Classification with Decision Trees

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
In this lecture, we will discuss a fundamental topic in data mining: classification.

More specifically, we are given a training set, which contains objects of two classes. The goal is to build a model from the training set, which which we can predict (hopefully as accurately as possible) the class of any object outside the training set.
Example 1.

Suppose that we have the following training set:

<table>
<thead>
<tr>
<th>age</th>
<th>education</th>
<th>occupation</th>
<th>loan default</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>high school</td>
<td>self-employed</td>
<td>yes</td>
</tr>
<tr>
<td>32</td>
<td>master</td>
<td>programmer</td>
<td>no</td>
</tr>
<tr>
<td>33</td>
<td>undergrad</td>
<td>lawyer</td>
<td>yes</td>
</tr>
<tr>
<td>37</td>
<td>undergrad</td>
<td>programmer</td>
<td>no</td>
</tr>
<tr>
<td>40</td>
<td>undergrad</td>
<td>self-employed</td>
<td>yes</td>
</tr>
<tr>
<td>45</td>
<td>master</td>
<td>self-employed</td>
<td>no</td>
</tr>
<tr>
<td>48</td>
<td>high school</td>
<td>programmer</td>
<td>no</td>
</tr>
<tr>
<td>50</td>
<td>master</td>
<td>lawyer</td>
<td>no</td>
</tr>
<tr>
<td>52</td>
<td>master</td>
<td>programmer</td>
<td>no</td>
</tr>
<tr>
<td>55</td>
<td>high school</td>
<td>self-employed</td>
<td>no</td>
</tr>
</tbody>
</table>

Now we are given a new customer (50, high school, self-employed) with an unknown "default" value. How should we predict this value?
Formally, we are given a training set in the form of a table $R$ where

- there are $d \geq 1$ describing attributes $A_1, A_2, \ldots, A_d$, and one class attribute $C$
  - We will consider that $C$ is binary, namely, there are only two classes: yes and no.
- each row $t$ of $R$ describes a distinct object such that $t[A_i]$ $(1 \leq i \leq d)$ is the object’s value on attribute $A_i$, and $t[C]$ is the object’s class.

The task of classification is to produce a function $M(A_1, A_2, \ldots, A_d)$ that

- takes as parameters $d$ describing values (one for each $A_i$ for $1 \leq i \leq d$), and
- returns a class value (i.e, either yes or no).

We refer to function $M$ as the model of the classification.
There are different classification methods. In this lecture, we will focus on the decision tree method, which represents the model $M$ as a tree.

**Example 2.**

Given an object (50, high school, self-employed), the above tree returns the class value “no” by descending a root-to-leaf path to the rightmost leaf.
Formally, we define a decision tree \( T \) to be a binary tree where:

- each leaf node carries a class value: yes or no.
- each internal node \( u \):
  - has two child nodes
  - carries the name \( A_u \) of a describing attribute and a predicate \( P_u \).

Given an object \( e \) with describing attributes \( e[A_1], \ldots, e[A_d] \), the model \( M \) decides its class value \( e[C] \) as follows:

1. \( u \leftarrow \) the root of \( T \)
2. if \( u \) is a leaf, then return the class value associated with \( u \)
3. if \( u \) is an internal node, check whether \( e[A_u] \) satisfies \( P_u \)
   - if so, \( u \leftarrow \) the left child of \( u \)
   - otherwise, \( u \leftarrow \) the right child of \( u \).
Think

What are the predicates associated with the internal nodes in the decision tree of Slide 5?

Notice that the decision tree of Slide 5 correctly classifies all the objects in the training set of Slide 3.

Think

Give another decision tree that can correctly classify all the objects in the training set of Slide 3.

In general, our objective is to produce a good decision tree from the training set $R$. In the next few slides, we will discuss an algorithm called the Hunt's algorithm which achieves the purpose well in practice.
Let us first introduce a notation. Given a node $u$ in a decision tree $T$, we define a set $R(u)$ of objects as follows:

- If $u$ is the root of $T$, then $R(u)$ includes all the objects in the original training set $R$.
- Otherwise, $R(u)$ is the set of objects in $R$ that satisfy the predicates of all the edges on the path from the root of $T$ to $u$.

**Fact**

If $v_1$ and $v_2$ are child nodes of $u$, then $R(u) = R(v_1) \cup R(v_2)$.

We say that $R(u)$ is split into $R(v_1)$ and $R(v_2)$.

**Think**

For each node $u$ in the decision tree on Slide 5, explain what is its $R(u)$.
Hunt’s Algorithm

The algorithm builds a decision tree $T$ in a top-down and greedy manner. More specifically, at each node $u$ of $T$, it adopts the “best” way to split $R(u)$ according to an appropriate metric.

