# **Springboard Decision Tree Specialty Coffee Case Study - Tier 3**

# The Scenario

Imagine you've just finished the Springboard Data Science Career Track course, and have been hired by a rising popular specialty coffee company - RR Diner Coffee - as a data scientist. Congratulations!

RR Diner Coffee sells two types of thing:

* specialty coffee beans, in bulk (by the kilogram only)
* coffee equipment and merchandise (grinders, brewing equipment, mugs, books, t-shirts).

RR Diner Coffee has three stores, two in Europe and one in the USA. The flagshap store is in the USA, and everything is quality assessed there, before being shipped out. Customers further away from the USA flagship store have higher shipping charges.

You've been taken on at RR Diner Coffee because the company are turning towards using data science and machine learning to systematically make decisions about which coffee farmers they should strike deals with.

RR Diner Coffee typically buys coffee from farmers, processes it on site, brings it back to the USA, roasts it, packages it, markets it, and ships it (only in bulk, and after quality assurance) to customers internationally. These customers all own coffee shops in major cities like New York, Paris, London, Hong Kong, Tokyo, and Berlin.

Now, RR Diner Coffee has a decision about whether to strike a deal with a legendary coffee farm (known as the **Hidden Farm**) in rural China: there are rumours their coffee tastes of lychee and dark chocolate, while also being as sweet as apple juice.

It's a risky decision, as the deal will be expensive, and the coffee might not be bought by customers. The stakes are high: times are tough, stocks are low, farmers are reverting to old deals with the larger enterprises and the publicity of selling *Hidden Farm* coffee could save the RR Diner Coffee business.

Your first job, then, is ***to build a decision tree to predict how many units of the Hidden Farm Chinese coffee will be purchased by RR Diner Coffee's most loyal customers.***

To this end, you and your team have conducted a survey of 710 of the most loyal RR Diner Coffee customers, collecting data on the customers':

* age
* gender
* salary
* whether they have bought at least one RR Diner Coffee product online
* their distance from the flagship store in the USA (standardized to a number between 0 and 11)
* how much they spent on RR Diner Coffee products on the week of the survey
* how much they spent on RR Diner Coffee products in the month preceding the survey
* the number of RR Diner coffee bean shipments each customer has ordered over the preceding year.

You also asked each customer participating in the survey whether they would buy the Hidden Farm coffee, and some (but not all) of the customers gave responses to that question.

You sit back and think: if more than 70% of the interviewed customers are likely to buy the Hidden Farm coffee, you will strike the deal with the local Hidden Farm farmers and sell the coffee. Otherwise, you won't strike the deal and the Hidden Farm coffee will remain in legends only. There's some doubt in your mind about whether 70% is a reasonable threshold, but it'll do for the moment.

To solve the problem, then, you will build a decision tree to implement a classification solution.

As ever, this notebook is **tiered**, meaning you can elect that tier that is right for your confidence and skill level. There are 3 tiers, with tier 1 being the easiest and tier 3 being the hardest. This is ***tier 3***, so it will be challenging.

**1. Sourcing and loading**

* Import packages
* Load data
* Explore the data

**2. Cleaning, transforming and visualizing**

* Cleaning the data
* Train/test split

**3. Modelling**

* Model 1: Entropy model - no max\_depth
* Model 2: Gini impurity model - no max\_depth
* Model 3: Entropy model - max depth 3
* Model 4: Gini impurity model - max depth 3

**4. Evaluating and concluding**

* How many customers will buy Hidden Farm coffee?
* Decision

**5. Random Forest**

* Import necessary modules
* Model
* Revise conclusion

# 0. Overview

This notebook uses decision trees to determine whether the factors of salary, gender, age, how much money the customer spent last week and during the preceding month on RR Diner Coffee products, how many kilogram coffee bags the customer bought over the last year, whether they have bought at least one RR Diner Coffee product online, and their distance from the flagship store in the USA, could predict whether customers would purchase the Hidden Farm coffee if a deal with its farmers were struck.

