Machine Learning
Decision Trees & Decision Trees-Based Models

Kien C Nguyen

August 7, 2024

INDUSTRIAL UNIVERSITY OF HOCHIMINH CITY

Contents

Decision Trees – Introduction

- Also called Classification And Regression Trees or CART models
- The input space is recursively partitioned, and a local model is defined in each resulting region.
- We can represent this structure with a tree, with one leaf per region.
- A mean output is associated with each of these regions
- We then have a piecewise constant surface.
- The model can be written as

$$f(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = \sum_{m=1}^{M} w_m \mathbb{I}(\mathbf{x} \in \mathbb{R}_m) = \sum_{m=1}^{M} w_m \phi(\mathbf{x}; \mathbf{v}_m)$$

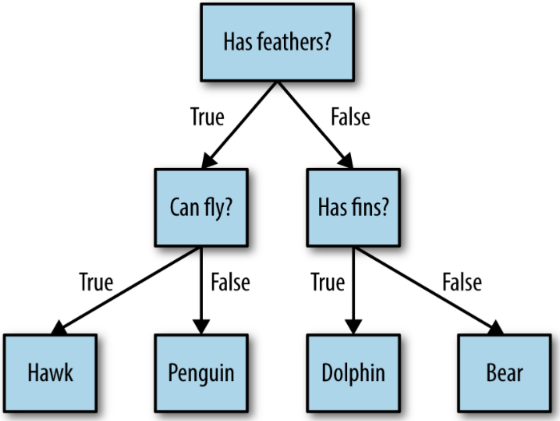
R_m is the m'th region, w_m is the mean output in this region, $\mathbf{v}_m$ encodes the choice of variable to split on and the threshold value, on the path from the root to the m'th leaf.

Decision Trees – Introduction

- Imagine you want to distinguish between the following four animals: bears, hawks, penguins, and dolphins.
- Your goal is to get to the right answer by asking as few if/else questions as possible.
- You might start off by asking whether the animal has feathers, a question that narrows down your possible animals to just two.
- If the answer is “yes,” you can ask another question that could help you distinguish between hawks and penguins.
- For example, you could ask whether the animal can fly.
- If the animal doesn’t have feathers, your possible animal choices are dolphins and bears, and you will need to ask a question to distinguish between these two animals—for example, asking whether the animal has fins.

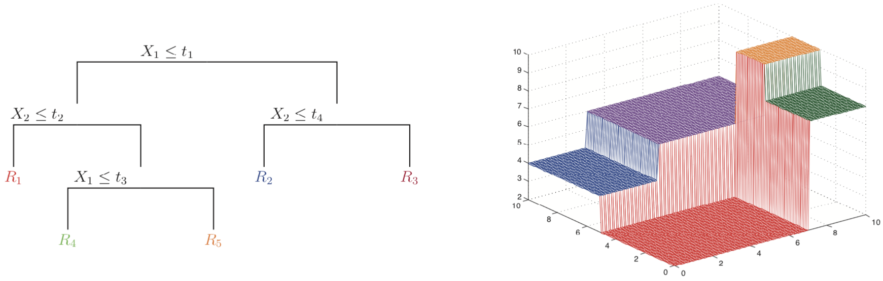
Decision Trees - Example 1

Figure: A decision tree to distinguish among several animals. (Source: Müller & Guido [1])



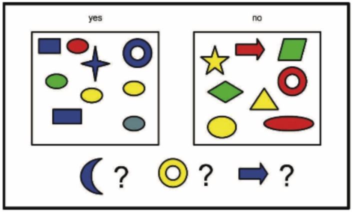
Decision Trees - Example 2

Figure: Decision Trees. (Source: K. Murphy [2])

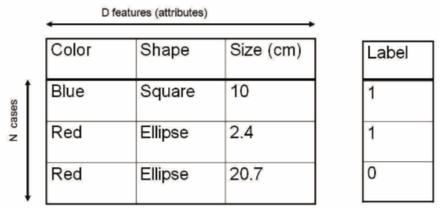


Decision Trees

Figure: Some labeled training examples of colored shapes, along with 3 unlabeled test cases. (Source: K. Murphy [1])