**Algorithm** Hunt($R$)

/* $R$ is the training set; the function returns the root of a decision tree */
1. if all the objects in $R$ belong to the same class
2. return a leaf node with the value of this class
3. if all the objects in $R$ have the same describing attribute values
4. return a leaf node whose class value is the majority one in $R$
5. find the “best” split attribute $A^*$ and predicate $P^*$ /* details next slide */
6. $R_1 \leftarrow$ the set of objects in $R$ satisfying $P^*$; $R_2 \leftarrow R \setminus R_1$
7. $u_1 \leftarrow$ Hunt($R_1$); $u_2 \leftarrow$ Hunt($R_2$)
8. create a root $u$ with left child $u_1$ and right child $u_2$
9. set $A_u \leftarrow A^*$, and $P_u \leftarrow P^*$
10. return $u$
It remains to explain how to implement Line 5 of the pseudocode. We aim to resolve the following two issues:

1. What are the candidate ways to define a split predicate on a training set $R$?
2. How to evaluate the quality of a candidate split?

After this, we can implement Line 5 by setting $A^*$ and $P^*$ according to the candidate split of the best quality.
Candidate Split Predicate

Candidate splits can be defined in various methods (which give different variants of the Hunt’s algorithm). Next, we describe an intuitive method that works well in practice.

A split predicate concerns one describing attribute $A$. We distinguish two types of $A$:

- **Ordinal**: there is an ordering on the values of $A$.
- **Nominal**: no ordering makes sense on the values of $A$.

**Example 3.**

In the training set of Slide 3, age and education are ordinal attributes, whereas occupation is nominal.
For an ordinal attribute $A$, we define a candidate split predicate to be a condition of the form $A \leq v$, where $v$ is a value of $A$ that appears in $R$, such that $R$ has at least one object satisfying the condition, and has at least one object violating the condition.

For a nominal attribute $A$, we define a candidate split predicate to be a condition of the form $A \in S$, where $S$ is a subset of the values of $A$ that appear in $R$, such that $R$ has at least one object satisfying the condition, and has at least one object violating the condition.

Example 4.

In the training set of Slide 3, “age $\leq 40$”, “education $\leq$ undergrad”, and “occupation $\in \{\text{self-employed, lawyer}\}$” are all candidate split predicates. But “age $\leq 41$”, “education $\leq$ elementary”, and “occupation $\in \{\text{professor, lawyer}\}$” are not. Also, “age $\leq 55$” is not either (why?).
Quality of a Split

Now we proceed to tackle the second issue of Slide 10. Let us first introduce a key concept called GINI index.

Let $R$ be a set of objects whose class values are known. Define:

\[ n = |R| \]
\[ n_y = \text{number of objects in } R \text{ in the yes class} \]
\[ p_y = \frac{n_y}{n} \]
\[ p_n = 1 - p_y \]

Then, the GINI index of $R$, denoted as $\text{GINI}(R)$, to be:

\[ \text{GINI}(R) = 1 - (p_y^2 + p_n^2) \]
Example 5.

- If $p_y = 1$ and $p_n = 0$ (i.e., maximum purity), then $\text{GINI}(R) = 0$.
- If $p_y = 0.75$ and $p_n = 0.25$, then $\text{GINI}(R) = 0.375$.
- If $p_y = 0.5$ and $p_n = 0.5$ (i.e., maximum impurity), then $\text{GINI}(R) = 0.5$.

The next lemma is fundamental:

Lemma 6.

$\text{GINI}(R)$ ranges from 0 to 0.5. It increases as $|p_y - p_n|$ decreases.
We are now ready to quantify the quality of a split. Suppose that we have fixed a split predicate \( P \) (and hence, also the corresponding split attribute \( A \)). The split breaks the training set \( R \) into:

- \( R_1 \): the set of objects in \( R \) satisfying \( P \).
- \( R_2 = R \setminus R_1 \).

We define the split’s GINI index as:

\[
GINI_{split} = \frac{|R_1|}{|R|} GINI(R_1) + \frac{|R_2|}{|R|} GINI(R_2)
\]

The smaller \( GINI_{split} \) is, the better the split quality.

**Think**

Why don’t we simply define \( GINI_{split} \) to be \( GINI(R_1) + GINI(R_2) \)?
At this point, we have completed the description of the Hunt's algorithm on Slide 23. We have, however, intentionally left out an important issue: the algorithm often suffers from a phenomenon called **overfitting**, which adversely affects the accuracy of the resulting decision tree.

Fortunately, this issue can be adequately dealt with by introducing a small heuristic to the algorithm. However, to appreciate why the heuristic makes sense, we need to gain a better understanding on what is overfitting and what causes it. We will achieve the purpose by taking a probabilistic view into the essence of classification.
Let us look at a simplified classification task. Suppose that there are no describing attributes at all. Let $U$ be the set of people in the whole world. We want to learn a model that, given a random person, predicts whether s/he drinks.

