Data Mining:
Classification/Supervised Learning
Potpourri

Contents

1. C4.5. and continuous attributes: incorporating continuous attributes into C4.5 Algorithm.
2. C4.5. and overfit: dealing with overfit in decision trees
3. \( k \)NN: \( k \) Nearest Neighbors learning: a lazy evaluation learning algorithm.
4. Bagging and Boosting: ensembles of classifiers to the rescue!
5. Random Forests: taking ensemble classifiers to the next level

1 Handling of Continuous Attributes in C4.5. Algorithm

Notation. Let \( D \) be a dataset over the list of attributes \( A = \{A_1, \ldots, A_n\} \). Let \( A_i \in A \) be a continuous attribute.

A binary split of \( D \) on attribute \( A_i \) at value \( \alpha \) is a pair \( D^- \subseteq D, D^+ \subseteq D \), such that:

1. \( D^- \cup D^+ = D \)
2. \( D^- \cap D^+ = \emptyset \)
3. \((\forall d \in D^-) d[A_i] \leq \alpha\);
4. \((\forall d \in D^+) d[A_i] > \alpha\);

Idea. On each step of C4.5 Algorithm, for each continuous attribute \( A_i \) find a binary split with the best information gain (or information gain ratio). More specifically, the entropy of a binary split of \( D \) on \( A_i \) using \( \alpha \) is

\[
entropy_{A_i, \alpha}(D) = \frac{|D^-|}{|D|} \cdot entropy(D^-) + \frac{|D^+|}{|D|} \cdot entropy(D^+).
\]
function selectSplittingAttribute(A, D, threshold); //uses information gain
begin
    p₀ := entropy(D);
    for each Aᵢ ∈ A do
        if Aᵢ is continuous then
            x := findBestSplit(Aᵢ, D);
            p[Aᵢ] := entropyAᵢ,x(D);
        else
            p[Aᵢ] := entropyAᵢ(D);
        endif
        Gain[Aᵢ] = p₀ − p[Aᵢ]; //compute info gain
    endfor
    best := arg(findMax(Gain[]));
    if Gain[best] > threshold then return best
    else return NULL;
end

function findBestSplit(Aᵢ, D); //finds best binary split for a continuous attribute
begin
    initialize associative arrays counts₁[], ..., countsₖ[];
    initialize associative array Gain;
    p₀ := entropy(D);
    sort D in ascending order on Aᵢ;
    for l = 1 to |D| do //Step 1: scan data
        d := D[l]; // Select next data point from D in order of Aᵢ values
        alpha[l] := d[Aᵢ]; // Remember the l-th split value
        for j = 1 to k do
            if class(d) == cⱼ then
                countsⱼ[l] := countsⱼ[l − 1] + 1;
            else
                countsⱼ[l] := countsⱼ[l − 1] + 0;
            endif
        endfor
    endfor
    for l = 1 to |D| do //computes entropy of binary split at alpha[l]
        Gain[l] := p₀ − entropy(D, Aᵢ, (counts₁[l], ..., countsₖ[l]));
    endfor
    best := arg(findMax(Gain[]));
    return alpha[best];
end

Figure 1: A modified version of selectSplittingAttribute() function for the C4.5 Algorithm. This version finds the best binary split for any continuous attribute.
The information gain obtained by using $A_i$ with the binary split at $\alpha$ is:

$$Gain_{A_i,\alpha}(D) = entropy(D) - entropy_{A_i,\alpha}(D).$$

**Finding best binary split.** The new version of the `selectSplittingAttribute()` function is in Figure 1.

- When attribute $A_i$ is **continuous**, new `selectSplittingAttribute()` calls `findBestSplit()` function, also shown in Figure 1.
- To find the best binary split, we
  - scan the dataset $D$ and determine the list of all values of $A_i$.
  - Note, that while $dom(A_i)$ is continuous, $D$ contains finitly many distinct values of $A_i$!
  - For each value $x$ in of $A_i$ from $D$ find $entropy_{A_i,x}(D)$.
  - Find $x$ with the largest information gain and return it.

**Other adjustments to C4.5.** One more adjustment to C4.5 needs to be made.

- if a **categorical attribute** is selected to split $D$ on the current step of the algorithm, this attribute is **removed from the attribute list** passed in the recursive calls to C4.5. (same as before)
- if a **continuous attribute** is selected to split $D$ on the current step of the algorithm, this attribute is **kept in the attribute list** passed in the recursive calls to C4.5. (**new**)

**Dealing with "Ghost" Paths**

One important adjustment to C4.5 needs to be made to enable it to deal with situations when a subset $D_i$ does not have data point for some value $a_j$ of an attribute $A$ on which $D_i$ is being split.

