KNN and Decision Tree

1. KNN Overview

Step 1: Compute distance to other training records (e.g. euclidean distance).

Step 2: Identity k nearest neighbours.

Step 3: Use class labels of nearest neighbours to determine the class label of unknown records (using majority vote, weight the vote according to distance)

Notice:

If k is too small $\rightarrow$ sensitive to noise points

If k is too large $\rightarrow$ neighbourhood may include points from other classes.

Overall, KNN tries to classify an unlabelled observation based on it's k surrounding neighbours. It is also known as lazy learner because it involves minimal training of model. Hence, it doesn't use training data make generalisation on unseen data set.

Issues with KNN:

1. Scaling issues (distance measures from being dominated by one of the attributes)

   example: income of a person may vary from $10k to $1M.
2. High dimensional data (problem with euclidean measure)
3. Produce counter-intuitive results

   example:

   [1 1 1 1 1 0] and [0 1 1 1 1 1] has d=1.4142

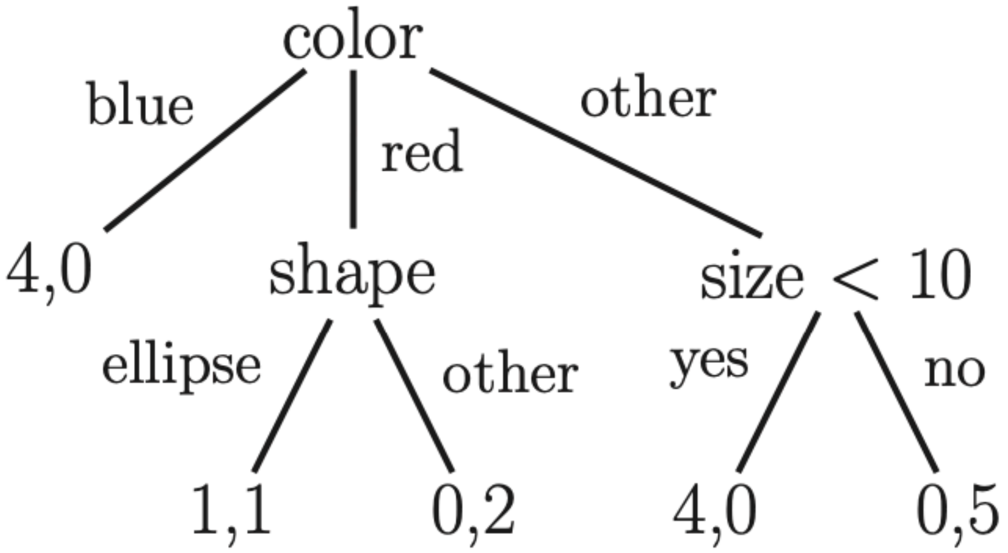
   [1 0 0 0 0 0] and [0 0 0 0 0 1] also has d = 1.4142

   solution: normalise the vectors to unit length

2. Decision Tree Overview

Decision trees are also called Classification and regression trees (CART) models. This family of models recursively partition the input space and define a local model in each resulting region of input space. This can be represented by a tree, with one leaf per region.

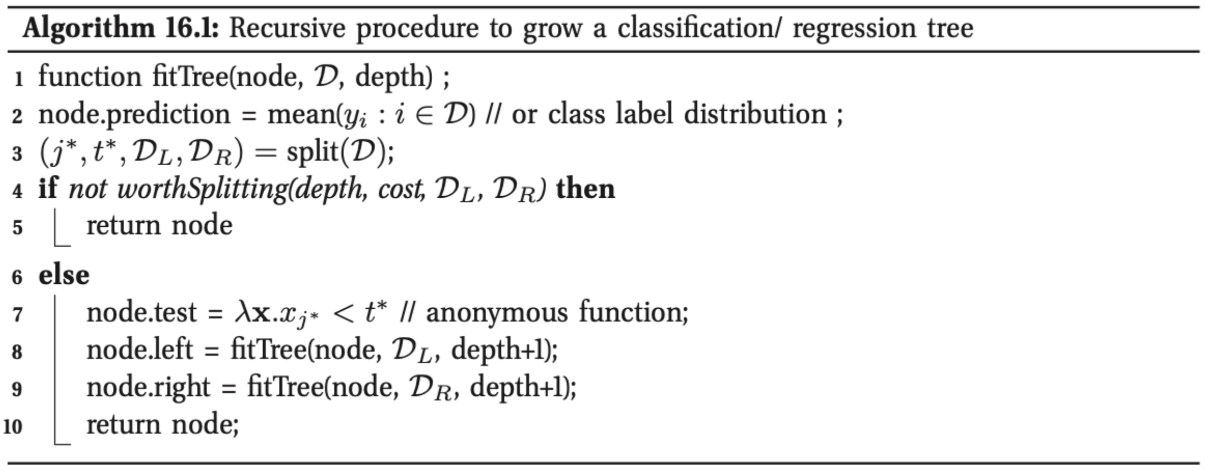
Example:



In the classification settings, we store the distribution over class labels in each leaf, instead of mean response.

(4,0) $\rightarrow$ 4 positive examples and 0 negative examples, hence we predict $p(y = 1 \mid x) = 4/4$ is x is blue.

If it is red, we then check the shape, if it is ellipse, we predict $p(y = 1 \mid x) = 1/2$.



Algorithm 16.1: Recursive procedure to grow a classification/ regression tree

```
function fitTree(node, $\mathcal{D}$, depth) ;
node.prediction = mean($y_i : i \in \mathcal{D}$) // or class label distribution ;
$(j^*, t^*, \mathcal{D}_L, \mathcal{D}_R) = \text{split}(\mathcal{D})$;
if not worthSplitting(depth, cost, $\mathcal{D}_L$, $\mathcal{D}_R$) then
    return node
else
    node.test = $\lambda \mathbf{x}.x_{j^*} < t^*$ // anonymous function;
    node.left = fitTree(node, $\mathcal{D}_L$, depth+1);
    node.right = fitTree(node, $\mathcal{D}_R$, depth+1);
    return node;
```

Regression cost: $cost(D) = \sum(y_i - \bar{y})^2$, where $\bar{y} = \frac{1}{|D|}\sum y_i$

Classification cost:

- Misclassification rate: $\frac{1}{|D|}\sum \mathbb{I}(y_i \neq \hat{y})$
- Entropy: $-\sum \hat{\pi}_c \log \hat{\pi}_c$, where $\hat{\pi}_c$ is the probability a random entry in the leaf belongs to class c.
- Gini index: $1 - \sum \hat{\pi}_c^2$

Pros:

- easy to interpret
- easy to handle mixed inputs (discrete and continuous)
- perform automatic variable selection
- relatively robust to outliers, scale well to large data sets
- can handle missing inputs

Cons:

- cannot predict very accurately (due to the greedy nature of the tree construction algorithm)
- unstable, small changes to the input data can have large effects on the structure of the tree.

Solution:

Random forests using bagging technique which train M different trees on different subsets of the data and chosen randomly with replacement.