

Welcome to the module 3a coding part: Random Forest!

This notebook was originally created by Vaisakh Kusabhadran, Amisha Dhawan, Yuomi Zavaleta (all SFSU students) and Pleuni Pennings (SFSU bio professor). In 2023 it was edited by Lucy Moctezuma and Lorena Benitez-Rivera.

✓ OBJECTIVE OF THIS NOTEBOOK:

In this notebook you will make a random forest. A random forest is a collection of random-ish decision trees. This model essentially combines the output of multiple decision trees to reach a particular prediction. Most jargon in Machine Learning is just ugly, imo. But Random Forest sounds nice to me, a little poetic. If I were a singer-songwriter, I could imagine titling my album after random forests.



In this Notebook we will revisit the Cleveland Heart Disease Dataset, this time we will use a Random Forest to make our predictions rather than just using a single Decision Tree. Below is a summary of the variables that this dataset contains.

- #3 Age: age in years
- #4 Sex: sex (1 = male; 0 = female)
- #9 Chest_pain_type
 - Value 1: typical angina
 - Value 2: atypical angina
 - Value 3: non-anginal pain
 - Value 4: asymptomatic
- #10 At_rest_bp: resting blood pressure (in mm Hg on admission to the hospital)
- #12 Cholesterol: serum cholesterol in mg/dl
- #16 Fast_blood_sug: (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
- #19 Rest_ecg: resting electrocardiographic results
 - Value 0: normal
 - Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)
 - Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria
- #32 Maxhr: thalach: maximum heart rate achieved
- #38 Exer_angina: exang: exercise induced angina (1 = yes; 0 = no)
- #40 Oldpeak: ST depression induced by exercise relative to rest
- #41 Slope: the slope of the peak exercise ST segment
 - Value 1: upsloping
 - Value 2: flat
 - Value 3: downsloping
- #44 Ca: number of major vessels (0-3) colored by flourosopy
- #51 Thal: Thallium or stress test 3 = normal; 6 = fixed defect; 7 = reversable defect. See this [website](#) for more info on the thallium or stress test.
- #58 Diag: num: diagnosis of heart disease (angiographic disease status)
 - Value 0: no vessel with 50% diameter narrowing
 - Value 1: one vessel with 50% diameter narrowing

▽ Preparing libraries and data and dealing with missing data

Importing all the necessary libraries needed for processing


```
# Importing packages that deal with Data manipulation and visualization
import pandas as pd
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt

# importing libraries for ML model
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
import graphviz
```

Reading the dataset from the github repository

```
# Loading Data from Class github
columns = ["Age", "Sex", "Chest_pain_type", "At_rest_bp", "Cholesterol", "Fast_blood_sug", "Rest_ecg", "Maxhr", "Exer_angina", "Oldpeak", "Slope", "Ca", "Thal", "Diag"]
cleveland_data = pd.read_csv('https://raw.githubusercontent.com/pleunipennings/CSC508Data/main/processed.cleveland.data.txt', header=None, names=columns)

cleveland_data.head()
```



	Age	Sex	Chest_pain_type	At_rest_bp	Cholesterol	Fast_blood_sug	Rest_ecg	Maxhr	Exer_angina	Oldpeak	Slope	Ca	Thal	Diag
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0	0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	2
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0	0
4	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0	0.0	3.0	0

Replacing '?' in the dataset with the median value for that column.

```
# dealing with missing values
cleveland_data = cleveland_data.replace('?', np.nan)
cleveland_data['Thal'] = cleveland_data['Thal'].fillna(cleveland_data['Thal'].median())
cleveland_data['Ca'] = cleveland_data['Ca'].fillna(cleveland_data['Ca'].median())

# checking number of missing values afterwards
(cleveland_data==np.nan).sum()
```



	0
Age	0
Sex	0
Chest_pain_type	0
At_rest_bp	0
Cholesterol	0
Fast_blood_sug	0
Rest_ecg	0
Maxhr	0
Exer_angina	0
Oldpeak	0
Slope	0
Ca	0
Thal	0
Diag	0

dtype: int64

➡ The dimension of the table is: (303, 14)

Converting the labels to binary values so that the model is trained just to predict the presence/absence of heart disease.

```

⇒ array([0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0,
1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0,
1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1,
1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0])

```

```
# Isolating the features Dataset
features = cleveland_data.drop(columns='Diag')
features
```

303 rows × 13 columns

3/7

```
# looking at the training data labels
print("Number of people without Heart Disease: ", np.bincount(train_labels)[0])
print("Number of people with Heart Disease: ", np.bincount(train_labels)[1])
```

```
↗ Number of people without Heart Disease: 125
   Number of people with Heart Disease: 102
```

```
# looking at the testing data labels
print("Number of people without Heart Disease: ", np.bincount(test_labels)[0])
print("Number of people with Heart Disease: ", np.bincount(test_labels)[1])
```

```
↗ Number of people without Heart Disease: 39
   Number of people with Heart Disease: 37
```

✓ Training the model using test set.