For this purpose, we are given a training set $R \subseteq U$. Using the Hunt’s algorithm, we obtain from $R$ a decision tree $T$.

It is easy to see that $T$ has only a single leaf. Let $c$ be the class value of this leaf (i.e., $c =$ either yes or no). Then, for every object in $U$, we will predict its class value as $c$. 
Which value of \( c \) would be good for \( U \)? This ought to be related to how many people in \( U \) belong to the yes class, and how many to the no class. Specifically, let

\[
\pi_y = \frac{\text{number of people in } U \text{ of yes}}{|U|}
\]
\[
\pi_n = \frac{\text{number of people in } U \text{ of no}}{|U|}
\]

For predicting a person taken randomly from \( U \), we should set \( c \) to yes if \( \pi_y > \pi_n \), or to no otherwise.

**Example 7.**

Suppose \( \pi_y = 0.7 \) and \( \pi_n = 0.3 \). Then, if we set \( c \) to yes, we will be correct 70\% of the time. On the other hand, if we set \( c \) to no, we will be correct only 30\% of the time.
However, we do not know the real values of $\pi_y$ and $\pi_n$. Hence, we rely on $R$ to guess the relationship between those two values. If there are more yes objects in $R$ than no objects, we guess $\pi_y > \pi_n$, and hence, set $c$ to yes; otherwise, we set $c$ to no. This is precisely what the Hunt’s algorithm is doing.

How to increase the likelihood of obtaining a good guess? There are two conditions:

1. $R$ is a random sample set of $U$.
2. The size $s$ of $R$ is large.

Think

Why are both conditions necessary?

The second condition is particularly relevant to our subsequent discussion. This is very intuitive: if you do not have enough training data, you should not hope to build a reliable decision tree. We say that in such a case your training data loses statistical significance.
With the above insight, we are now ready to explain the issue of overfitting. As the Hunt’s algorithm builds the decision tree $T$ in a top-down manner, the size $R(u)$ of the current node $u$ continuously decreases as we go deeper into $T$. When $|R(u)|$ has become too small, then statistical significance is lost, such that the subtree of $u$ we grow according to the Hunt’s algorithm becomes unreliable. The consequence is that even though the subtree may fit the training set very well, it does not predict well the classes of objects outside the training set. Therefore, overfitting occurs.

**Think**

Use the probabilistic view in the previous few slides to explain why it is a bad idea to grow the subtree of $u$ when $|R(u)|$ is small.
Hunt’s Algorithm (Modified)

We now add a heuristic to the algorithm to reduce overfitting.

\textbf{algorithm} Hunt\((R)\)

\begin{verbatim}
/* \(R\) is the training set; the function returns the root of a decision tree */
1. if all the objects in \(R\) belong to the same class
2. return a leaf node with the value of this class
3. if (all the objects in \(R\) have the same describing attribute values)
   or (\(|R|\) is too small)
4. return a leaf node whose class value is the majority one in \(R\)
5. find the “best” split attribute \(A^*\) and predicate \(P^*\)
6. \(R_1 \leftarrow \) the set of objects in \(R\) satisfying \(P^*\); \(R_2 \leftarrow R \setminus R_1\)
7. \(u_1 \leftarrow\) Hunt\((R_1)\); \(u_2 \leftarrow\) Hunt\((R_2)\)
8. create a root \(u\) with left child \(u_1\) and right child \(u_2\)
9. set \(A_u \leftarrow A^*\), and \(P_u \leftarrow P^*\)
10. return \(u\)
\end{verbatim}

\textbf{Remark:} Judging whether \(|R|\) is too small is application dependent. A simple heuristic is to introduce a threshold \(\tau\) such that \(|R|\) is deemed too small if \(|R| < \tau\).
Heuristic Warning

We finish this introduction to decision trees by pointing out that the Hunt’s algorithm is a heuristic algorithm that does not emphasize on theoretical guarantees. Such an algorithm is designed based on some good rationales, but there does not exist an unambiguous yardstick to compare the quality of two algorithms that follow different rationales. Heuristic algorithms seem to fit nicely “problems” which do not have unique answers. Classification is such a “problem” (for this reason, we called it a “task” when we defined it at the beginning): indeed, what is the “best decision tree” is subjective.
Heuristic Warning (cont.)

Here is a short list of places that could be designed in different ways:

- Definition of a candidate split.
  - To alleviate overfitting, it may make sense to require a candidate split to generate only sufficiently large subsets. Currently, a candidate split may create subsets of size 1.

- Quality assessment of a candidate split.

- Stopping condition (i.e., when do we want to make a node a leaf).