LEARN DATA SCIENCE

Random Forests, Decision Trees, and Ensemble Methods Explained

Author: Dylan Storey / Posted on December 4, 2018

The random forest, first described by Breimen et al (2001), is an ensemble approach for building predictive models. The "forest" in this approach is a series of decision trees that act as "weak" classifiers that as individuals are poor predictors but in aggregate form a robust prediction. Due to their simple nature, lack of assumptions, and general high performance they've been used in probably every domain where machine learning has been applied.

Random forests don't make any strong assumptions about the scale and normality of incoming data. They perform well with mixed numerical and categorical data, don't require much tuning to get a reasonable first version of a predictive model, are fast to train, are intuitive to understand, provide

feature importance as a feature of the model, are inherently able to handle missing data, and have been implemented in every language. These characteristics make random forests a great starting point for any project where you're building a predictive model or exploring the feasibility of applying machine learning to a new domain.

What Is A Decision Tree?

In order to understand a random forest, some general background on decision trees is needed.

Classification and Regression Tree models, or CART models, were introduced by Breimen et al. In these models, a top down approach is applied to observation data. The general idea is that given a set of observations, the following question is asked: is every target variable in this set the same (or nearly the same)? If yes, label the set of observations with the most frequent class; if no, find the best rule that splits the observations into the purest set of observations.

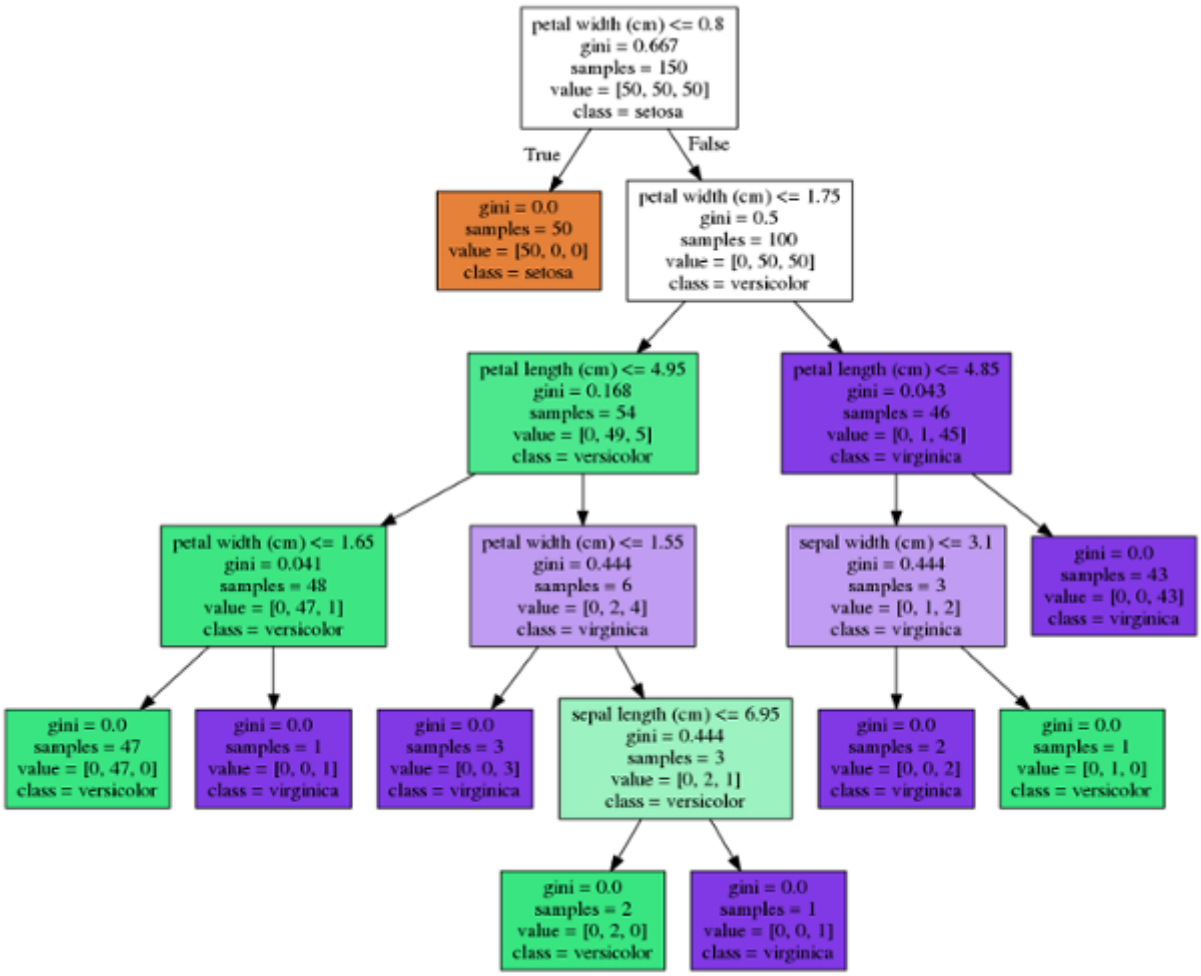
How It Works

Here is an example as applied to the iris data set:

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier,export_graphviz
import graphviz

iris_data = load_iris()
model = DecisionTreeClassifier()
model.fit(iris_data.data, iris_data.target)

dot = export_graphviz(model,
                      out_file=None,
                      feature_names=iris_data.feature_names,
```

