldots m_n\}$, the information content of any individual message is measured in bits by the sum, across all messages in M of the probability of each massage times the log to the base 2 of that probability.

$$I(M) = \sum - p(m_i) \log 2\ p(m_i)$$

Applying this to the problem of decision tree induction, we can regard a set of examples as a set of possible messages about the categorization of an example. The probability of a message (a given category) is the number of examples with that category divided by the size of the example set. For example, in the table in section 27.1, there are 14 examples. Six of the examples have high risk, so p(risk = high) = 6/14. Similarly, p(risk = moderate) = 3/14, and p(risk = low) = 5/14. So, the information in any example in the set is:

I(example set) = -6/14 log (6/14) -3/14 log (3/14) -5/14 log (5/14)
= - 6/14 * (-1.222) - 3/14 * (-2.222) - 5/14 * (-1.485)
= 1.531 bits

We can think of the recursive tree induction algorithm as gaining information about the example set at each iteration. If we assume a set of

training instances, C, and a property P with n values, then P will partition C into n subsets, {c_1, c_2, . . . cY}. The information needed to finish inducing the tree can be measured as the sum of the information in each subset of the partition, weighted by the size of that partition. That is, the expected information gain to complete the tree, E, is computed by:

$$E(P) = S\ (|c_i|/|C|) * I(c_i)$$

Therefore, the information gained for property P is:

$$Gain(P) = I(C) - E(P)$$

The ID3 algorithm uses this value to rank candidate partitions.

Implementing Information Theoretic Evaluation

We will implement this in a subclass of `AbstractDecisionTreeNode` called `InformationTheoreticDecisionTreeNode`. This class will implement the two abstract methods of the parent class, along with needed constructors. The `createChildNode` method is called in `AbstractDecisionTreeNode` to create the proper type of child node. `EvaluatePartitionQuality` computes the information gain of a partition. It calls the private methods `computeInformation` and `log2`.

```
public class InformationTheoreticDecisionTreeNode
  extends  AbstractDecisionTreeNode
{
  public InformationTheoreticDecisionTreeNode(
     ExampleSet examples)
     throws IllegalArgumentException
  {
     super(examples);
  }
  public InformationTheoreticDecisionTreeNode(
     ExampleSet examples,
     Set<String> selectionProperties)
     throws IllegalArgumentException
  {
     super(examples, selectionProperties);
  }
   protected AbstractDecisionTreeNode
     createChildNode(
     ExampleSet examples,
     Set<String> selectionProperties)
     throws IllegalArgumentException
  {
     return new
        InformationTheoreticDecisionTreeNode(
           examples, selectionProperties);
     }
```

```
  protected double evaluatePartitionQuality(
         HashMap<String, ExampleSet> part,
         ExampleSet examples)
     throws IllegalArgumentException
  {
     double examplesInfo =
        computeInformation(examples);
     int totalSize = examples.getSize();
     double expectedInfo = 0.0;
     Iterator<String> iter =
        part.keySet().iterator();
     while(iter.hasNext())
     {
        ExampleSet ex = part.get(iter.next());
        int partSize = ex.getSize();
        expectedInfo += computeInformation(ex)
           * partSize/totalSize;
  }
     return examplesInfo - expectedInfo;
  }
  private double computeInformation(
         ExampleSet examples)
     throws IllegalArgumentException
  {
     Set<String> categories =
        examples.getCategories();
     double info = 0.0;
     double totalCount = examples.getSize();
        Iterator<String> iter =
            categories.iterator();
     while (iter.hasNext())
     {
        String cat = iter.next();
        double catCount = examples.
           getExampleCountByCategory(cat);
        info += -(catCount/totalCount)*
              log2(catCount/totalCount);
     }
     return info;
  }
  private double log2(double a)
  {
     return Math.log10(a)/Math.log10(2);
  }
}
```

Exercises

1. Construct two or three different trees that correctly classify the training examples in the table of section 27.1. Compare their complexity using average path length from root to leaf as a simple metric. What informal heuristics would use in constructing the simplest trees to match the data? Manually build a tree using the information theoretic test selection algorithm from the ID3 algorithm. How does this compare with your informal heuristics?

2. Extend the definition of `AbstractExample` to enforce the immutable object pattern using `AbstractProperty` as an example.

3.The methods `AbstractExample` and `AbstractProperty` throw exceptions defined in Java, such as `IllegalArgumentException` or `UnsupportedOperationException` when passed illegal values or implementers try to violate the immutable object pattern. An alternative approach would use user-defined exceptions, defined as subclasses of java.lang.RuntimeException. Implement this approach, and discuss its advantages and disadvantages.

4. The implementation of `ExampleSet` in section 27.2.3 stores component examples as a simple vector. This requires iteration over all examples to partition the example set on a property, count categories, etc. Redo the implementation using a set of maps to allow constant time retrieval of examples having a certain property value, category, etc. Evaluate performance for this implementation and that given in the chapter.

5. Complete the implementation for the credit risk example. This will involve creating subclasses of `AbstractProperty` for each property, and an appropriate subclass of `AbstractExample`. Also, write a class and methods to test your code.