The version of C4.5. from the previous handout contained this pseudocode:

```plaintext
foreach v ∈ dom(A_g) do
    D_v := {t ∈ D | t[A_g] = v};
    if D_v ≠ ∅ then
        T_v := C45(D_v, A − {A_g}, threshold); //recursive call
        ... endfor
endif
endfor
```

The if statement in this fragments checks whether the set of data points $t ∈ D$ for which $t.A = v$, the selected value of the attribute $A$ is empty. The recursive calls to C4.5 only happen if it is **not empty**. If $D_v = \emptyset$, no action is taken.

The problem with this approach is that this tree will fail to classify a data point $x$ which has $x.A = v$, and whose other attribute values match the path from the root of the decision tree to the node in question.
Algorithm C4S(D, A, threshold);
if for all $d \in D$ class$(d) = c$, then
  // Step 1: check termination conditions
  create leaf node $r$;
  label$(r) := c$;
  $T := r$;
else if $A = \emptyset$ then
  $c := \text{find most frequent label}(D)$;
  create leaf node $r$;
  label$(r) := c$;
  $T := r$;
else // Step 2: select splitting attribute
  $A_g := \text{selectSplittingAttribute}(A, D, \text{threshold})$;
  if $A_g = \text{NULL}$ then //no attribute is good for a split
    create leaf node $r$;
    label$(r) := \text{find most frequent label}(D)$;
    $T := r$;
  else // Step 3: Tree Construction
    create tree node $r$;
    $T := r$;
    label$(r) := A_g$;
    foreach $v \in \text{dom}(A_g)$ do
      $D_v := \{t \in D | t[A_g] = v\}$;
      if $D_v \neq \emptyset$ then
        if $A_g$ is categorical then
          $T_v := \text{C4S}(D_v, A - \{A_g\}, \text{threshold})$; //recursive call
        else // $A_g$ is numeric
          $T_v := \text{C4S}(D_v, A, \text{threshold})$; //recursive call
        endif
        append $T_v$ to $r$ with an edge labeled $v$;
      else // Create a "ghost" leaf node
        create a "ghost" leaf node $r_v$;
        label$(r_v) := \text{find most frequent label}(D)$;
        append $r_v$ to $r$ with an edge labeled $v$;
      endif
    endif
  endif
endif
endif
return $T$;

Figure 2: Updated version of the C4.5 algorithm for decision tree induction. This version handles numeric attribute splits, and creates "ghost" leaf nodes to ensure that all input data points can be successfully classified.
We can adjust C4.5 to produce the best possible prediction in such cases by creating a "ghost" leaf node that matches the value $v$ of the attribute $A$. The classification decision for this leaf node is the **plurality class** of the dataset $D$.

Figure 2 adjusts the pseudocode accordingly.

## C4.5. and Overfitting

**Overfitting.** Let $D_{\text{training}}$ be a training set for a classification problem, and $D_{\text{test}}$ be a test set. Let $f$ be a classifier trained on $D_{\text{training}}$.

$f$ **overfits the data**, if there exists another classifier $f'$ which has higher accuracy than $f$ on $D_{\text{training}}$ but lower accuracy than $f$ on $D_{\text{test}}$.

**Cases of overfitting:**

- **Noise in data.** (e.g., wrong class labels)
- **Randomness phenomena.** (training set is not representative of the application domain)
- **Complexity of model.** (too many attributes, some may not be needed for classification)

**Dealing with overfitting.** Two main approaches:

- **Pre-pruning or stopping early.** E.g., the third termination condition in Algorithm C4.5 terminates tree construction early using the user-specified threshold parameter.

  - **Post-pruning or pruning a constructed tree.** In this approach, the classification algorithm is allowed to possibly overfit the data, but a separate pruning algorithm will then check the classifier for overfitting.

## $k$-Nearest Neighbors Classification ($k$NN)

C4.5. and many other classification techniques (Neural Nets, SVN, Rule Induction) are **eager**: these techniques analyze the training set and construct a classifier **before any test data is read**.

The principle of **lazy evaluation** is to **postpone any data analysis until an actual question has been asked**.

In case of supervised learning, **lazy evaluation** means **not building a classifier** in advance of reading data from the test data set.
**k-Nearest Neighbors Classification algorithm** (*kNN*). *kNN* is a simple, but surprisingly robust lazy evaluation algorithm. The idea behind *kNN* is as follows:

- The input of the algorithm is a training set \( D_{\text{training}} \), an instance \( d \) that needs to be classified and an integer \( k > 1 \).
- The algorithm computes the *distance* between \( d \) and every item \( d' \in D \).
- The algorithm selects \( k \) most similar or closest to \( d \) records from \( D \): \( d_1, \ldots, d_k \), \( d_i \in D \).
- The algorithm assigns to \( d \) the class of the plurality of items from the list \( d_1, \ldots, d_k \).