(a)

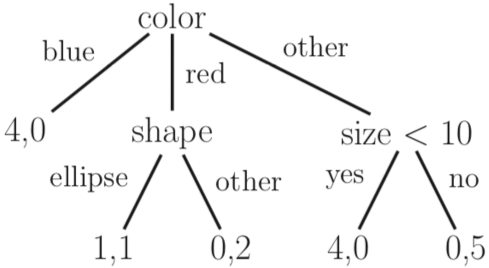

D features (attributes)
N cases

Color	Shape	Size (cm)	Label
Blue	Square	10	1
Red	Ellipse	2.4	1
Red	Ellipse	20.7	0

(b)

Decision Trees

Figure: A simple decision tree for the data in Figure 7. A leaf labeled as (n_1, n_0) means that there are n_1 positive examples that match this path, and n_0 negative examples. In this tree, most of the leaves are "pure", meaning they only have examples of one class or the other; the only exception is leaf representing red ellipses, which has a label distribution of (1, 1). We could distinguish positive from negative red ellipses by adding a further test based on size. However, it is not always desirable to construct trees that perfectly model the training data, due to overfitting. (Source: K. Murphy [1])



Decision Trees

- We split a tree so as to minimize a cost function
- Finding the optimal partitioning of the data is NP-complete
- A greedy approach is as follows
- The split function chooses the best feature, and the best value for that feature:

$$(j^*, t^*) = \arg \min_{j \in \{1,\ldots,D\}} \min_{t \in \mathcal{T}_j} \text{cost}\left(\{\mathbf{x}_i, y_i : x_{ij} \leq t\}\right) + \text{cost}\left(\{\mathbf{x}_i, y_i : x_{ij} > t\}\right. \tag{1}$$

- A normalized measure of the reduction in cost

$$\Delta \triangleq \text{cost}(\mathcal{D}) - \left(\frac{|\mathcal{D}_L|}{|\mathcal{D}|} \text{cost}\left(\mathcal{D}_L\right) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \text{cost}\left(\mathcal{D}_R\right) \right) \tag{2}$$

Decision Trees – Regression cost

- The cost can be written as

$$C(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \bar{y})^2$$

  where $\bar{y} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} y_i$ is the mean output in set $\mathcal{D}$
- Alternatively, we can also use a linear regression model for each leaf, using the features that have been used on the path from the root to the leaf and calculate the residual errors.

Decision Trees – Classification cost

- The class-conditional probabilities are given as

$$\hat{\pi}_c = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}\{y_i = c\}$$

- The most probable class label is given by $\hat{y}_c = argmax_c \ \hat{\pi}_c$

Given these definitions, there are several common error measures for evaluating a proposed partition.

Decision Tree Regression – Classification cost

- Misclassification rate:

$$\frac{1}{|\mathcal{D}|}\sum_{i\in\mathcal{D}}\mathbb{I}\{y_i \neq \hat{y}\} = 1 - \hat{\pi}_c$$

- Entropy

$$\mathbb{H}(\hat{\pi}) = -\sum_{c=1}^{C} \hat{\pi}_c log \hat{\pi}_c$$

- Gini index:

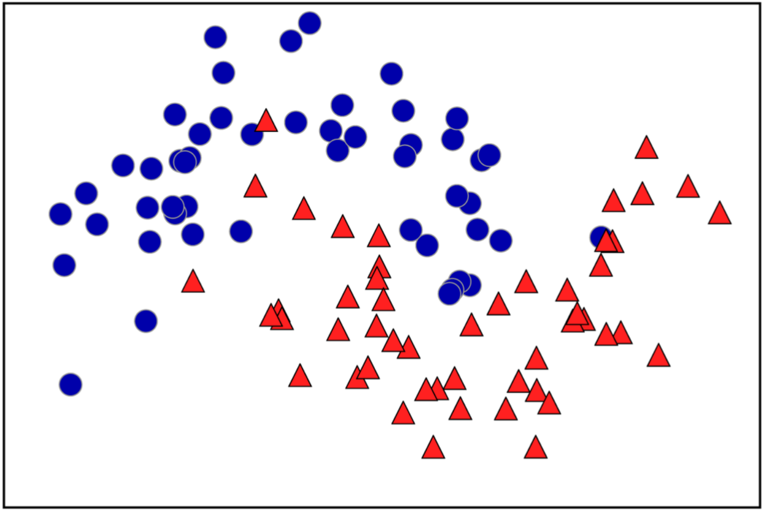
$$\sum_{c=1}^{C} \hat{\pi}_c(1-\hat{\pi}_c) = 1 - \sum_{c=1}^{C} \hat{\pi}_c^2$$

