Comparing Decision Tree and Random Forest Classifier

Table of contents

**Author: Daniel Hassler**

## Sample Data Used in Classification

To compare a DecisionTree and a RandomForestClassifier, the first step I took was to gather some data and run some visualizations and analysis. Through Kaggle, I was able to obtain a small dataset on person features and their BMI (Body Mass Index) data. The data consists of just around 400 samples with features: gender, height, and weight, and the goal is to predict BMI.

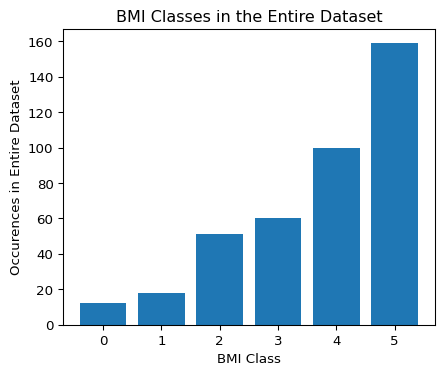
```{python}  
import numpy as np  
import sklearn  
import seaborn as sns  
import pandas as pd  
from sklearn.metrics import accuracy\_score  
from sklearn.model\_selection import train\_test\_split, cross\_val\_score, StratifiedKFold, GridSearchCV  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.tree import DecisionTreeClassifier, plot\_tree  
from sklearn import tree  
import matplotlib.pyplot as plt  
```

```{python}  
data = pd.read\_csv("./datasets/bmi\_train.csv")  
category\_mapping = {'Male': 0, 'Female': 1}  
data['Gender\_Encoded'] = data['Gender'].map(category\_mapping) # converts categorical data to numeric data.  
X = data.drop(['Gender','Index'], axis=1)  
y = data.drop(['Gender', 'Gender\_Encoded', 'Height', 'Weight'], axis=1)  
print("All X shape: ", X.shape)  
print("All y shape: ", y.shape)  
  
validation\_data = pd.read\_csv("./datasets/bmi\_validation.csv")  
print("Validation data shape: ", validation\_data.shape)  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)  
print("X\_train shape: ", X\_train.shape)  
print("y\_train shape: ", y\_train.shape)  
print("X\_test shape: ", X\_test.shape)  
print("y\_test shape: ", y\_test.shape)  
```

All X shape: (400, 3)  
All y shape: (400, 1)  
Validation data shape: (100, 3)  
X\_train shape: (320, 3)  
y\_train shape: (320, 1)  
X\_test shape: (80, 3)  
y\_test shape: (80, 1)

In the above code snippet, I first populated my data into a Pandas dataframe and then split up the data into a “training” and “testing” datasets. I decided to go with an 80/20% split between train and test (with its corresponding labels), as that seems to be the most standard approach in the industry. The significant benefit here is that I possess labeled data on both sets, a challenge in practice. This enables me to make comparisons between predictions and outcomes on my data, eliminating the need to procure any additional “test” data.

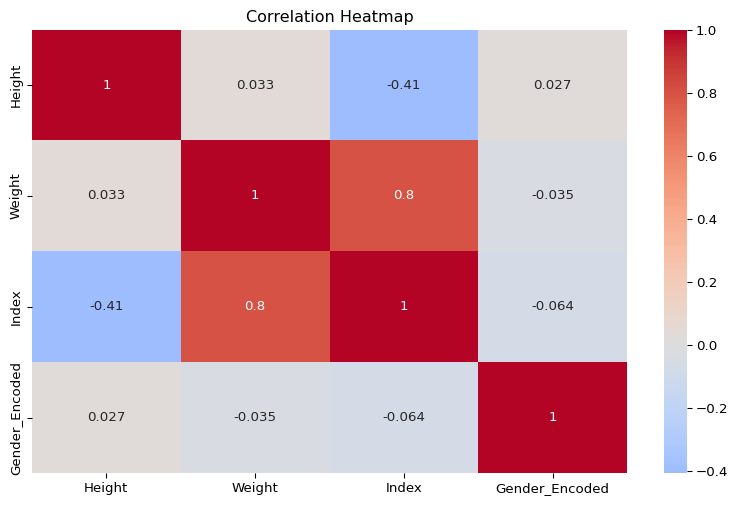
```{python}  
# Class imbalance, more obesity.  
unique\_values, counts = np.unique(y, return\_counts=True)  
plt.bar(unique\_values, counts)  
plt.title("BMI Classes in the Entire Dataset")  
plt.xlabel("BMI Class")  
plt.ylabel("Occurences in Entire Dataset")  
plt.show()  
```