# 1. Sourcing and loading

## 1a. Import Packages

import pandas as pd  
import numpy as np  
from sklearn import tree, metrics  
from sklearn.model\_selection import train\_test\_split  
from sklearn.ensemble import RandomForestClassifier  
import seaborn as sns  
import matplotlib.pyplot as plt  
from io import StringIO   
from IPython.display import Image   
import pydotplus  
import pandoc  
import nbconvert as nbf

## 1b. Load data

# Read in the data to a variable called coffeeData  
coffeeData = pd.read\_csv('data/RRDinerCoffeeDatacopy.csv') # data/RRDinerCoffeeData.csv

## 1c. Explore the data

As we've seen, exploration entails doing things like checking out the **initial appearance** of the data with head(), the **dimensions** of our data with .shape, the **data types** of the variables with .info(), the **number of non-null values**, how much **memory** is being used to store the data, and finally the major summary statistcs capturing **central tendancy, dispersion and the null-excluding shape of the dataset's distribution**.

How much of this can you do yourself by this point in the course? Have a real go.

# Call head() on your data   
coffeeData.head()

Age Gender num\_coffeeBags\_per\_year spent\_week spent\_month SlrAY \  
0 36 Female 0 24 73 42789   
1 24 Male 0 44 164 74035   
2 24 Male 0 39 119 30563   
3 20 Male 0 30 107 13166   
4 24 Female 0 20 36 14244   
  
 Distance Online Decision   
0 0.003168 0 1.0   
1 0.520906 0 NaN   
2 0.916005 1 1.0   
3 0.932098 1 NaN   
4 0.965881 0 1.0

# Call .shape on your data  
coffeeData.shape

(702, 9)

# Call info() on your data  
coffeeData.info()

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 702 entries, 0 to 701  
Data columns (total 9 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Age 702 non-null int64   
 1 Gender 702 non-null object   
 2 num\_coffeeBags\_per\_year 702 non-null int64   
 3 spent\_week 702 non-null int64   
 4 spent\_month 702 non-null int64   
 5 SlrAY 702 non-null int64   
 6 Distance 702 non-null float64  
 7 Online 702 non-null int64   
 8 Decision 474 non-null float64  
dtypes: float64(2), int64(6), object(1)  
memory usage: 49.5+ KB

# Call describe() on your data to get the relevant summary statistics for your data   
coffeeData.describe()

Age num\_coffeeBags\_per\_year spent\_week spent\_month \  
count 702.000000 702.000000 702.000000 702.000000   
mean 34.243590 2.710826 32.853276 107.923077   
std 13.927945 1.593629 15.731878 55.348485   
min 16.000000 0.000000 0.000000 0.000000   
25% 23.000000 1.000000 24.250000 62.000000   
50% 28.000000 3.000000 36.000000 113.500000   
75% 46.000000 4.000000 43.000000 150.750000   
max 90.000000 5.000000 62.000000 210.000000   
  
 SlrAY Distance Online Decision   
count 702.000000 702.000000 702.000000 474.000000   
mean 43819.843305 4.559186 0.531339 0.639241   
std 26192.626943 3.116275 0.499373 0.480728   
min 1617.000000 0.003168 0.000000 0.000000   
25% 22812.250000 1.877812 0.000000 0.000000   
50% 41975.000000 4.196167 1.000000 1.000000   
75% 60223.000000 6.712022 1.000000 1.000000   
max 182058.000000 10.986203 1.000000 1.000000

# 2. Cleaning, transforming and visualizing

## 2a. Cleaning the data

Some datasets don't require any cleaning, but almost all do. This one does. We need to replace '1.0' and '0.0' in the 'Decision' column by 'YES' and 'NO' respectively, clean up the values of the 'gender' column, and change the column names to words which maximize meaning and clarity.

First, let's change the name of spent\_week, spent\_month, and SlrAY to spent\_last\_week and spent\_last\_month and salary respectively.