Building decision trees

- Let's go through the process of building a decision tree for the 2D classification dataset.
- The dataset consists of two half-moon shapes, with each class consisting of 50 data points.
- We will refer to this dataset as two_moons.
- Learning a decision tree means learning the sequence of if/else questions that gets us to the true answer most quickly.
- In the machine learning setting, these questions are called tests
- Usually data does not come in the form of binary yes/no features as in the animal example, but is instead represented as continuous features such as in the 2D dataset shown.
- The tests that are used on continuous data are of the form "Is feature i larger than value a?"

Building decision trees

Figure: Two_moons dataset. (Source: Müller & Guido [1])

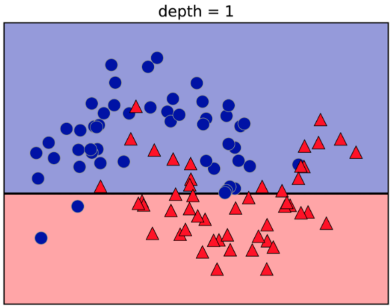


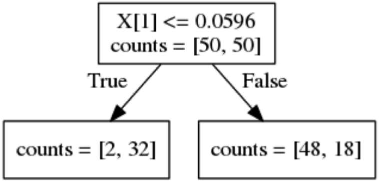
Building decision trees

- To build a tree, the algorithm searches over all possible tests and finds the one that is most informative about the target variable.
- Splitting the dataset vertically at x[1]=0.0596 yields the most information; it best separates the points in class 1 from the points in class 2.
- The top node, also called the root, represents the whole dataset, consisting of 50 points belonging to class 0 and 50 points belonging to class 1.
- The split is done by testing whether x[1] ¡= 0.0596, indicated by a black line.
- If the test is true, a point is assigned to the left node, which contains 2 points belonging to class 0 and 32 points belonging to class 1.
- Otherwise the point is assigned to the right node, which contains 48 points belonging to class 0 and 18 points belonging to class 1.

Building decision trees

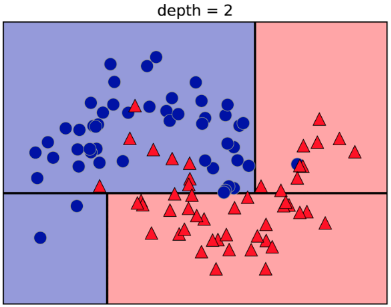
Figure: Two_moons dataset. (Source: Müller & Guido [1])

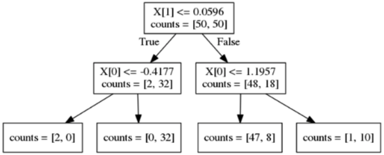




Building decision trees

Figure: Decision boundary of tree with depth 2 (left) and corresponding decision tree (right). (Source: Müller & Guido [1])



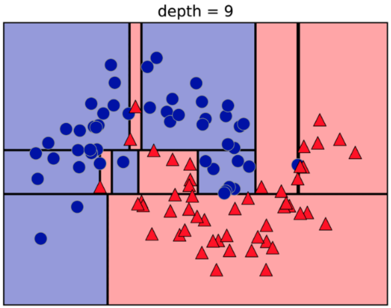


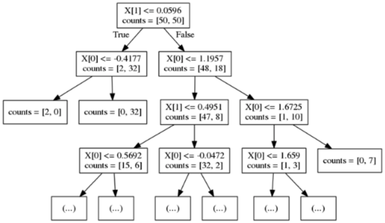
Building decision trees

- The recursive partitioning of the data is repeated until each region in the partition (each leaf in the decision tree) only contains a single target value (a single class or a single regression value).
- A leaf of the tree that contains data points that all share the same target value is called pure. The final partitioning for this dataset is shown in the next slide