The labels are all discrete and sequential, consisting of whole numbers between 0 and 5, further enforcing my intuition for using a classifier approach. A “0” in my case represents someone with an **exeptionally low** BMI, whereas a “5” depicts an **exceptionally high** BMI. Based on the distribution of the data, there appears to be a huge class imbalance, heavily favoring the amount of **exceptionally high** instances in the dataset; this was something I needed to keep in mind when building the classifiers for this dataset.

```{python}  
correlation\_matrix = data.corr()  
  
# Display a heatmap of the correlation matrix  
plt.figure(figsize=(10, 6))  
sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', center=0)  
plt.title('Correlation Heatmap')  
plt.show()  
```

C:\Users\dwh71\AppData\Local\Temp\ipykernel\_22768\1253239144.py:1: FutureWarning:  
  
The default value of numeric\_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric\_only to silence this warning.



The correlation matrix depicts the correlation between features (height, weight, gender, BMI) in the dataset. It uses the pearson’s correlation coefficient to compute this:

Based on the features presented, most are not correlated strongly, but there is a glaring strong correlation between weight and BMI.

## Building the Decision Tree Model

In order to start the model building process, I decided to tune the hyperparamters first by running a GridSearch

```{python}  
param\_grid = {  
 'max\_depth': [i for i in range(2, 10)],  
 'min\_samples\_leaf': [2 \*\* i for i in range(0, 4)],  
 'criterion': ["entropy", "gini"]  
}  
  
dt = DecisionTreeClassifier(random\_state=42)  
  
grid\_search\_dt = GridSearchCV(dt, param\_grid, cv=StratifiedKFold(n\_splits=5), scoring='accuracy')  
grid\_search\_dt.fit(X\_train, y\_train)  
best\_params\_dt = grid\_search\_dt.best\_params\_  
print("Best Hyperparameters:", best\_params\_dt)  
print("Best Score:", grid\_search\_dt.best\_score\_)  
```

Best Hyperparameters: {'criterion': 'entropy', 'max\_depth': 7, 'min\_samples\_leaf': 2}  
Best Score: 0.81875

I recognized that max\_depth was an important hyperparameter for the DecisionTree (DT), as the depth of the tree heavily influences overfitting, but other hyperparameters are important as well, such as:

* min\_samples\_leaf: the minimum amount of samples needed in a leaf node of the DT. For example, when min\_samples\_leaf is set to 10, that means a node won’t split if it has fewer than 10 samples. When this number is higher, the model can create a more generalized tree, although, when the number is smaller, it’ll create more specific splits, resulting in a more complex tree (more potential for overfitting).
* criterion: this hyperparameter chooses whether to use entropy or Gini index as a way to calculate dissimilarity in a node. I found that in most cases, entropy outpreformed the Gini index.