# Check out the names of our data's columns   
coffeeData.columns

Index(['Age', 'Gender', 'num\_coffeeBags\_per\_year', 'spent\_week', 'spent\_month',  
 'SlrAY', 'Distance', 'Online', 'Decision'],  
 dtype='object')

# Make the relevant name changes to spent\_week and spent\_per\_week.  
# Rename the columns  
coffeeData.rename(columns={'spent\_week': 'spent\_last\_week',   
 'spent\_month': 'spent\_last\_month',   
 'SlrAY': 'salary'}, inplace=True)

# Check out the column names  
coffeeData.columns

Index(['Age', 'Gender', 'num\_coffeeBags\_per\_year', 'spent\_last\_week',  
 'spent\_last\_month', 'salary', 'Distance', 'Online', 'Decision'],  
 dtype='object')

# Let's have a closer look at the gender column. Its values need cleaning.  
  
coffeeData["Gender"].describe()

count 702  
unique 9  
top Male  
freq 355  
Name: Gender, dtype: object

# See the gender column's unique values   
# Inspect unique values in the Gender column  
unique\_genders = coffeeData['Gender'].unique()  
unique\_genders

array(['Female', 'Male', 'female', 'F', 'f ', 'FEMALE', 'MALE', 'male',  
 'M'], dtype=object)

We can see a bunch of inconsistency here.

Use replace() to make the values of the gender column just Female and Male.

# Replace all alternate values for the Female entry with 'Female'  
coffeeData["Gender"] = coffeeData["Gender"].replace(["female", "f ", "FEMALE", "F"], "Female")

# Check out the unique values for the 'gender' column  
unique\_genders = coffeeData['Gender'].unique()  
unique\_genders

array(['Female', 'Male', 'MALE', 'male', 'M'], dtype=object)

# Replace all alternate values with "Male"  
coffeeData["Gender"] = coffeeData['Gender'].replace(["MALE", "male", "M"], "Male")

# Let's check the unique values of the column "gender"  
unique\_genders = coffeeData['Gender'].unique()  
unique\_genders

array(['Female', 'Male'], dtype=object)

# Check out the unique values of the column 'Decision'  
unique\_Decisions = coffeeData['Decision'].unique()  
unique\_Decisions

array([ 1., nan, 0.])

We now want to replace 1.0 and 0.0 in the Decision column by YES and NO respectively.

# Replace 1.0 and 0.0 by 'Yes' and 'No'  
coffeeData["Decision"] = coffeeData["Decision"].replace(1., "YES")  
coffeeData["Decision"] = coffeeData["Decision"].replace(0., "NO")  
coffeeData.info()

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 702 entries, 0 to 701  
Data columns (total 9 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Age 702 non-null int64   
 1 Gender 702 non-null object   
 2 num\_coffeeBags\_per\_year 702 non-null int64   
 3 spent\_last\_week 702 non-null int64   
 4 spent\_last\_month 702 non-null int64   
 5 salary 702 non-null int64   
 6 Distance 702 non-null float64  
 7 Online 702 non-null int64   
 8 Decision 474 non-null object   
dtypes: float64(1), int64(6), object(2)  
memory usage: 49.5+ KB

# Count occurrences of each gender category  
gender\_counts = coffeeData['Gender'].value\_counts()  
gender\_counts

Gender  
Male 358  
Female 344  
Name: count, dtype: int64

# Check that our replacing those values with 'YES' and 'NO' worked, with unique()  
unique\_Decisions = coffeeData['Decision'].unique()  
unique\_Decisions

array(['YES', nan, 'NO'], dtype=object)

unique\_Decisions\_counts = coffeeData['Decision'].value\_counts()  
unique\_Decisions\_counts

Decision  
YES 303  
NO 171  
Name: count, dtype: int64

## 2b. Train/test split

To execute the train/test split properly, we need to do five things:

1. Drop all rows with a null value in the Decision column, and save the result as NOPrediction: a dataset that will contain all known values for the decision
2. Visualize the data using scatter and boxplots of several variables in the y-axis and the decision on the x-axis
3. Get the subset of coffeeData with null values in the Decision column, and save that subset as Prediction
4. Divide the NOPrediction subset into X and y, and then further divide those subsets into train and test subsets for X and y respectively
5. Create dummy variables to deal with categorical inputs

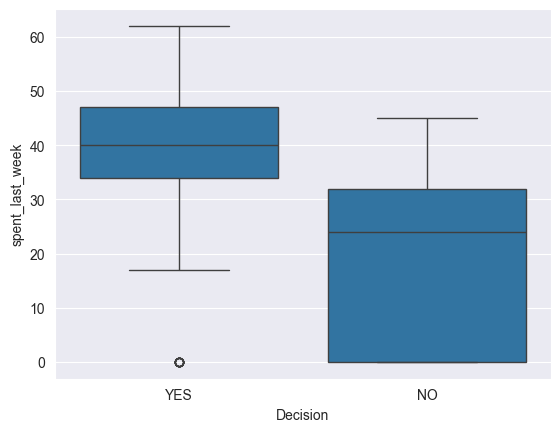
### 1. Drop all null values within the Decision column, and save the result as NoPrediction