Building decision trees

Figure: Decision boundary of tree with depth 9 (left) and part of the corresponding tree (right) (Source: Müller & Guido [1])





Decision Trees - Classification examples

Residential status has three attributes with the numbers of goods and bads in each attribute in a sample of a previous customer, shown in Table. If one wants to split the tree on this characteristic, what should the split be?

Residential status	Owner	Tenant	With Parents
Number of goods	1000	400	80
Number of bads	200	200	120
Good:bad odds	5 : 1	2 : 1	0.67 : 1

Decision Trees - Using entropy for classification cost

$l = \text{parent}, \quad r = \text{owner} + \text{tenant}:$

$$i(v) = -\left(\frac{520}{2000}\right)\ln\left(\frac{520}{2000}\right) - \left(\frac{1480}{2000}\right)\ln\left(\frac{1480}{2000}\right) = 0.573,$$

$$p(l) = \frac{200}{2000} = 0.1, \quad i(l) = -\left(\frac{80}{200}\right)\ln\left(\frac{80}{200}\right) - \left(\frac{120}{200}\right)\ln\left(\frac{120}{200}\right) = 0.673,$$

$$p(r) = \frac{1800}{2000} = 0.9, \quad i(r) = -\left(\frac{400}{1800}\right)\ln\left(\frac{400}{1800}\right) - \left(\frac{1400}{1800}\right)\ln\left(\frac{1400}{1800}\right) = 0.530,$$

$$I = 0.573 - 0.1(0.673) - 0.9(0.530) = 0.0287;$$

Decision Trees - Using entropy for classification cost

$l = \text{parent} + \text{tenant}, \quad r = \text{owner}:$

$$i(v) = -\left(\frac{520}{2000}\right)\ln\left(\frac{520}{2000}\right) - \left(\frac{1480}{2000}\right)\ln\left(\frac{1480}{2000}\right) = 0.573,$$

$$p(l) = \frac{800}{2000} = 0.4, \quad i(l) = -\left(\frac{320}{800}\right)\ln\left(\frac{320}{800}\right) - \left(\frac{480}{800}\right)\ln\left(\frac{480}{800}\right) = 0.673,$$

$$p(r) = \frac{1200}{2000} = 0.6, \quad i(r) = -\left(\frac{200}{1200}\right)\ln\left(\frac{200}{1200}\right) - \left(\frac{1000}{1200}\right)\ln\left(\frac{1000}{1200}\right) = 0.451,$$

$$I = 0.573 - 0.4(0.673) - 0.6(0.451) = 0.0332.$$

Decision Trees - Using gini index for classification cost

$l = \text{parent}, \quad r = \text{owner} + \text{tenant}:$

$$i(v) = \left(\frac{1480}{2000}\right)\left(\frac{520}{2000}\right) = 0.1924,$$

$$p(l) = \frac{200}{2000} = 0.1, \quad i(l) = \left(\frac{80}{200}\right)\left(\frac{120}{200}\right) = 0.24,$$

$$p(r) = \frac{1800}{2000} = 0.9, \quad i(r) = \left(\frac{400}{1800}\right)\left(\frac{1400}{1800}\right) = 0.1728,$$

$$I = 0.1924 - 0.1(0.24) - 0.9(0.1728) = 0.01288;$$

Decision Trees - Using gini index for classification cost

$$l = \text{parent} + \text{tenant}, \quad r = \text{owner}:$$

$$i(v) = \left(\frac{520}{2000}\right)\left(\frac{1480}{2000}\right) = 0.1924,$$

$$p(l) = \frac{800}{2000} = 0.4, \quad i(l) = \left(\frac{320}{800}\right)\left(\frac{480}{800}\right) = 0.24,$$