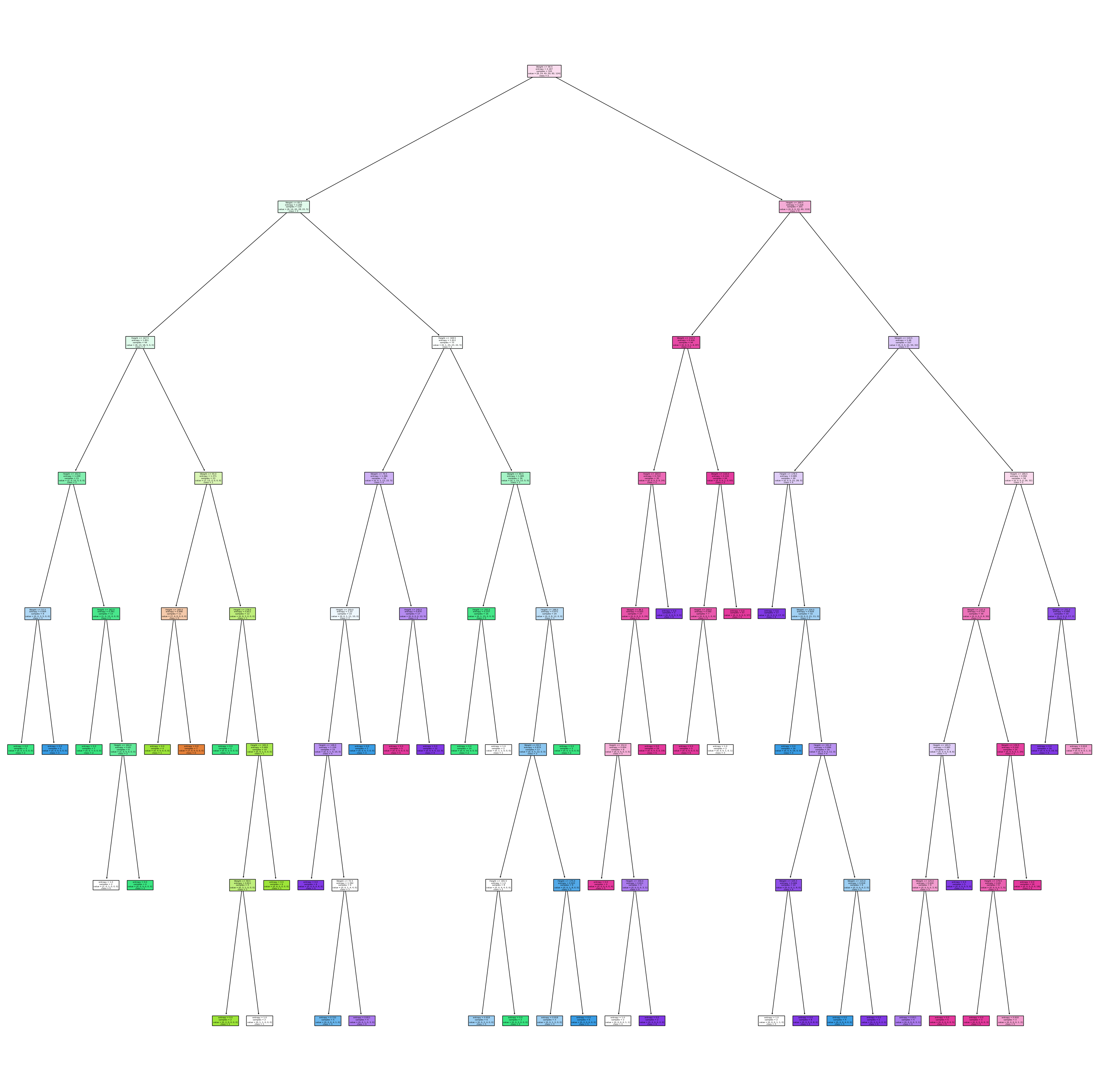
Now that I’ve determined the necessary hyperparameters for this classifier, I initialize the GridSearchCV object to analyze every combination of the above hyperparameters. Within its search, it goes through an important cross-validation step (cv) that splits the training data into multiple folds and iterates through each fold for each hyperparameter combination.

There were a few options I could’ve chose from for the cv parameter in GridSearchCV, but in order to account for class imbalance like I stated earlier, I decided to go with a StratifiedKFold cross-validator. StratifiedKFold accounts for class label imbalance by keeping an equal precentage of classes for training and testing represented in the dataset. Below is a picture representing this:

```{python}  
dt = DecisionTreeClassifier(max\_depth=best\_params\_dt["max\_depth"], min\_samples\_leaf=best\_params\_dt["min\_samples\_leaf"], criterion=best\_params\_dt["criterion"])  
dt.fit(X\_train, y\_train)  
y\_pred = dt.predict(X\_test)  
```

I then created a DecisionTreeClassifier with the ‘best’ tuned hyperparameters from the above grid search and populated the y\_pred array with the predictions from the test dataset. After that, I plotted the tree out using Sklearn’s plot\_tree method.