# NoPrediction will contain all known values for the decision  
# Call dropna() on coffeeData, and store the result in a variable NOPrediction   
# Call describe() on the Decision column of NoPrediction after calling dropna() on coffeeData  
NOPrediction = coffeeData.dropna()  
NOPrediction.describe(include='all')

Age Gender num\_coffeeBags\_per\_year spent\_last\_week \  
count 474.000000 474 474.000000 474.000000   
unique NaN 2 NaN NaN   
top NaN Female NaN NaN   
freq NaN 241 NaN NaN   
mean 35.417722 NaN 2.590717 32.592827   
std 13.604135 NaN 1.585211 15.758109   
min 16.000000 NaN 0.000000 0.000000   
25% 25.000000 NaN 1.000000 24.000000   
50% 29.000000 NaN 3.000000 35.000000   
75% 49.000000 NaN 4.000000 43.000000   
max 90.000000 NaN 5.000000 62.000000   
  
 spent\_last\_month salary Distance Online Decision   
count 474.000000 474.000000 474.000000 474.000000 474   
unique NaN NaN NaN NaN 2   
top NaN NaN NaN NaN YES   
freq NaN NaN NaN NaN 303   
mean 106.727848 44731.892405 5.102898 0.512658 NaN   
std 56.101085 25567.674430 3.354891 0.500368 NaN   
min 0.000000 3150.000000 0.003168 0.000000 NaN   
25% 60.000000 27284.250000 2.077965 0.000000 NaN   
50% 113.500000 42791.500000 4.832168 1.000000 NaN   
75% 150.000000 61621.750000 7.869086 1.000000 NaN   
max 210.000000 172658.000000 10.986203 1.000000 NaN

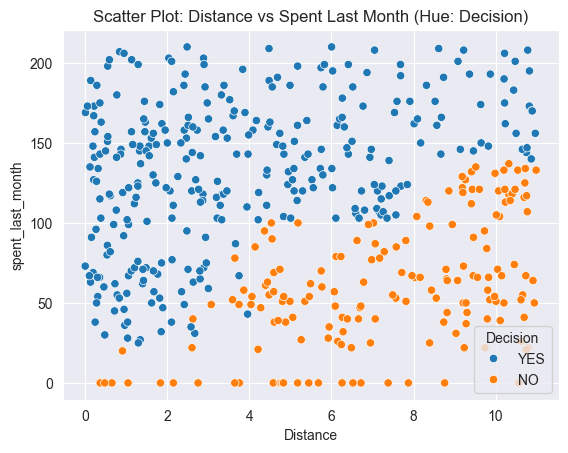
### 2. Visualize the data using scatter and boxplots of several variables in the y-axis and the decision on the x-axis

# Exploring our new NOPrediction dataset  
# Make a boxplot on NOPrediction where the x axis is Decision, and the y axis is spent\_last\_week  
sns.boxplot(y="spent\_last\_week", x= "Decision", data=NOPrediction)  
plt.show()



Can you admissibly conclude anything from this boxplot? Write your answer here: Yes, but with reservations. The persons who are more likely to buy the new coffee, are more likely to have been persons who spent last week, but the spread of how much was spent overlaps between the two groups significantly. The persons who spent last week 35 - 48 were more likely to respond with a "YES".

# Make a scatterplot on NOPrediction, where x is distance, y is spent\_last\_month and hue is Decision   
# Create scatter plot for "Distance" vs "spent\_month" with hue as "Decision"  
sns.scatterplot(y="spent\_last\_month", x= "Distance", hue = "Decision", data =NOPrediction)  
plt.title("Scatter Plot: Distance vs Spent Last Month (Hue: Decision)")  
plt.show()



Can you admissibly conclude anything from this scatterplot? Remember: we are trying to build a tree to classify unseen examples. Write your answer here: The scatterplot of Distance (x) vs Spent Last Month (y) reveals a near clear demarcation. The further the distance, and the more the Spend Last Month, the more likely the Decision to be "NO".