```
                                class_names=iris_data.target_names,
                                filled=True,
                                impurity=None,
                                )

graph = graphviz.Source(dot)
graph.render("iris_decision_tree")
```

In this tree, the decision for determining the species of an iris is as follows:



To read this tree, start from the top white node, using the first line to determine how the decision was made to split the current observations into two new nodes. Now, we can use this decision tree to classify new observations.

Observation 1

A flower with a petal width of 0.7 , petal length of 1.0, and a sepal width of 3.0.

From the root node :

- Is the petal width < 0.8 ?
    - Yes -> go left.

The flower is from the species *setosa*.

Observation 2

A flower with a petal width of 0.9 , petal length of 1.0, and a sepal width of 3.0.

Solution

From the root node :

- Is the petal width less than or equal to 0.8 ?
    - No -> go right.
- Is the petal width less than or equal to 1.75 ?
    - Yes -> go left.
- Is the petal length less than or equal to 4.95 ?
    - Yes -> go left.
- Is the petal width less than or equal to 1.65 ?
    - Yes -> go left

The flower is from species *versicolor*.

Limitations To Decision Trees

While great for producing models that are easy to understand and implement, decision trees also tend to overfit on their training data—making them perform poorly if data shown to them later don't closely match to what they were trained on.

In the special case of regression trees, they also can only predict within the range of labels that they've seen before, which means that they have explicit upper and lower bounds on the numbers they can produce.

Ensemble Methods

Ensemble methods are algorithms that combine multiple algorithms into a single predictive model in order to decrease variance, decrease bias, or improve predictions.