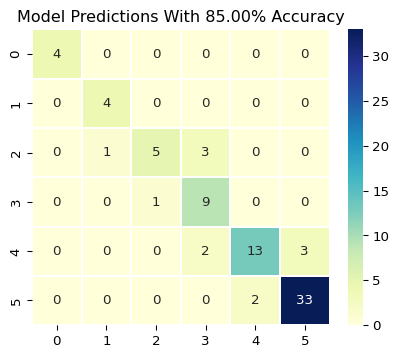
```{python}  
plt.figure(figsize=(40, 40))  
plot\_tree(dt, feature\_names=X\_train.columns.tolist(), class\_names=['0', '1', '2','3','4','5'], filled=True)  
plt.show()  
```



After plotting the tree, I created a confusion matrix, showing where my predictions fell. Currently, the model sits around 75-86% accurate due to the above hyperparameter values and the randomly generated tree with those hyperparameter values. Not bad for a small dataset with class imbalance.

```{python}  
confusion\_matrix = sklearn.metrics.confusion\_matrix(y\_test, y\_pred)  
conf\_df = pd.DataFrame(confusion\_matrix, index=[f"{i}" for i in range(6)], columns=[f"{i}" for i in range(6)])  
heatmap = sns.heatmap(conf\_df, annot=True, fmt="d", linewidths=0.35, cmap="YlGnBu")  
plt.title(f"Model Predictions With {(np.sum(confusion\_matrix.diagonal()) / y\_test.shape[0]) \* 100:.2f}% Accuracy")  
```

Text(0.5, 1.0, 'Model Predictions With 85.00% Accuracy')



## RandomForestClassifer (Ensemble approach)

As above with the DecisionTreeClassifer, I first started to implement the RandomForestClassifier by tuning the hyperparameter values. Since a RandomForest is just a collection of DecisionTrees, RandomForestClassifiers, like a DecisionTreeClassifier, have mostly the same hyperparameters, but the RandomForestClassifier has an extra one for the amount of DecisionTrees that should be included in the forest (n\_estimators).

Though this step wasn’t as necessary, since I already did the hyperparameter tuning part for the DecisionTree, but I decided to include it again for the RandomForest with the number of estimators.

It is important to note that the n\_estimators hyperparameter won’t cause the model to overfit. In fact, it actually does better at generalization when increasing the number of estimators due to the diversity of opinions the model presents for each unique DecisionTree. The only way overfitting can happen in a RandomForest depends on how the underlying DecisionTrees are set up, not the quantity of them.

```{python}  
param\_grid = {  
 'max\_depth': [i for i in range(2, 10)],  
 'min\_samples\_leaf': [2 \*\* i for i in range(0, 4)],  
 'criterion': ["entropy", "gini"]  
}  
  
rf = RandomForestClassifier(random\_state=42)  
  
print(y\_train.shape)  
grid\_search\_rf = GridSearchCV(rf, param\_grid, cv=StratifiedKFold(n\_splits=5), scoring='accuracy')  
grid\_search\_rf.fit(X\_train, y\_train.values.ravel())  
best\_params\_rf = grid\_search\_rf.best\_params\_  
print("Best Hyperparameters:", best\_params\_rf)  
print("Best Score:", grid\_search\_rf.best\_score\_)  
```