### 3. Get the subset of coffeeData with null values in the Decision column, and save that subset as Prediction

# Get just those rows whose value for the Decision column is null   
Prediction = coffeeData[pd.isnull(coffeeData["Decision"])]  
Prediction.head()

Age Gender num\_coffeeBags\_per\_year spent\_last\_week spent\_last\_month \  
1 24 Male 0 44 164   
3 20 Male 0 30 107   
7 24 Female 0 20 34   
11 24 Female 0 40 153   
12 21 Female 0 38 122   
  
 salary Distance Online Decision   
1 74035 0.520906 0 NaN   
3 13166 0.932098 1 NaN   
7 17425 1.193188 0 NaN   
11 84803 1.655096 1 NaN   
12 42338 1.714179 1 NaN

# Call describe() on Prediction  
Prediction.describe(include='all')

Age Gender num\_coffeeBags\_per\_year spent\_last\_week \  
count 228.000000 228 228.000000 228.000000   
unique NaN 2 NaN NaN   
top NaN Male NaN NaN   
freq NaN 125 NaN NaN   
mean 31.802632 NaN 2.960526 33.394737   
std 14.302293 NaN 1.585514 15.697930   
min 16.000000 NaN 0.000000 0.000000   
25% 22.000000 NaN 2.000000 25.750000   
50% 25.000000 NaN 3.000000 37.000000   
75% 39.000000 NaN 4.000000 44.000000   
max 67.000000 NaN 5.000000 62.000000   
  
 spent\_last\_month salary Distance Online Decision   
count 228.000000 228.000000 228.000000 228.000000 0   
unique NaN NaN NaN NaN 0   
top NaN NaN NaN NaN NaN   
freq NaN NaN NaN NaN NaN   
mean 110.407895 41923.741228 3.428836 0.570175 NaN   
std 53.786536 27406.768360 2.153102 0.496140 NaN   
min 0.000000 1617.000000 0.010048 0.000000 NaN   
25% 65.000000 15911.500000 1.699408 0.000000 NaN   
50% 113.500000 40987.500000 3.208673 1.000000 NaN   
75% 151.250000 58537.000000 5.261184 1.000000 NaN   
max 210.000000 182058.000000 10.871566 1.000000 NaN

### 4. Divide the NOPrediction subset into X and y

# Check the names of the columns of NOPrediction  
NOPrediction.columns

Index(['Age', 'Gender', 'num\_coffeeBags\_per\_year', 'spent\_last\_week',  
 'spent\_last\_month', 'salary', 'Distance', 'Online', 'Decision'],  
 dtype='object')

# Let's do our feature selection.  
# Make a variable called 'features', and a list containing the strings of every column except "Decision"  
features = ["Age", "Gender", "num\_coffeeBags\_per\_year", "spent\_last\_week", "spent\_last\_month",  
 "salary", "Distance", "Online"]  
  
# Make an explanatory variable called X, and assign it: NoPrediction[features]  
X = NOPrediction[features]   
  
# Make a dependent variable called y, and assign it: NoPrediction.Decision  
y = NOPrediction.Decision

### 5. Create dummy variables to deal with categorical inputs

One-hot encoding replaces each unique value of a given column with a new column, and puts a 1 in the new column for a given row just if its initial value for the original column matches the new column. Check out [this resource](https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f) if you haven't seen one-hot-encoding before.

**Note**: We will do this before we do our train/test split as to do it after could mean that some categories only end up in the train or test split of our data by chance and this would then lead to different shapes of data for our X\_train and X\_test which could/would cause downstream issues when fitting or predicting using a trained model.

# One-hot encode all features in X.  
X = pd.get\_dummies(X)

### 6. Further divide those subsets into train and test subsets for X and y respectively: X\_train, X\_test, y\_train, y\_test

# Call train\_test\_split on X, y. Make the test\_size = 0.25, and random\_state = 246  
X\_train, X\_test, y\_train, y\_test= train\_test\_split(X, y,   
 test\_size = 0.25,  
 random\_state = 246)

# 3. Modelling

It's useful to look at the scikit-learn documentation on decision trees <https://scikit-learn.org/stable/modules/tree.html> before launching into applying them. If you haven't seen them before, take a look at that link, in particular the section 1.10.5.