$$p(r) = \frac{1200}{2000} = 0.6, \quad i(r) = \left(\frac{200}{1200}\right)\left(\frac{1000}{1200}\right) = 0.1389,$$

$$I = 0.1924 - 0.4(0.24) - 0.6(0.1389) = 0.01306.$$

Decision Tree Classification

```
# Splitting the dataset
from sklearn.model_selection\
 import train_test_split
X_train, X_test, y_train, y_test\
 = train_test_split(X, y, test_size = 0.25,\
  random_state = 0)
```

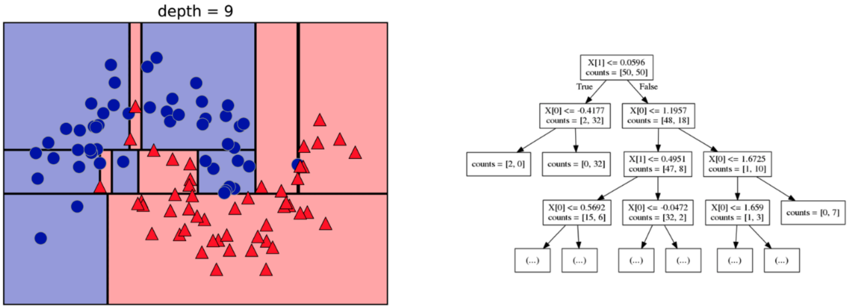
Decision Tree Classification

```
# Fitting Decision Tree Classifier to the test set
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(
        criterion = 'entropy',
         random_state = 0)
classifier.fit(X_train, y_train)

# Predicting the test set
y_pred = classifier.predict(X_test)
```

Building decision trees

Figure: Decision boundary of tree with depth 9 (left) and part of the corresponding tree (right) (Source: Müller & Guido [1])



Controlling complexity of decision trees

- Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data.
- The presence of pure leaves mean that a tree is 100% accurate on the training set; each data point in the training set is in a leaf that has the correct majority class.
- The overfitting can be seen on the left.
- The regions determined to belong to class 1 in the middle of all the points belonging to class 0.
- On the other hand, there is a small strip predicted as class 0 around the point belonging to class 0 to the very right.
- This is not how one would imagine the decision boundary to look, and the decision boundary focuses a lot on single outlier points that are far away from the other points in that class.

Preventing overfitting

- There are two common strategies to prevent overfitting:
  - Stopping the creation of the tree early (also called pre-pruning),
  - Building the tree but then removing or collapsing nodes that contain little information (also called post-pruning or just pruning).
- Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.
- Decision trees in scikit-learn are implemented in the DecisionTreeRegressor and DecisionTreeClassifier classes. scikit-learn only implements pre-pruning, not post-pruning.

Preventing overfitting

- Let's look at the effect of pre-pruning in more detail on the Breast Cancer dataset.
- We import the dataset and split it into a training and a test part.
- Then we build a model using the default setting of fully developing the tree (growing the tree until all leaves are pure).

No pre-pruning

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection\
                                    import train_test_split
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = \
        train_test_split(
    cancer.data, cancer.target,
                         stratify=cancer.target,
                         random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(
        tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(
        tree.score(X_test, y_test)))
```

Accuracy on training set: 1.000
Accuracy on test set: 0.937

Pre-pruning

- If we don't restrict the depth of a decision tree, the tree can become arbitrarily deep and complex.
- Unpruned trees are therefore prone to overfitting and not generalizing well to new data.
- Now let's apply pre-pruning to the tree, which will stop developing the tree before we perfectly fit to the training data.
- One option is to stop building the tree after a certain depth has been reached. Here we set max_depth=4, meaning only four consecutive questions can be asked.
- Limiting the depth of the tree decreases overfitting.
- This leads to a lower accuracy on the training set, but an improvement on the test set:

With pre-pruning

```
tree = DecisionTreeClassifier(max_depth=4,
        random_state=0) tree.fit(
                X_train, y_train)
print("Accuracy on training set: {:.3f}".format(
        tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(
        tree.score(X_test, y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Analyzing decision trees

- We can visualize the tree using the export_graphviz function from the tree module.
- This writes a file in the .dot file format, which is a text file format for storing graphs.