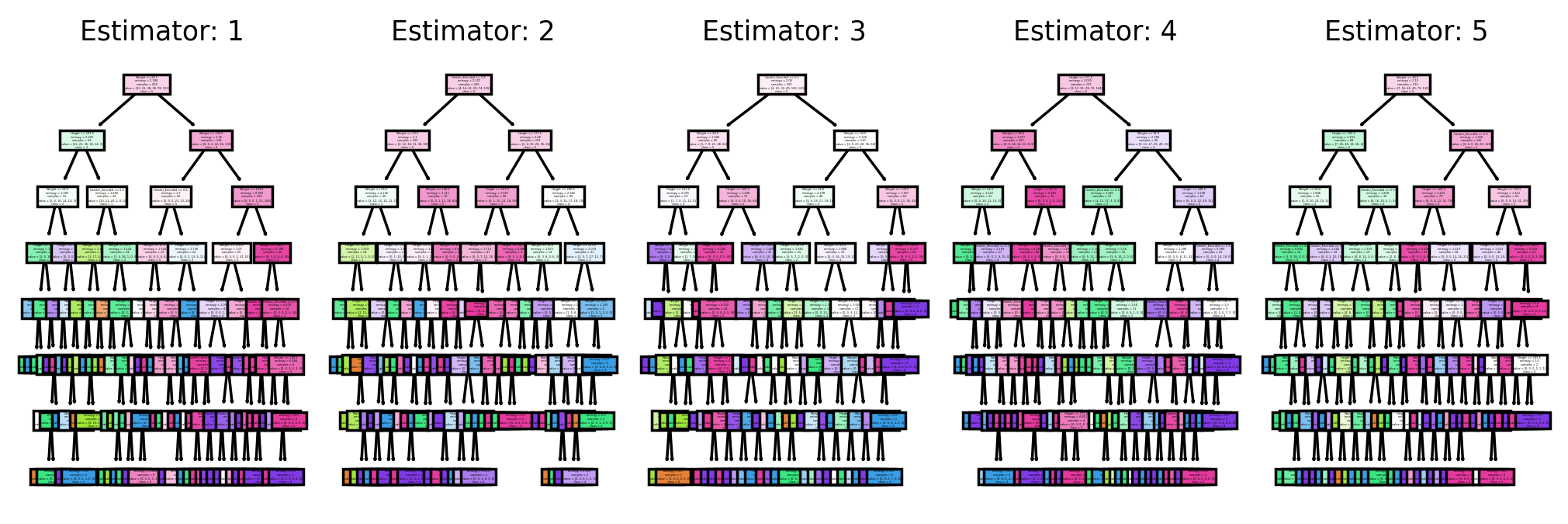
(320, 1)  
Best Hyperparameters: {'criterion': 'entropy', 'max\_depth': 7, 'min\_samples\_leaf': 1}  
Best Score: 0.853125

```{python}  
rf = RandomForestClassifier(n\_estimators=300,   
 max\_depth=best\_params\_rf["max\_depth"],   
 min\_samples\_leaf=best\_params\_rf["min\_samples\_leaf"],  
 criterion=best\_params\_rf["criterion"])  
rf.fit(X\_train, y\_train.values.ravel())  
y\_pred = rf.predict(X\_test)  
```

The above code snippet creates the RandomForestClassifier with the same hyperparameters as the DecisionTree, in addition to the number of estimators (number of decision trees in the forest), trains the classifier, then stores a prediction array.

Here is a visualizaiton of a subset of DecisionTrees in this RandomForest:

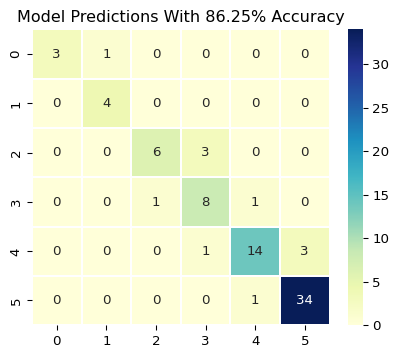
```{python}  
 fig, axes = plt.subplots(nrows = 1,ncols = 5,figsize = (10,3), dpi=250)  
 for index in range(5):  
 tree.plot\_tree(rf.estimators\_[index],  
 feature\_names = X\_train.columns.tolist(),   
 class\_names= [f"{i}" for i in range(6)],  
 filled = True,  
 ax = axes[index])  
  
 axes[index].set\_title('Estimator: ' + str(index + 1), fontsize = 10)  
 plt.show()  
```



After running the model, I checked the accuracy output of the prediction array and found that the RandomForestClassifier was able to increase the accuracy of the predictions by a considerable amount on average.