## Model 1: Entropy model - no max\_depth

We'll give you a little more guidance here, as the Python is hard to deduce, and scikitlearn takes some getting used to.

Theoretically, let's remind ourselves of what's going on with a decision tree implementing an entropy model.

Ross Quinlan's **ID3 Algorithm** was one of the first, and one of the most basic, to use entropy as a metric.

**Entropy** is a measure of how uncertain we are about which category the data-points fall into at a given point in the tree. The **Information gain** of a specific feature with a threshold (such as 'spent\_last\_month <= 138.0') is the difference in entropy that exists before and after splitting on that feature; i.e., the information we gain about the categories of the data-points by splitting on that feature and that threshold.

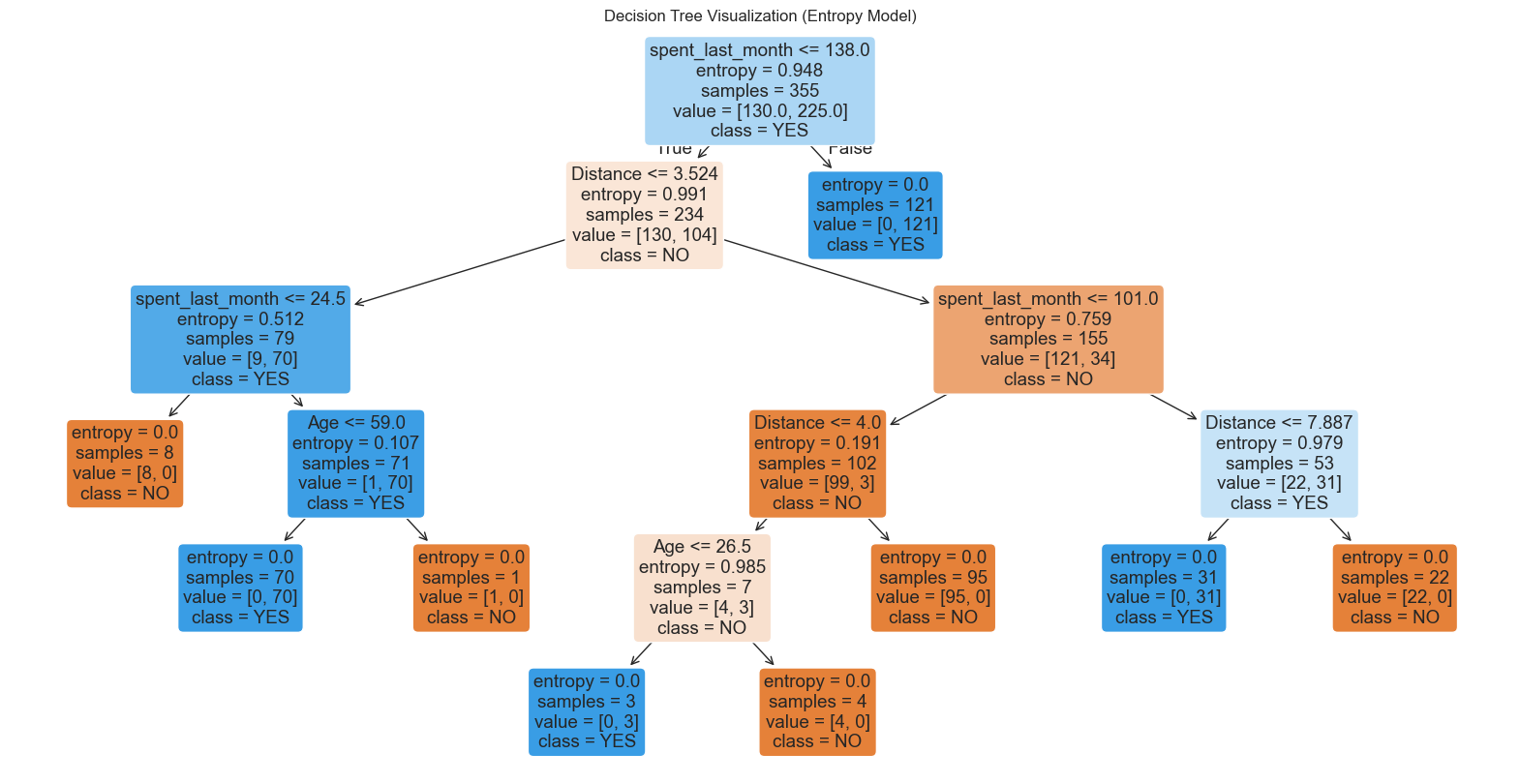
Naturally, we want to minimize entropy and maximize information gain. Quinlan's ID3 algorithm is designed to output a tree such that the features at each node, starting from the root, and going all the way down to the leaves, have maximial information gain. We want a tree whose leaves have elements that are *homogeneous*, that is, all of the same category.