Ensemble methods are usually broken into two categories:

1. **Parallel:** An ensemble method where the models that make up the building blocks of the larger methods are generated independent of each other (i.e., they can be trained/generated as trivially parallel problems applied to the dataset).

2. **Sequential:** An ensemble methods where the learners are generated in a sequential order and are dependent on each other (i.e., they can only be trained one at a time, as the next model will require information from the training upstream of it).

The random forest algorithm relies on a parallel ensemble method called "bagging" to generate its weak classifiers.

Bagging

Bagging is a colloquial term for bootstrap aggregation. Bootstrap aggregation is a method that allows us to decrease the variance of an estimate by averaging multiple estimates that are measured from random subsamples of a population.

Bootstrap Sampling

The first portion of bagging is the application of bootstrap sampling to obtain subsets of the data. These subsets are then fed into one model that will comprise the final ensemble method. This is a straightforward process, given a set of observation data, *n* observations are selected at random and with replacement to form the subsample. This subsample is what is then fed into the machine learning algorithm of choice to train the model.

Aggregation

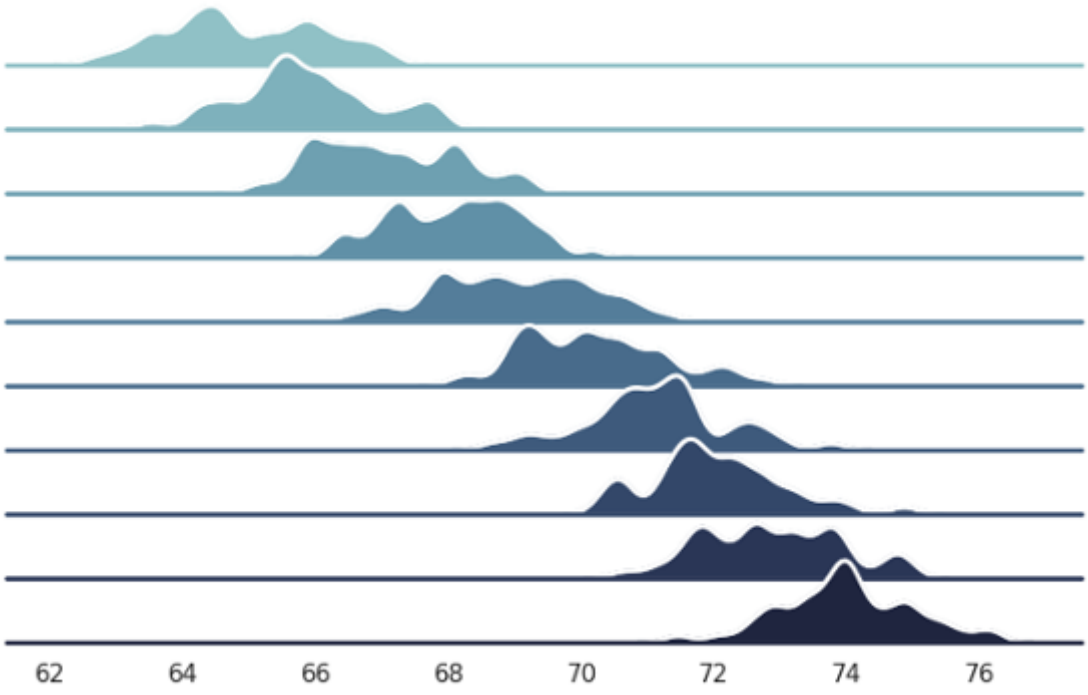
After all of the models have been built, their outputs must be aggregated into a single coherent prediction for the larger model. In the case of a classifier model, this is usually just a winners take all strategy—whichever category receives the most votes is the final outcome predicted. In the case of a regression problem, a simple average of predicted outcome values is used.