**Distance/similarity measures.** The distance (or similarity) between two records can be measured in a number of different ways.

*Note:* Similarity measures increase as the similarity between two objects increases. Distance measures decrease as the similarity between two objects increases.

1. **Euclidean distance.** If \( D \) has continuous attributes, each \( d \in D \) is essentially a point in \( N \)-dimensional space (or an \( N \)-dimensional vector). Euclidean distance:

\[
d(d_1, d_2) = \sqrt{\sum_{i=1}^{n} (d_1[A_i] - d_2[A_i])^2},
\]

works well in this case.

2. **Manhattan distance.** If \( D \) has ordinal, but not necessarily continuous attributes, Manhattan distance may work a bit better:

\[
d(d_1, d_2) = \sum_{i=1}^{n} |d_1[A_i] - d_2[A_i]|.
\]

3. **Cosine similarity.** Cosine distance between two vectors is the cosine of the angle between them. Cosine similarity ignores the amplitude of the vectors, and measures only the difference in their *direction*:

\[
sim(d_1, d_2) = \cos(d_1, d_2) = \frac{d_1 \cdot d_2}{||d_1|| \cdot ||d_2||} = \frac{\sum_{i=1}^{n} d_1[A_i] \cdot d_2[A_i]}{\sqrt{\sum_{i=1}^{n} d_1[A_i]^2} \cdot \sqrt{\sum_{i=1}^{n} d_2[A_i]^2}}.
\]

If \( d_1 \) and \( d_2 \) are *colinear* (have the same direction), \( \sim(d_1, d_2) = 1 \). If \( d_1 \) and \( d_2 \) are *orthogonal*, \( \sim(d_1, d_2) = 0 \).
Ensemble Learning

**Bagging**

Bagging = Bootstrap aggregating.

**Bootstrapping** is a statistical technique that one to gather many alternative versions of the single statistic that would ordinarily be calculated from one sample.

**Typical bootstrapping scenario.** (case resampling) Given a sample $D$ of size $n$, a bootstrap sample of $D$ is a sample of $n$ data items drawn randomly with replacement from $D$.

**Note:** On average, about 63.2% of items from $D$ will be found in a bootstrapping sample, but some items will be found multiple times.

**Bootstrap Aggregating for Supervised Learning.** Let $D$ be a training set, $|D| = N$. We construct a bagging classifier for $D$ as follows:

**Training Stage:** Given $D$, $k$ and a learning algorithm $\text{BaseLearner}$:

1. Create $k$ bootstrapping replications $D_1, \ldots, D_k$ of $D$ by using case resampling bootstrapping technique.
2. For each bootstrapping replication $D_i$, create a classifier $f_i$ using the $\text{BaseLearner}$ classification method.

**Testing Stage:** Given $f_1, \ldots, f_k$ and a test record $d$:

1. Compute $f_1(d), \ldots, f_k(d)$.
2. Assign as $\text{class}(d)$, the majority (plurality) class among $f_1(d), \ldots, f_k(d)$.

**Boosting**

**Boosting** is a collection of techniques that generate an ensemble of classifiers in a way that each new classifier tries to correct classification errors from the previous stage.

**Idea.** Boosting is applied to a specific classification algorithm called $\text{BaseLearner}$.

Each item $d \in D$ is assigned a weight. On first step, $w(d) = \frac{1}{|D|}$. On each step, a classifier $f_i$ is built. Any errors of classification, i.e., items $d \in D$, such that $f(d) \neq \text{class}(d)$ are given higher weight.

On the next step, the classification algorithm is made to “pay more attention” to items in $D$ with higher weight.

The final classifier is constructed by weighting the votes of $f_1, \ldots, f_k$ by their weighted classification error rate.

**AdaBoost.** The Adaptive Boosting algorithm [2] (AdaBoost) is shown in Figure 3.

\[^1\text{It is also commonly called weak classifier.}\]
Algorithm AdaBoost($D, \text{BaseLearner}, k$) begin

foreach $d_i \in D$ do $D_1(i) = \frac{1}{|D|}$;

for $t = 1$ to $k$ do //main loop

$f_t := \text{BaseLearner}(D_t)$;

$e_t := \sum_{\text{class}(d_i) \neq f_t(d_i)} D_t(i)$; // $f_t$ is constructed to minimize $e_t$

if $e_t > 0.5$ then // large error: redo

$k := k - 1$;

break;
endif

$\alpha_t := \frac{1}{2} \ln \frac{1-e_t}{e_t}$; //reweighting parameter

foreach $d_i \in D$ do $D_{t+1}(i) := D_t(i) \cdot e^{-\alpha_t \cdot \text{class}(d_i) \cdot f_t(d_i)}$; //reweigh each tuple in $D$