The first model will be the hardest. Persevere and you'll reap the rewards: you can use almost exactly the same code for the other models.

# Declare a variable called entr\_model and use tree.DecisionTreeClassifier.   
entr\_model = tree.DecisionTreeClassifier(criterion="entropy", random\_state = 1234)  
  
# Call fit() on entr\_model  
entr\_model.fit(X\_train, y\_train)  
  
# Call predict() on entr\_model with X\_test passed to it, and assign the result to a variable y\_pred   
y\_pred = entr\_model.predict(X\_test)   
  
# Call Series on our y\_pred variable with the following: pd.Series(y\_pred)  
y\_pred = pd.Series(y\_pred)   
  
# Check out entr\_model  
entr\_model

DecisionTreeClassifier(criterion='entropy', random\_state=1234)

from sklearn.model\_selection import train\_test\_split  
from sklearn.tree import DecisionTreeClassifier, plot\_tree  
  
  
plt.figure(figsize=(20, 10))  
plot\_tree(entr\_model, filled=True, feature\_names=X\_train.columns, class\_names=entr\_model.classes\_, rounded=True)  
plt.title("Decision Tree Visualization (Entropy Model)")  
plt.show()



## Model 1: Entropy model - no max\_depth: Interpretation and evaluation

# Run this block for model evaluation metrics   
print("Model Entropy - no max depth")  
print("Accuracy:", metrics.accuracy\_score(y\_test,y\_pred))  
print("Balanced accuracy:", metrics.balanced\_accuracy\_score(y\_test,y\_pred))  
print('Precision score for "Yes"' , metrics.precision\_score(y\_test,y\_pred, pos\_label = "YES"))  
print('Precision score for "No"' , metrics.precision\_score(y\_test,y\_pred, pos\_label = "NO"))  
print('Recall score for "Yes"' , metrics.recall\_score(y\_test,y\_pred, pos\_label = "YES"))  
print('Recall score for "No"' , metrics.recall\_score(y\_test,y\_pred, pos\_label = "NO"))

Model Entropy - no max depth  
Accuracy: 0.9915966386554622  
Balanced accuracy: 0.9878048780487805  
Precision score for "Yes" 0.9873417721518988  
Precision score for "No" 1.0  
Recall score for "Yes" 1.0  
Recall score for "No" 0.975609756097561

What can you infer from these results? Write your conclusions here:

Overall Interpretation: The model is performing exceptionally well, with near-perfect accuracy, precision, and recall. Strengths: High precision means the model avoids false alarms. High recall means it does not miss many actual cases. Balanced accuracy ensures both classes are well-represented. Potential Concern: With a 100% training accuracy, there’s likely some overfitting. The model might be too complex and could struggle with new, unseen data. A model with no max depth often grows too complex. Pruning or limiting the depth may be helpful to generalize better.