Feature Bagging

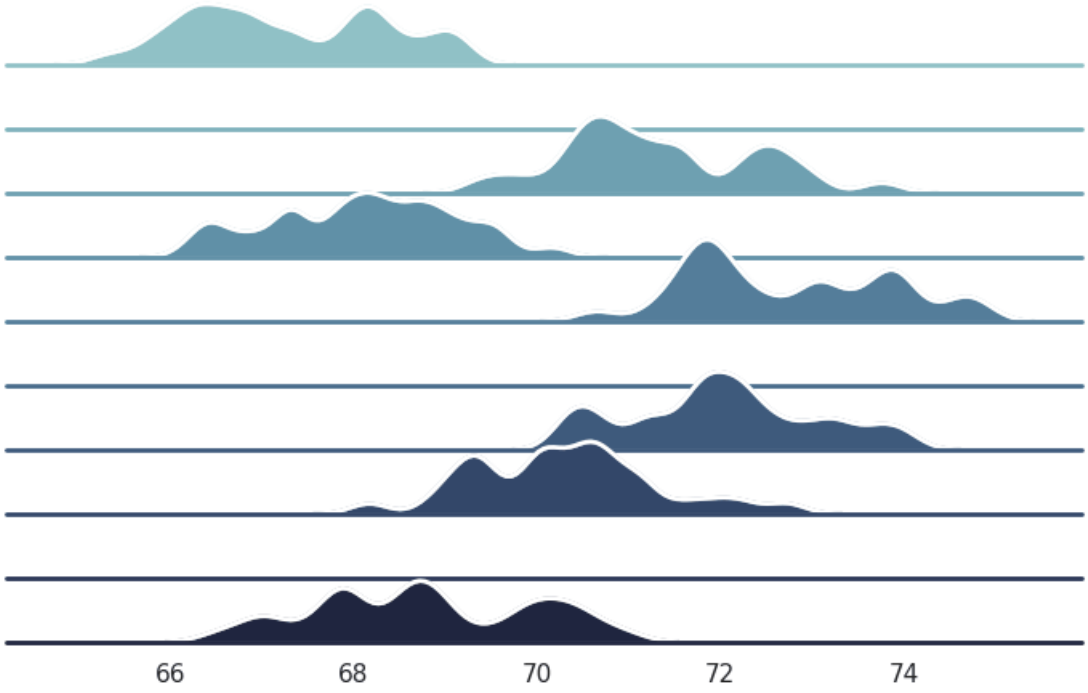
Feature bagging (or the random subspace method) is a type of ensemble method that is applied to the features (columns) of a dataset instead of to the observations (rows). It is used as a method of reducing the correlation between features by training base predictors on random subsets of features instead of the complete feature space each time.

How It Looks

A complete set of observations might be graphed as a joy plot and look like this:



The bagging method might then sample this observation repeatedly, creating a series of observations that look like these:



Each of the above distributions would then be fed into a distinct model for training.

The code to generate these plots is available here.

The Random Forest

Based on what was previously covered in decision trees and ensemble methods, it should come as little surprise as to where the random forest gets its name or how they're constructed at a high-level, but let's go over it anyways.

A random forest is comprised of a set of decision trees, each of which is trained on a random subset of the training data. These trees predictions can then be aggregated to provide a single prediction from a series of predictions.

Building A Random Forest

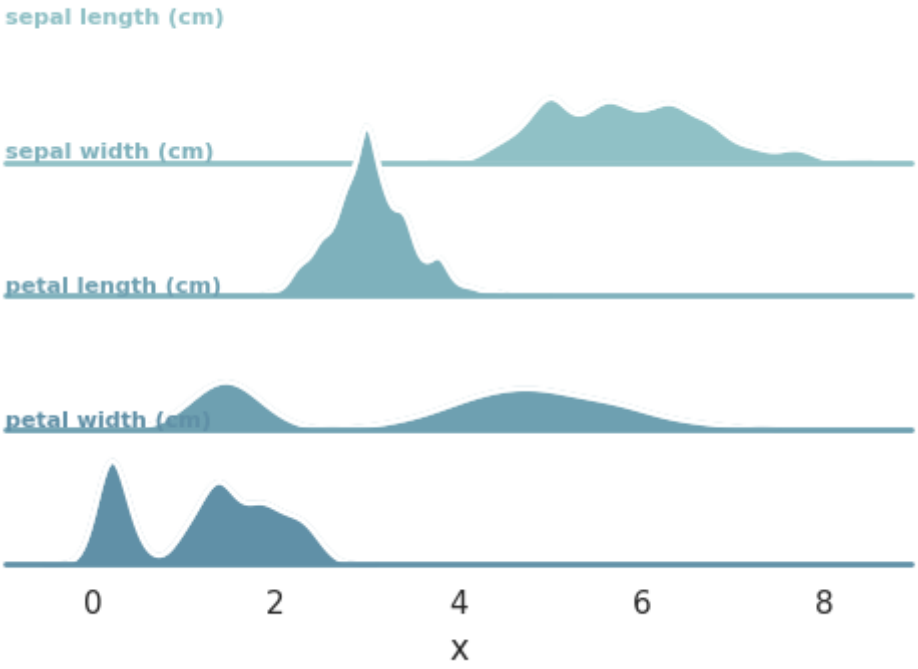
A random forest is built using the following procedure:

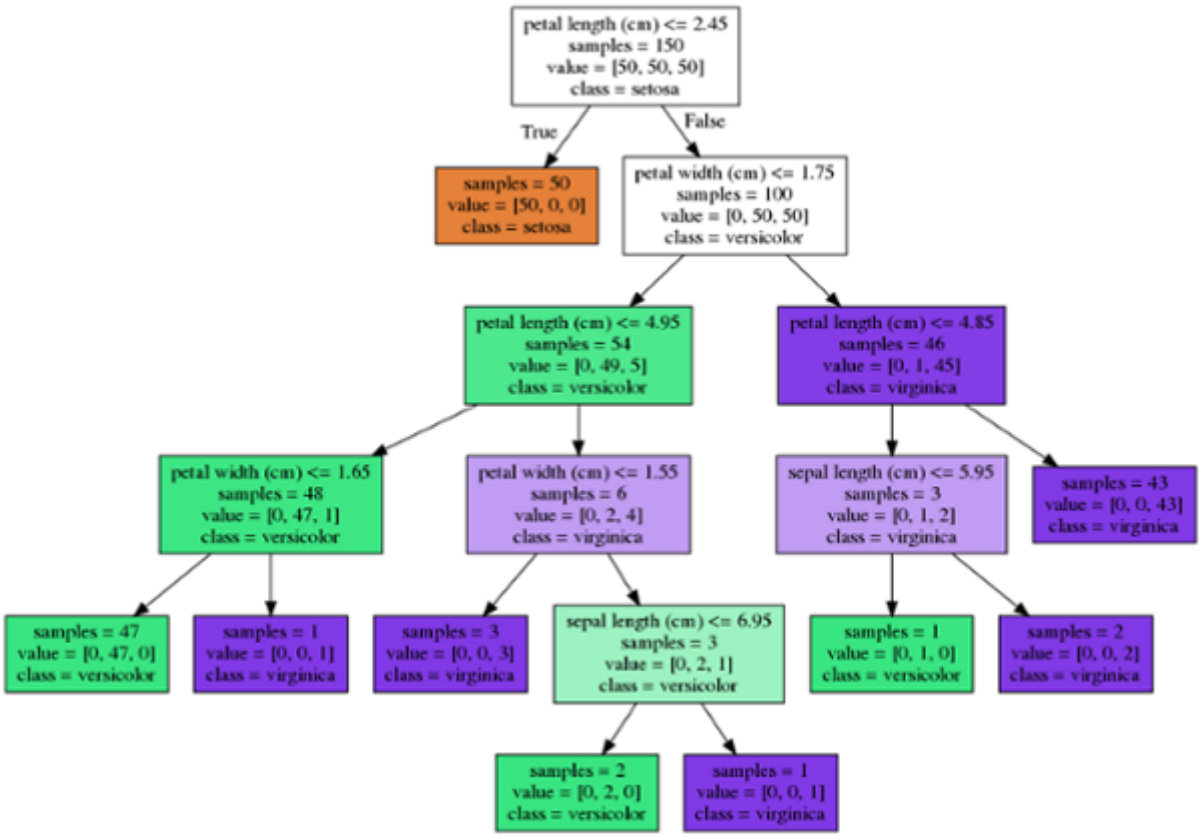
- Choose the number of trees you'd like in your forest (M)
- Choose the number of samples you'd like for each tree (n)
- Choose the number of features you'd like in each tree (f)
- For each tree in M:
    - Select n samples with replacement from all observations
    - Select f features at random
    - Train a decision tree using the data set of n samples with f features
    - Save the decision tree

How It Looks