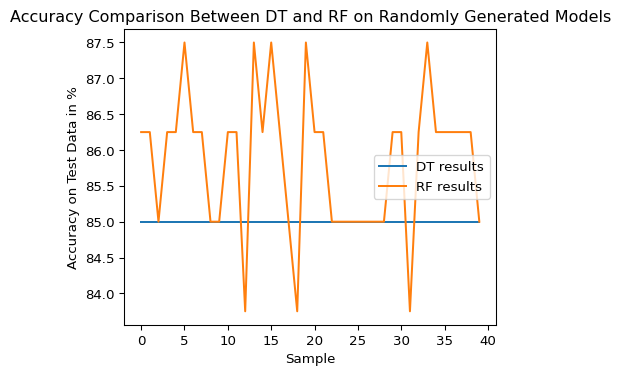
```{python}  
confusion\_matrix = sklearn.metrics.confusion\_matrix(y\_test, y\_pred)  
conf\_df = pd.DataFrame(confusion\_matrix, index=[f"{i}" for i in range(6)], columns=[f"{i}" for i in range(6)])  
heatmap = sns.heatmap(conf\_df, annot=True, fmt="d", linewidths=0.35, cmap="YlGnBu")  
plt.title(f"Model Predictions With {(np.sum(confusion\_matrix.diagonal()) / y\_test.shape[0]) \* 100:.2f}% Accuracy")  
```

Text(0.5, 1.0, 'Model Predictions With 86.25% Accuracy')



Finally, I decided to calculate the accuracy preformance on multiple samples of RandomForestClassifiers and DecisionTrees at the same time and plot them out in a line chart.

```{python}  
'''  
Plot a graph that compares the two models, randomly generated with tuned hyperparameter models  
'''  
dt\_results = []  
rf\_results = []  
n\_samples = 40  
indexes = [i for i in range(n\_samples)]  
for i in indexes:  
 dt = DecisionTreeClassifier(max\_depth=best\_params\_dt["max\_depth"],   
 min\_samples\_leaf=best\_params\_dt["min\_samples\_leaf"], criterion=best\_params\_dt["criterion"])  
 dt.fit(X\_train, y\_train)  
 y\_pred\_dt = dt.predict(X\_test)  
   
 rf = RandomForestClassifier(n\_estimators=300,   
 max\_depth=best\_params\_rf["max\_depth"],   
 min\_samples\_leaf=best\_params\_rf["min\_samples\_leaf"],  
 criterion=best\_params\_rf["criterion"])  
 rf.fit(X\_train, y\_train.values.ravel())  
 y\_pred\_rf = rf.predict(X\_test)  
   
 confusion\_matrix\_dt = sklearn.metrics.confusion\_matrix(y\_test, y\_pred\_dt)  
 confusion\_matrix\_rf = sklearn.metrics.confusion\_matrix(y\_test, y\_pred\_rf)  
   
 dt\_results.append((np.sum(confusion\_matrix\_dt.diagonal()) / y\_test.shape[0]) \* 100)  
 rf\_results.append((np.sum(confusion\_matrix\_rf.diagonal()) / y\_test.shape[0]) \* 100)  
  
plt.plot(indexes, dt\_results, label="DT results")  
plt.plot(indexes, rf\_results, label="RF results")  
plt.xlabel("Sample")  
plt.ylabel("Accuracy on Test Data in %")  
plt.title("Accuracy Comparison Between DT and RF on Randomly Generated Models")  
plt.legend()  
plt.show()  
```



## Results and Conclusions

After doing simple experimentation with these models, I have found that, on average, the RandomForestClassifier outpreforms just a singular DecisionTreeClassifier. There are several advantages to having a forest of DecisionTrees rather than a singular tree:

* More generalizability due to the ensemble approach to this problem
* Limits overfitting compared to a DT
* DT has high variance and instability, so having a forest of them results in a more collective approach.

Though there is more resource complexity with a forest, the benefits of using that over a DT is worth the tradeoff.