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot",
        class_names=["malignant", "benign"],
feature_names=cancer.feature_names,
        impurity=False,
        filled=True)
```

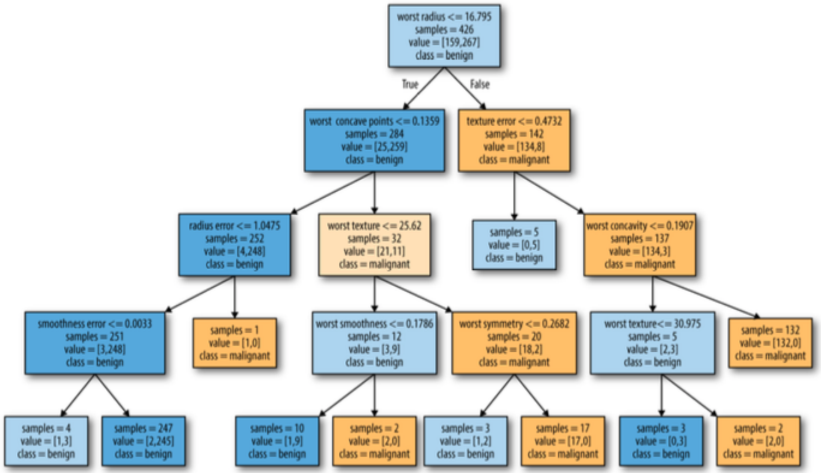
Analyzing decision trees

- We can read this file and visualize it using the graphviz module

```
import graphviz
with open("tree.dot") as f: dot_graph = f.read()
graphviz.Source(dot_graph)
```

Analyzing decision trees

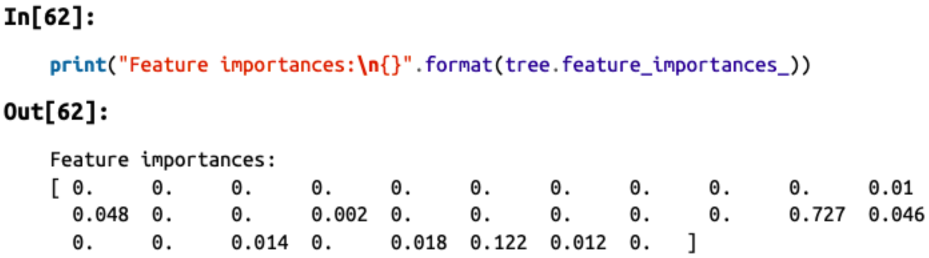
Figure: Analyzing decision trees (Source: Müller & Guido [1])



Feature importance in trees

- Instead of looking at the whole tree, which can be taxing, there are some useful properties that we can derive to summarize the workings of the tree.
- The most commonly used summary is feature importance, which rates how important each feature is for the decision a tree makes.
- It is a number between 0 and 1 for each feature, where 0 means "not used at all" and 1 means "perfectly predicts the target." The feature importances always sum to 1

Feature importances



In[62]:

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

Out[62]:

```
Feature importances:
[ 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.01
  0.048  0.     0.     0.002  0.     0.     0.     0.     0.     0.727  0.046
  0.     0.     0.014  0.     0.018  0.122  0.012  0.    ]
```

Feature importances