You will see here that actually training the model is super easy and fast. We just need to decide how many random trees we'll make. We use random state = 42 so that we can all get exactly the same results, but if you change the random state, you'll get slightly different results. It's called random for a reason!

```
# Create and fit Random Forest Trees
rf = RandomForestClassifier(n_estimators = 1000, max_features = 'sqrt', bootstrap = True, random_state = 42)
rf.fit(train_features, train_labels)
```

```
↗ RandomForestClassifier
   RandomForestClassifier(n_estimators=1000, random_state=42)
```

There are 1000 trees that have been created in the code above. This means that we can actually checkout a single decision tree. Since python starts counts from 0. In the code below try changing the index to any value from [0, 999] so that you can see exactly how each of the decision tree looks like!

✓ Question 1

Try using index [1] and then index [2] in the code below. You can change the index in the first argument of the function `tree.export_graphviz(rf.estimators_[index])`. Are the root nodes different for these trees? Why do you think the root nodes are different for those trees?

Question 1 answer:

- Yes, in Random Forest models, each decision tree is trained on a different random subset of the data, leading to variations in the structure of the trees, including differences in the root nodes.
- Random Forests use a technique called bagging, where each tree is trained on a random subset of the training data and selects features randomly at each split. This reduces overfitting and ensures that the trees are not all identical, resulting in different features chosen as the root nodes across different trees.

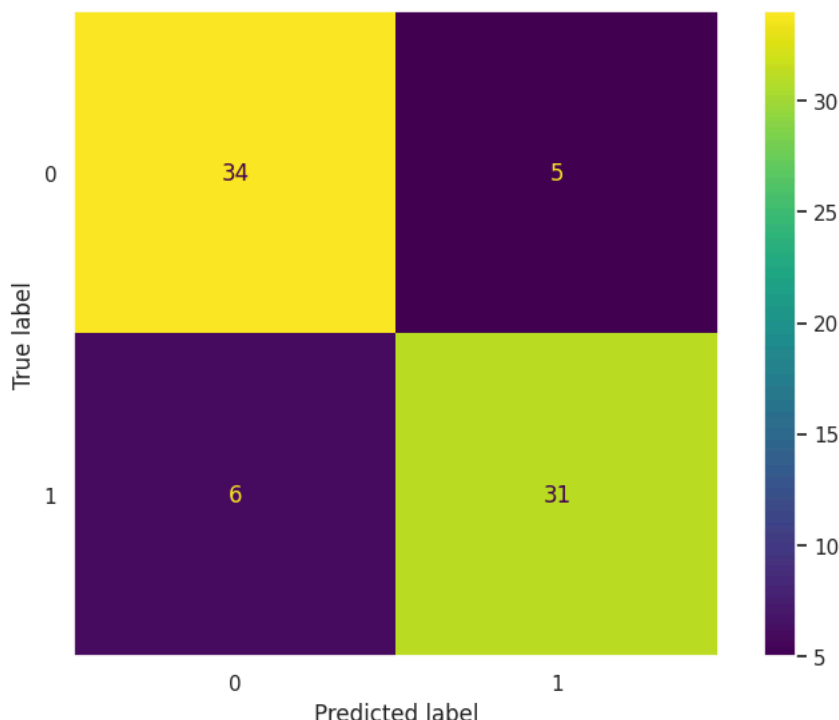
```
# looking at a single Decision tree inside the random forest
# change the index to 7
single_tree = tree.export_graphviz(rf.estimators_[7], out_file=None,
                                   feature_names=features.columns,
                                   filled=True, rounded=True,
                                   special_characters=True,
                                   max_depth= 3)

graph = graphviz.Source(single_tree)
graph.render("Classification tree")
display(graph)
```


We use the confusion matrix to visualize how well our model did.

```
# Creating a Confusion matrix for our results
sns.set(rc={'figure.figsize':(8,7,6,27)})
print(metrics.confusion_matrix(test_labels, predictions))
pltl = metrics.ConfusionMatrixDisplay.from_estimator(rf, test_features, test_labels)
plt.grid(visible=None)
```

```
[[34  5]
 [ 6 31]]
```



```
# calculating the accuracy
acc = round(100*metrics.accuracy_score(predictions, test_labels),2)
print("Accuracy:", acc, "%")
```

```
Accuracy: 85.53 %
```

