

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 $A = \{A_1, \dots, A_n\}$. Let $A_i \in A$ be a **continuous attribute**.

A **binary split** of D on attribute A_i at value α 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 α is

$$\text{entropy}_{A_i, \alpha}(D) = -\frac{|D^-|}{|D|} \cdot \text{entropy}(D^-) - \frac{|D^+|}{|D|} \cdot \text{entropy}(D^+).$$

```

function selectSplittingAttribute(A, D, threshold); //uses information gain
begin
  p0 := entropy(D);
  for each  $A_i \in A$  do
    if  $A_i$  is continuous then
      x := findBestSplit( $A_i, D$ );
      p[ $A_i$ ] := entropy $_{A_i, x}(D)$ ;
    else
      p[ $A_i$ ] := entropy $_{A_i}(D)$ ;
    endif
    Gain[ $A_i$ ] = p0 - p[ $A_i$ ]; //compute info gain
  endfor
  best := arg(findMax(Gain[]));
  if Gain[best] > threshold then return best
  else return NULL;
end

```

```

function findBestSplit( $A_i, D$ ) //finds best binary split for a continuous attribute
begin
  initialize associative arrays counts1[], ..., countsk[];
  initialize associative array Gain;
  p0 := entropy(D);
  foreach  $d \in D$  do //Step 1: scan data
    for j = 1 to k do
      if class(d) ==  $c_j$  then
        countsj[d[ $A_i$ ]] := countsj[d[ $A_i$ ]] + 1;
      else
        countsj[d[ $A_i$ ]] := countsj[d[ $A_i$ ]] + 0; // instantiates countsj[d[ $A_i$ ]]
      endif
    endfor
  endfor
  foreach x: index of instance of countsi do
    //computes entropy of binary split at x
    Gain[x] := p0 - entropy(D,  $A_i, x, counts_i, \dots, counts_k$ );
  endfor
  best := arg(findMax(Gain[]));
  return 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 α is:

$$\text{Gain}_{A_i, \alpha}(D) = \text{entropy}(D) - \text{entropy}_{A_i, \alpha}(D).$$

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 $\text{dom}(A_i)$ is continuous, D contains finitely many distinct values of A_i !
 - For each value x in of A_i from D find $\text{entropy}_{A_i, x}(D)$.
 - 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)

C4.5. and Overfitting

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

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

Casuses 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, SVNs, 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 (*k*NN).** *k*NN is a simple, but surprisingly robust **lazy evaluation** algorithm. The idea behind *k*NN is as follows:

- The input of the algorithm is a training set $D_{training}$, 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, \dots, d_k , $d_i \in D$.
- The algorithm assigns to d the class of the plurality of items from the list d_1, \dots, 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:

$$d(d_1, d_2) = \sum_i^n |d_1[A_i] - d_2[A_i]|.$$

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

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

If d_1 and d_2 are *colinear* (have the same direction), $\text{sim}(d_1, d_2) = 1$. If d_1 and d_2 are *orthogonal*, $\text{sim}(d_1, d_2) = 0$.

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 **BaseLearner**:

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

Testing Stage: Given f_1, \dots, f_k and a test record d :

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

Boosting

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.

```

Algorithm AdaBoost( $D$ , BaseLerner,  $k$ ) begin
  foreach  $d_i \in D$  do  $D_1(i) = \frac{1}{|D|}$ ;
  for  $t = 1$  to  $k$  do //main loop
     $f_t :=$ BaseLerner( $D_t$ );
     $e_t := \sum_{class(d_i) \neq f_t(d_i)} D_t(i)$ ;
    //  $f_t$  is constructed to minimize  $e_t$ 
    if  $e_t > 0.5$  then // large error: redo
       $k := k - 1$ ;
      break;
    endif
     $a_t := \frac{1}{2} \ln \frac{1-e_t}{e_t}$ ; //reweighting parameter
    foreach  $d_i \in D$  do  $D_{t+1}(i) := D_t(i) \cdot e^{-\alpha_t \cdot class(d_i) \cdot f_t(d_i)}$ ; //reweigh each tuple in  $D$ 
     $Norm_t := \sum_{i=1}^{|D|} D_{t+1}(i)$ ;
    foreach  $d_i \in D$  do  $D_{t+1}(i) := \frac{D_{t+1}(i)}{Norm_t}$ ; //normalize new weights
  endfor

   $f_{final}(\cdot) := sign(\sum_{t=1}^k a_t \cdot f_t(\cdot))$ 
end

```

Figure 2: **AdaBoost**: an adaptive boosting algorithm. This version is for binary category variable $Y = \{-1, +1\}$.

Idea. **Boosting** is applied to a specific classification algorithm called **BaseLerner**¹.

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 $f(d) \neq class(d)$ 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, \dots, f_k by their weighted classification error rate.

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

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 $\mathcal{A}_1, \dots, \mathcal{A}_k$ are available, **direct voting** can be used to combine these classifiers.

¹It is also commonly called weak classifier.

Let D be a training set, and f_1, \dots, f_k are the classifiers produced by $\mathcal{A}_1, \dots, \mathcal{A}_k$ respectively on D . Then the combined classifier f is constructed to return the class label returned by the **plurality** of classifiers f_1, \dots, 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 $D = \{d_1, \dots, d_n\}$ be the training set, with $class(d_i)$ defined. Let $C = \{c_1, \dots, c_k\}$ be the class attribute, and let $A = \{A_1, \dots, A_N\}$ be the set of attributes for vectors from D , i.e., given $d \in D$, $d = (x_1, \dots, x_M)$.
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, \dots, 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_1^j, \dots, A_m^j 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_1^j, \dots, A_m^j .
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, \dots, T_N to discover classification decisions c^1, \dots, c^N . Choose, as $class(d)$, the most frequently occurring in c^1, \dots, 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.

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

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

Let $D_i = \{d \in D \mid \text{class}(d) = c_i\}$ 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:

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

References

- [1] Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5-32.
- [2] Y. Freund, R.E. Shapire. Experiments with a New Boosting Algorithm. In *Proceedings, 13th International Conference on Machine Learning (ICML'96)*, pp. 148–156, 1996.