```
def plot_feature_importances_cancer(model):
n_features = cancer.data.shape[1]
plt.barh(range(n_features),
        model.feature_importances_, align='center')
plt.yticks(np.arange(n_features),
cancer.feature_names)
plt.xlabel("Feature importance")
plt.ylabel("Feature")
plot_feature_importances_cancer(tree)
```

Feature importances

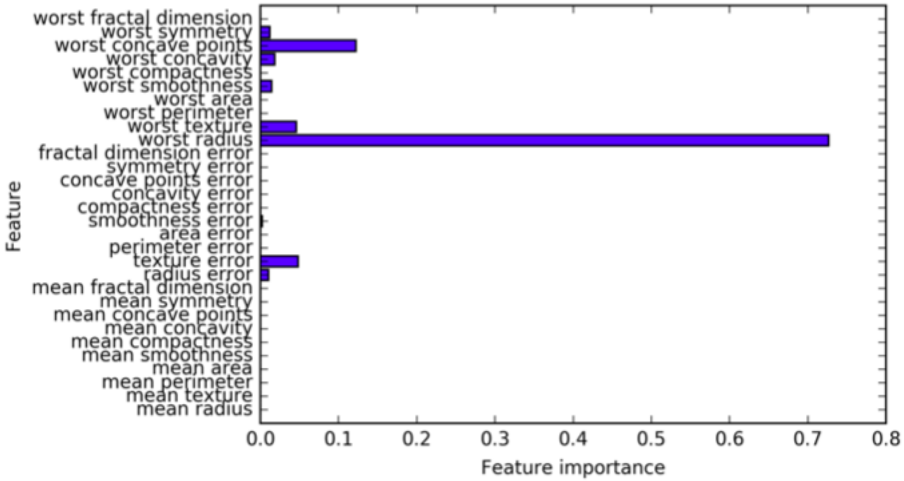


Figure: Feature importances computed from a decision tree learned on the Breast Cancer dataset (Source: Müller & Guido [1])

Decision Tree Classification – Some parameters

- **criterion** : string, optional (default="gini") The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.
- **max_depth** : int or None, optional (default=None) The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split** : int, float, optional (default=2) The minimum number of samples required to split an internal node:
  If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a fraction and $ceil(min\_samples\_split * n\_samples)$ are the minimum number of samples for each split.

Decision Tree Classification – Some parameters

- **min_samples_leaf** : int, float, optional (default=1) The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
  If int, then consider min_samples_leaf as the minimum number. If float, then min_samples_leaf is a fraction and $ceil(min_samples_leaf * n_samples)$are the minimum number of samples for each node. Changed in version 0.18: Added float values for fractions.
- **min_weight_fraction_leaf** : float, optional (default=0.) The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- **max_features** : int, float, string or None, optional (default=None) The number of features to consider when looking for the best split:

Contents

Bagging

- We can reduce the variance (without increasing the bias) of an estimate by averaging together multiple estimates.
- While the predictions of a single tree are highly sensitive to noise in its training set, the average of many trees is not, as long as the trees are not correlated (Wikipedia).
- For example, we can train N different trees on different subsets of the data, chosen randomly with replacement, and then compute the ensemble

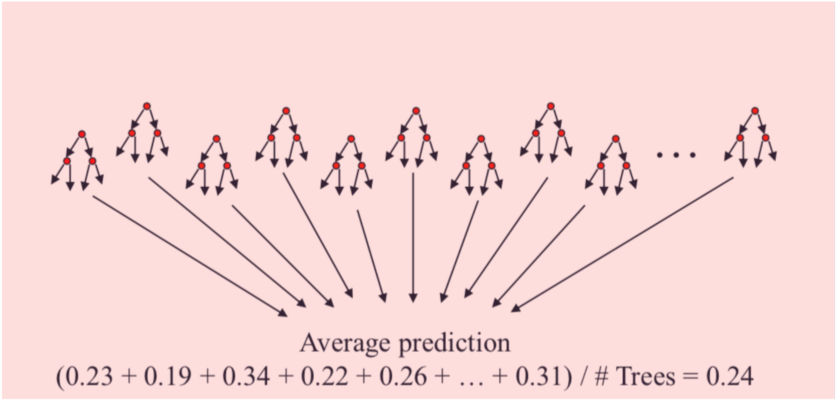
$$f(\mathbf{x}) = \sum_{n=1}^{N} \frac{1}{N} f_n(\mathbf{x})$$

  where f_m is the m'th tree.
- This technique is known as **bagging**, which stands for "bootstrap aggregating".

Bagging

- Draw 100 bootstrap samples of data
- Train a tree on each sample (100 trees)
- Average prediction of trees on out-of-bag samples

Figure: An example of bagging. (Source: UIUC CS446 Lecture notes)


Average prediction
(0.23 + 0.19 + 0.34 + 0.22 + 0.26 + … + 0.31) / # Trees = 0.24

Random Forest

- Unfortunately, simply re-running the same learning algorithm on different subsets of the data can result in highly correlated predictors, which limits the amount of variance reduction that is possible.
- The technique known as random forests tries to decorrelate the base learners by learning trees based on a randomly chosen subset of features, as well as a randomly chosen subset of data cases.
- This process is sometimes called "feature bagging". The reason for doing this is the correlation of the trees in an ordinary bootstrap sample: if one or a few features are very strong predictors for the response variable (target output), these features will be selected in many of the trees, causing them to become correlated (Wikipedia).
- Such models often have very good predictive accuracy, and have been widely used in many applications

Random Forest

- Draw N bootstrap samples of data
- Draw sample of available attributes at each split
- Train trees on each sample/attribute set (N trees)
- Average prediction of trees on out-of-bag samples

Random Forest Classification