First, lets remind ourselves what our data looks like as one entity:

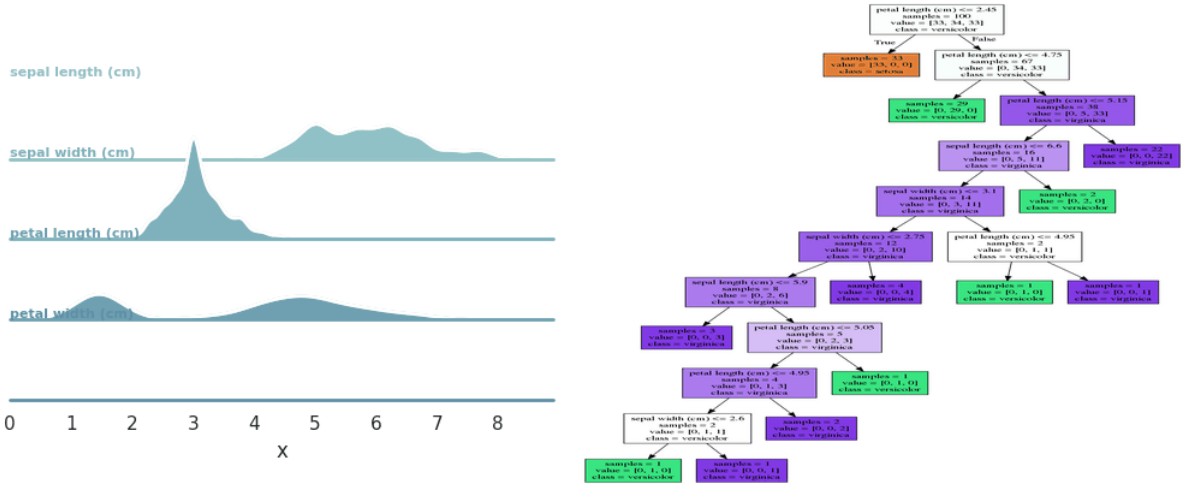
Full Data





And now once we've applied the bagging methods to this dataset:

Subsamples of Observations



As you can see, the bootstrapping and feature bagging process produces wildly different decision trees than just the single decision tree applied to all of the data.

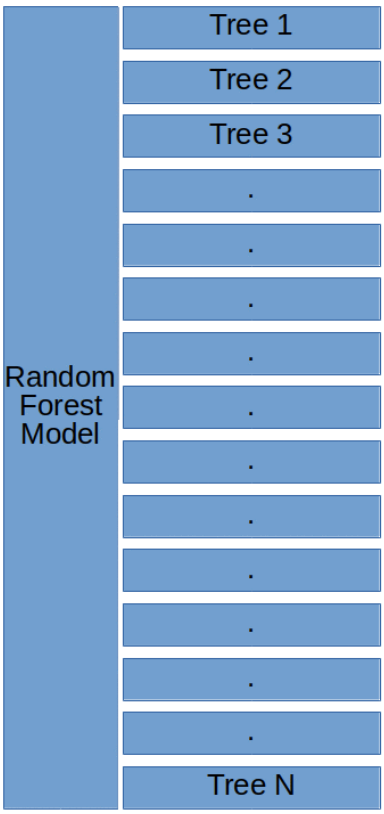
These multiple classifiers give us a number of things:

- A set of models that were trained without some features, meaning that in aggregate they're able to make predictions even with missing data.
- A set of models that viewed different subsets of data, meaning that they've all gotten slightly different ideas of how to make decisions based on different ideas of what the population looks like. This means that in aggregate they're able to make predictions even when the training data doesn't look exactly like what we're trying to predict.

How A Random Forest Makes A Prediction

- Given an observation (o).
- For each tree (t) in the model:
    - predict the outcome (p) using t applied to o
    - store p in list P

- If the model is a classifier:
    - return max_count(p)
- If the model is a regressor:
    - return avg(p)



Training a Random Forest

While you might be able to implement your own random forest from the ground up after going through this material, we will instead be using the scikit-learn implementation to go over how to train, make a prediction, and compare our random forest to a decision tree.

Note: The iris dataset doesn't have enough variance to make a random forest a better model than just a simple decision tree so for this example we'll be using the breast cancer dataset instead.

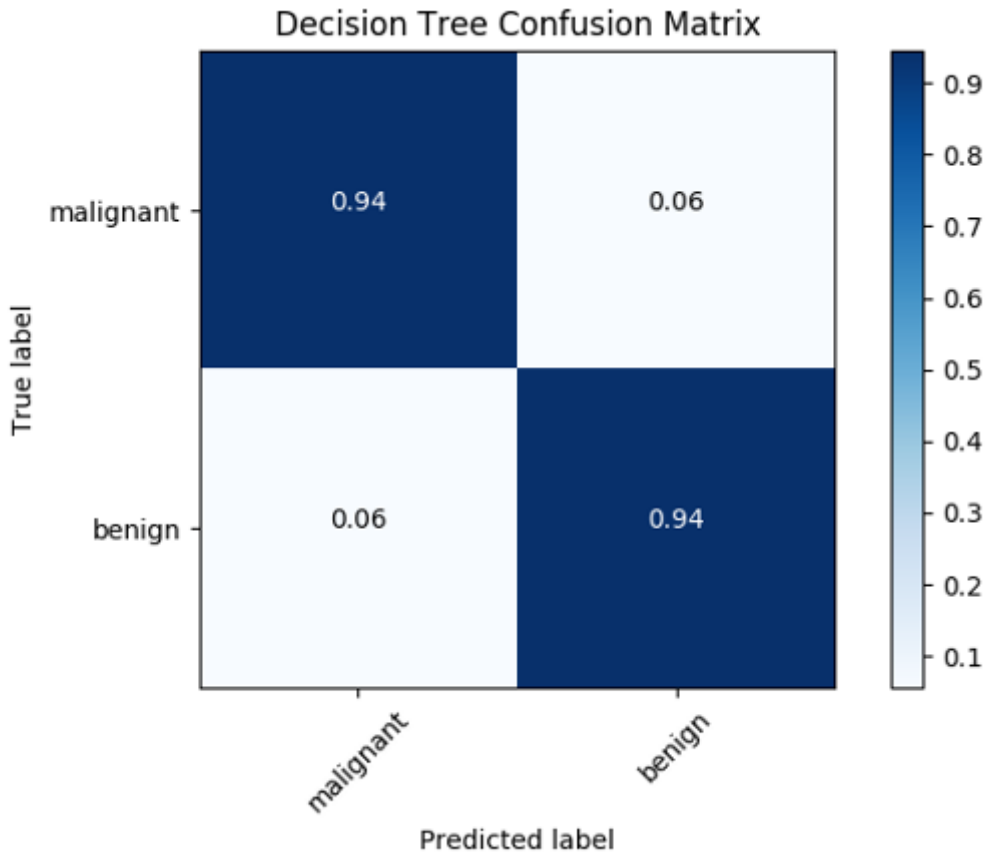
```
from sklearn.tree     import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix


bc = load_breast_cancer()
X = bc.data
y = bc.target

# Create our test/train split
X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=42)


## build our models
decision_tree = DecisionTreeClassifier()
random_forest = RandomForestClassifier(n_estimators=100)

## Train the classifiers
decision_tree.fit(X_train, y_train)
random_forest.fit(X_train, y_train)

# Create Predictions
dt_pred = decision_tree.predict(X_test)
rf_pred = random_forest.predict(X_test)

# Check the performance of each model
print('Decision Tree Model')
print(classification_report(y_test, dt_pred, target_names=bc.target_names))

print('Random Forest Model')
print(classification_report(y_test, rf_pred, target_names=bc.target_names))

#Graph our confusion matrix
dt_cm = confusion_matrix(y_test, dt_pred)
rf_cm = confusion_matrix(y_test, rf_pred)
```

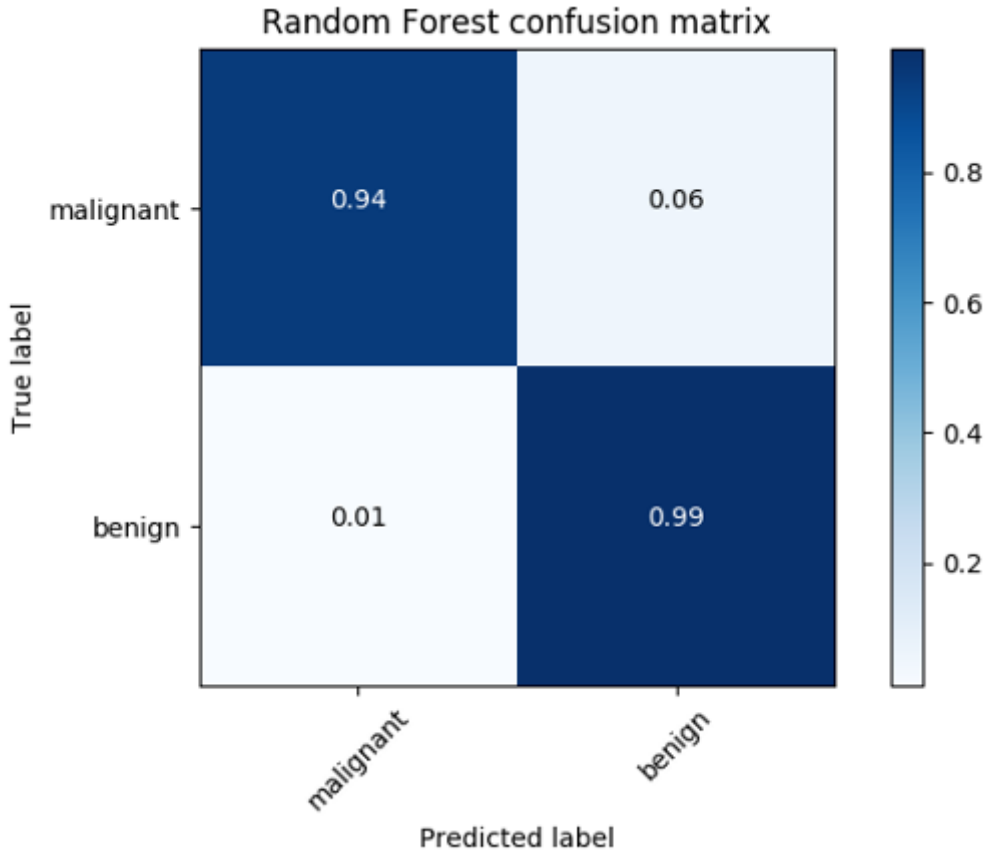
```
Decision Tree Model
             precision    recall  f1-score   support
          0       0.91      0.94      0.93        54
          1       0.97      0.94      0.95        89
avg / total       0.94      0.94      0.94       143
Random Forest Model
             precision    recall  f1-score   support
```

0	0.96	0.94	0.95	54
1	0.97	0.98	0.97	89
avg / total	0.97	0.97	0.96	143

It appears that based on common metrics of classification model performance, the random forest outperforms the decision tree.

Let's see what the performance increase actually looks like (code here):





As you can see, we're able to increase the number of correctly predicted benign tumors and decrease the number of benign tumors that are predicted as malignant. By using a random forest, we can more accurately predict the state of a tumor, potentially decreasing the amount of unneeded procedures performed on patients and decreasing patient stress about their diagnosis.

At this point, you'd usually start investigating hyperparameter tuning. This is a crucial part of the modeling process in order to ensure that your model is optimal. However, it is outside the scope of this article.

Feature Importance

Feature importance is used to describe how much impact a feature (column) has on how the model makes decisions. They're used in order to help guide future work by concentrating on things we are certain will have impact and perhaps ignoring things that don't, for simplifying models and preventing overfitting by removing columns that aren't impactful enough to be