$\text{Norm}_t := \sum_{i=1}^{\mid D \mid} D_{t+1}(i)$;

foreach $d_i \in D$ do $D_{t+1}(i) := \frac{D_{t+1}(i)}{\text{Norm}_t}$; //normalize new weights

endfor

endfor

$f_{\text{final}}(.) := \text{sign}(\sum_{t=1}^{k} \alpha_t \cdot f_t(.) )$

end

Figure 3: AdaBoost: an adaptive boosting algorithm. This version is for binary category variable $Y = \{-1, +1\}$.

Weak Classifiers. Some classifiers are designed to incorporate the weights of training set elements into consideration. But most, like C4.5, do not do so. In order to turn a classifier like C4.5 into a weak classifier suitable for AdaBoost, this classifier can be updated as follows:

- On step $t$, given the weighted training set $D_t$, we sample $D_t$ to build a training set $D'_t$. The sampling process uses $D_t(i)$ as the probability of selection of $d_i$ into $D'_t$ on each step.

Voting

When multiple classification algorithms $A_1, \ldots, A_k$ are available, direct voting can be used to combine these classifiers.

Let $D$ be a training set, and $f_1, \ldots, f_k$ are the classifiers produced by $A_1, \ldots, A_k$ respectively on $D$. Then the combined classifier $f$ is constructed to return the class label returned by the plurality of classifiers $f_1, \ldots, f_k$.

Random Forests

Random Forests[1] are an extension of bagging. A bagging technique resamples the training set with replacement, but keeps all attributes in the dataset "active" for each resampled training set.

Random Forests build a collection of decision trees, where each decision tree is built based on a subset of a training set and a subset of attributes.

In a nutshell, a Random Forests classifier works as follows:

1. Input: Let $D = \{d_1, \ldots, d_n\}$ be the training set, with class($d_i$) defined. Let $C = \{c_1, \ldots, c_k\}$ be the class attribute, and let $A = \{A_1, \ldots, A_N\}$ be the set of attributes for vectors from $D$, i.e., given $d \in D$, $d = (x_1, \ldots, x_M)$. 
2. **Attribute selection parameter:** A number $m \ll M$ is fixed throughout the run of a random forest classifier. This number indicates how many attributes is selected to build each decision tree in a forest.

3. **Forest construction:** The classifier builds $N$ decision trees $T_1, \ldots, T_N$. Each decision tree is built by selecting a subsample of the training set, and a subset of the attributes.

4. **Single decision tree construction:** Decision tree $T_j$ is built as follows.
   
   (a) Build a set $D_j \subseteq D$ drawing random $k$ data points from $D$ with replacement.
   
   (b) Select $m$ random attributes $A^j_1, \ldots, A^j_m$ from $A$ without replacement.
   
   (c) Using a decision tree induction procedure (see below), build a decision tree $T_j$ for the training set $D_j$ restricted to attributes $A^j_1, \ldots, A^j_m$.
   
   Do not prune the decision trees.

5. **Classification process:** For each data point $d \in D$, (attempt to) classify $d$ by traversing trees $T_1, \ldots, T_N$ to discover classification decisions $c^1, \ldots, c^N$. Choose, as $\text{class}(d)$, the most frequently occurring in $c^1, \ldots, c^N$ class.

**Caveats.** A decision tree $T_j$ may not contain all possible values (paths) for some attribute. This means that some trees won’t be able to classify some of the data points in $D$. The simplest way to deal with this is to ignore. Another way is to follow the description on how to deal with "ghost" paths.

**Decision tree induction procedures.** Both versions of ID3 (C4.5 without the pruning) and CART, a decision-tree induction algorithm that uses the Gini impurity instead of Information Gain-based measures, can be used.

**Gini impurity measure.** The Gini impurity measure quantifies how often a randomly chosen and randomly labelled data point from a training set will be mislabelled.

Let $D = \{d_1, \ldots, d_n\}$ be a training set.

Let $C = \{c_1, \ldots, c_k\}$ be a class variable.

Let $D_i = \{d \in D | \text{class}(d) = c_i\}$ be the set of all training set points from category $c_i$.

Let $f_i = |D_i|$.

The Gini impurity measure $I_G$ is defined as follows:

$$I_G(D) = \sum_{i=1}^{k} f_i \cdot (1 - f_i) = \sum_{i=1}^{k} f_i - \sum_{i=1}^{k} f_i^2 = 1 - \sum_{i=1}^{k} f_i^2 = \sum_{i \neq j} f_i \cdot f_j.$$ 

**References**