Question 2

How does the accuracy of the random forest classifier compare to the decision tree? If you only made 2 or 4 or 10 random trees in your random forest, would that change your accuracy? Try it out and report here.

```
from sklearn.ensemble import RandomForestClassifier

# Random Forest with 2 trees
rf_2 = RandomForestClassifier(n_estimators=2)
rf_2.fit(train_features, train_labels)
predictions_2 = rf_2.predict(test_features)
accuracy_2 = metrics.accuracy_score(test_labels, predictions_2)
print("Random Forest with 2 trees Accuracy:", round(100*accuracy_2, 2), "%")

# Random Forest with 4 trees
rf_4 = RandomForestClassifier(n_estimators=4)
rf_4.fit(train_features, train_labels)
predictions_4 = rf_4.predict(test_features)
accuracy_4 = metrics.accuracy_score(test_labels, predictions_4)
print("Random Forest with 4 trees Accuracy:", round(100*accuracy_4, 2), "%")

# Random Forest with 10 trees
rf_10 = RandomForestClassifier(n_estimators=10)
rf_10.fit(train_features, train_labels)
predictions_10 = rf_10.predict(test_features)
accuracy_10 = metrics.accuracy_score(test_labels, predictions_10)
print("Random Forest with 10 trees Accuracy:", round(100*accuracy_10, 2), "%")
```

```
Random Forest with 2 trees Accuracy: 69.74 %
Random Forest with 4 trees Accuracy: 75.0 %
```

Random Forest with 10 trees Accuracy: 81.58 %

Question 2 answer: The accuracy will improve as you increase the number of trees, but it may plateau after a certain point, for instance 10 trees. This is because more trees can capture more patterns and reduce variance in the prediction, but after a certain number of trees, the additional benefit becomes marginal.

Feature importance

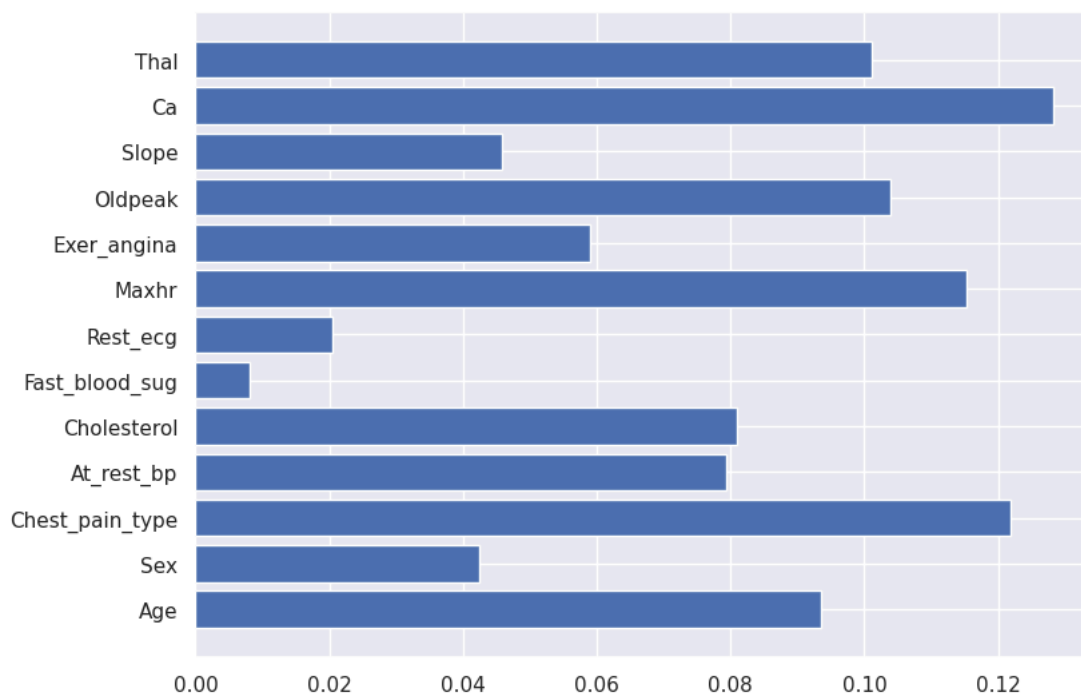
Visualizing your results is always an important part of any data science project. Now that we have a random forest based on 1000 random trees, we cannot easily visualize all the trees at once like we did for the decision tree, because it would be an overwhelming set of diagrams. But we can visualize the feature importance. I've seen this kind of plot in published articles. I like it because it helps us understand which features are most important for making predictions.

Feature importance is a measurement of how each feature decreases the amount of impurity (**Gini index**) in a node, weighted by the probability of reaching that node. The higher the value the more important the feature. This is usually calculated for each tree in the random forest and then averaged over the total number of trees. The graph below shows these averages.

[This blog has a fairly good explanation.](#)

```
importance = rf.feature_importances_  
# summarize feature importance  
print(importance)  
from matplotlib import pyplot  
pyplot.barh([x for x in range(len(importance))], importance, tick_label = columns[:-1], )  
pyplot.show()
```

```
[0.09359969 0.04243293 0.12195644 0.07935762 0.08087072 0.00814534  
0.02044649 0.11524036 0.05901268 0.10381362 0.04582646 0.1281699  
0.10112772]
```



Question 3

Do the features that were important in the decision tree we made earlier have high feature importance?

Question 3 answer: Yes, important features like Ca, Cholesterol, and Exer_angina from the decision tree remain significant, but additional features like Thal and Chest_pain_type also gain importance when averaged over multiple trees in the forest.