```
# Fitting Random Forest Classifier
# to the training set
from sklearn.ensemble import\
 RandomForestClassifier
classifier = RandomForestClassifier(
    n_estimators = 10,
    criterion = 'entropy',
    random_state = 0)
classifier.fit(X_train, y_train)

# Predicting the test set
y_pred = classifier.predict(X_test)
```

Random Forest Classification – Some parameters

- **n_estimators** : integer, optional (default=10) The number of trees in the forest.
- **criterion** : string, optional (default="gini") The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.
- **max_depth** : integer or None, optional (default=None) The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split** : int, float, optional (default=2) The minimum number of samples required to split an internal node: If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a fraction and $ceil(min\_samples\_split * n\_samples)$ are the minimum number of samples for each split.

Contents

Boosting

- An **adaptive basis-function model** (ABM) is a model of the form

$$f(x) = w_0 + \sum_{m=1}^{M} w_m \phi_m(x)$$

  where $\phi_m(x)$ is the m's basis function, which is learned from data
- In **boosting**, ϕ_m are generated by an algorithm called a **weak learner** or a **base learner**.
- This weak learner can be any classification or regression algorithm, but it is common to use a CART model.

Boosting

- Initialization:
  - Weigh all training samples equally
- Iteration Step:
  - Train model on (weighted) train set
  - Compute error of model on train set
  - Increase weights on training cases model gets wrong
- Typically requires 100's to 1000's of iterations
- Return final model:
  - Carefully weighted prediction of each model

XGBoost Classification

```
# Fitting XGBoost to the training set
from xgboost import XGBClassifier
classifier = XGBClassifier()
classifier.fit(X_train, y_train)

# Predicting the test set
y_pred = classifier.predict(X_test)
```

XGBoost Classification – Some parameters

- **loss** : 'deviance', 'exponential', optional (default='deviance') loss function to be optimized. 'deviance' refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss 'exponential' gradient boosting recovers the AdaBoost algorithm.
- **learning_rate** : float, optional (default=0.1) learning rate shrinks the contribution of each tree by learning_rate. There is a trade-off between learning_rate and n_estimators.
- **n_estimators** : int (default=100) The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance.
- **subsample** : float, optional (default=1.0) The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. subsample interacts with the parameter n_estimators. Choosing subsample ¡ 1.0 leads to a reduction of variance and an increase in bias.

Contents

References

[1] K. P. Murpy – Machine Learning – A Probabilistic Perspective, MIT Press, 2012.
[2] A C Müller, S. Guido, Introduction to Machine Learning with Python, O'Reilly Media, Inc., 2017
[3] UIUC CS 446 Machine Learning
[4] Udemy's Machine Learning, https://www.udemy.com/machinelearning/
[5] scikit-learn website – https://scikit-learn.org
[6] L. C. Thomas, D. B. Edelman, J. N. Crook Credit Scoring and Its Applications, Second Edition, 2017