generalized, and for helping explain why a model is making the decisions it does.

There are multiple methods for determining feature importance from a model. We'll be covering how its calculated in the scikit-learn implementation. This method is popular because it's cheap to compute, does a reasonably good job of determining importance, and is already implemented.

Unsurprisingly, in order to calculate the feature importance of the forest, we need to calculate the feature importance of the individual trees and then find a way to combine them.

Gini Impurity

Gini impurity is a measure of the chance that a new observation when randomly classified would be incorrect. It is bounded between 0 and 1 (0 being impossible to be wrong and 1 being guaranteed to be wrong).

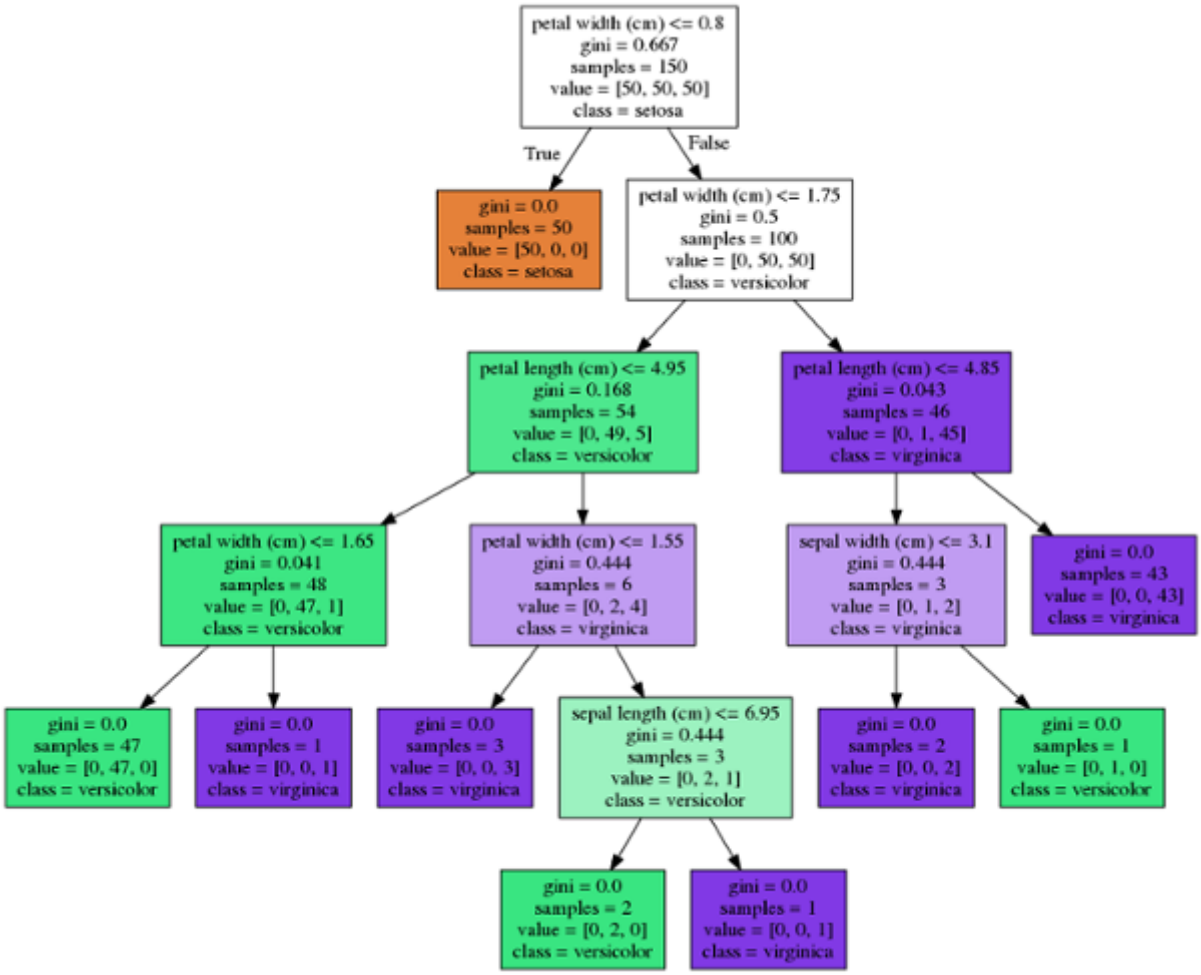
Gini impurity of a node is calculated with the following equation :

$$G(k) = \sum_{i=1}^{n} P(i) \times (1 - P(i))$$

where i is a predicted category and P(i) is the probability of a record being assigned to class i at random.

Example Calculation

Based on our first decision tree :



The gini impurity for the top node is :

50/150 * (1 - 50/150) + 50/150 * (1 - 50/150) + 50/150 * (1 - 50/150) = .667

What would the Gini impurity of next node to the left be?

Solution

50/50 * 1 - (50/50)

+ 0/50 * 1 - (0/ 50)

+ 0/50 * 1 - (0/ 50)

= 0

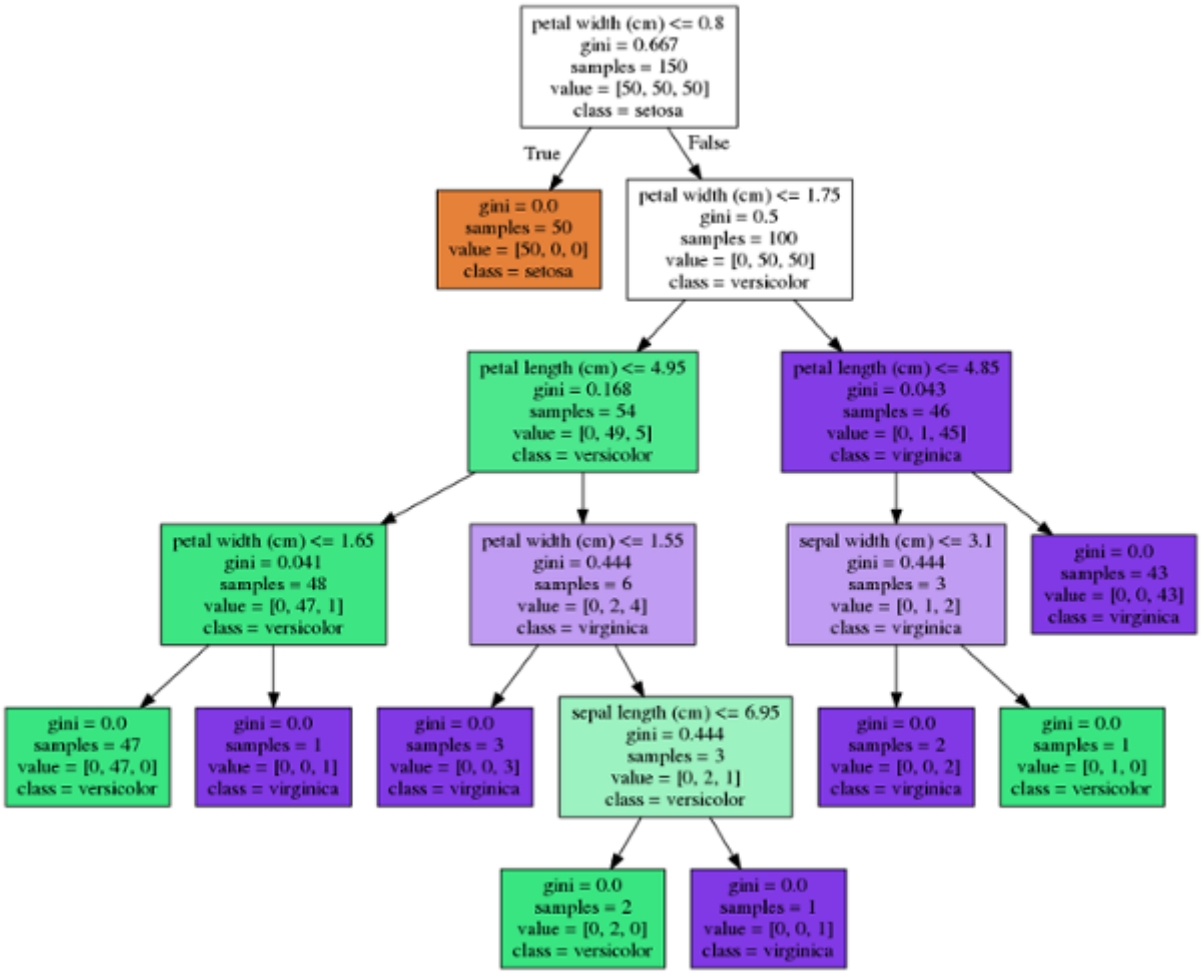
Gini Importance

The Gini importance is the total reduction of the Gini Impurity that comes from a feature. It is calculated as the weighted sum of the difference in Gini Impurity between a node and its antecedents. It is calculated using the below equation:

$$G_{importance}(k) = \sum_{i=1}^{n}(N \times G_{parent}) - (N_{child1} \times G_{child1}) - (N_{child2} \times G_{child2})$$

Example Calculation

Based on our first decision tree:



The Gini importance of sepal width is :

3×0.44−(0+0)

What would the importance of petal length be?

Solution

54 * 0.168 - (48 * 0.041 + 6 * 0.444)

+ 46 * 0.043 - (0 + 3 * 0.444)

+ 3 * 0.444 - (0 + 0)

= 6.418

Once you have the Gini importance of each feature, you simply divide by the sum of each importance to get the normalized feature importance for the model.

What's the normalized feature importances from this model?

$$G_{importance}(sepal\ length) = 0$$

$$G_{importance}(sepal\ width) = 1.332$$

$$G_{importance}(petal\ length) = 6.418$$

$$G_{importance}(petal\ width) = 92.30$$

Solution

Sum of Gini Importances = 100.05

sepal length = 0 / 100.05 = 0

sepal width = 1.332 / 100.05 = 0.0133

petal length = 6.418 / 100.05 = 0.064

petal width = 92.30 / 100.05 = 0.922

Expanding To The Random Forest

Now that we can calculate feature importance for the weak learners, expanding it to the ensembled model is as simple as calculating the average importance for a feature from the trees as the importance of the random forest.

Getting Feature Importance Via Sklearn

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris.data
y = iris.target

# Create our test/train split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# build our model
random_forest = RandomForestClassifier(n_estimators=100)

# Train the classifier
random_forest.fit(X_train, y_train)

# Get our features and weights
feature_list = sorted(zip(map(lambda x: round(x, 2), random_forest.feature_import
```

```
                reverse=True)

    # Print them out
    print('feature\t\timportance')
    print("\n".join(['{}\t\t{}'.format(f,i) for i,f in feature_list]))
    print('total_importance\t\t',  sum([i for i,f in feature_list]))
```

```
feature     importance
petal length (cm)       0.47
petal width (cm)        0.39
sepal length (cm)       0.1
sepal width (cm)        0.04
total_importance         1.0
```

Based on these weights, it's pretty clear that petal shape plays a big role in determining iris species. This might allow us to make recommendations about how we go about collecting data in the future (or not collecting data) or might give us some ideas about engineering new features around petal morphology to make our model better in the future.

Author **Dylan Storey**

Dylan Storey is a former researcher in microbial food safety, plant pathology, and genomics with a PhD in Life Sciences. He is currently working as a Data Scientist for a Fortune 50 company.

Enjoyed this post? Don't forget to share.

Related Content

LEARN DATA SCIENCE

Character-Based Neural Networks for NLP in the Real World

LEARN MORE ›

LEARN DATA SCIENCE

How Deep Neural Nets Really Learn

LEARN MORE ›

LEARN DATA SCIENCE

Supervised Learning With

Python

LEARN MORE >

SUBSCRIBE TO OUR NEWSLETTER →

Technology Solutions Resources Tools Company

Cookie Preferences