## Model 2: Gini impurity model - no max\_depth

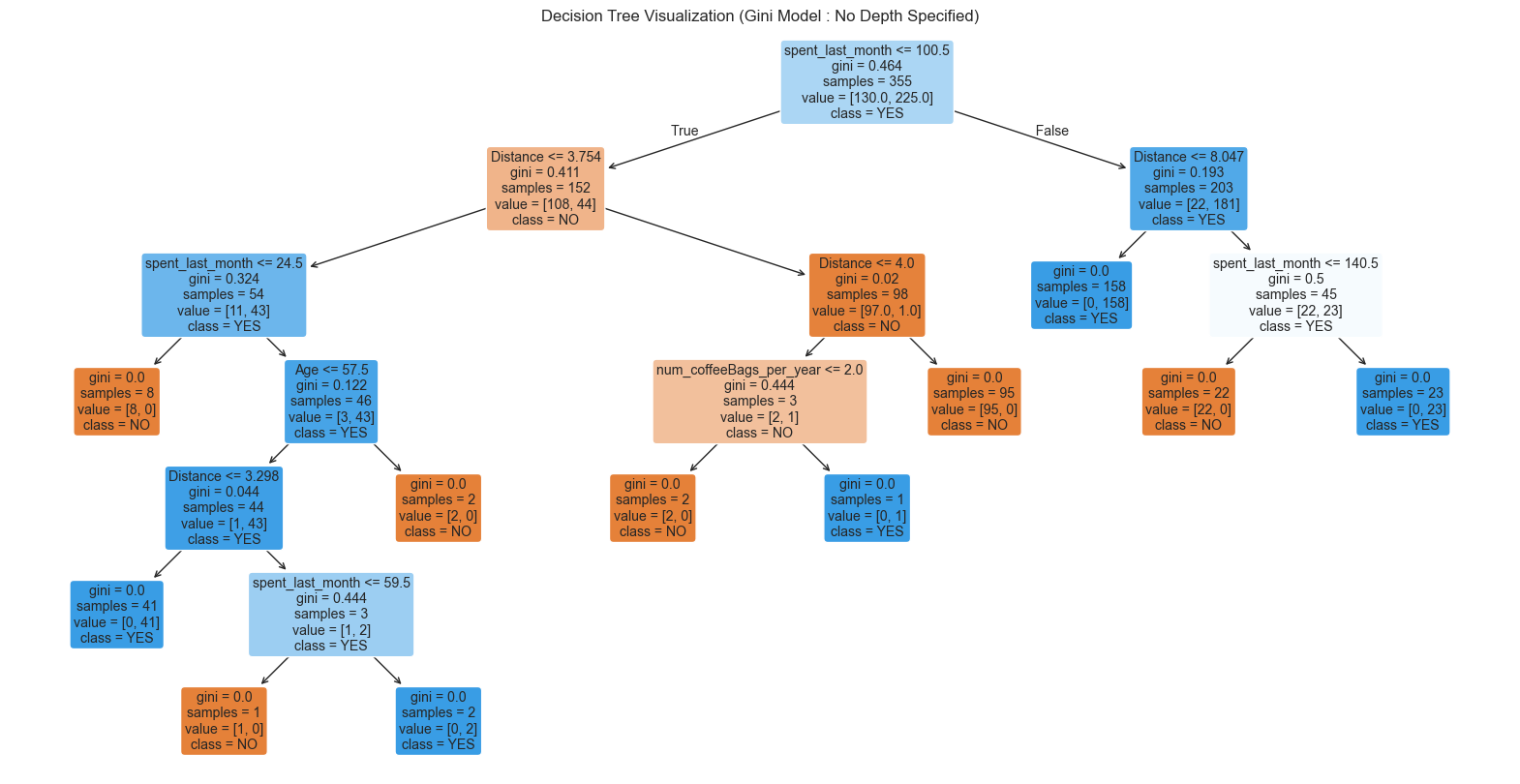
Gini impurity, like entropy, is a measure of how well a given feature (and threshold) splits the data into categories.

Their equations are similar, but Gini impurity doesn't require logorathmic functions, which can be computationally expensive.

# Make a variable called gini\_model, and assign it exactly what you assigned entr\_model with above, but with the  
# criterion changed to 'gini'  
gini\_model = DecisionTreeClassifier(criterion='gini', random\_state=1234)  
  
# Call fit() on the gini\_model as you did with the entr\_model  
gini\_model.fit(X\_train, y\_train)  
  
# Call predict() on the gini\_model as you did with the entr\_model   
y\_pred\_gini = gini\_model.predict(X\_test)  
  
# Turn y\_pred into a series, as before  
y\_pred\_gini\_series = pd.Series(y\_pred\_gini)   
  
# Check out gini\_model  
gini\_model

DecisionTreeClassifier(random\_state=1234)

# As before, but make the model name gini\_model  
 # Visualize the decision tree  
   
plt.figure(figsize=(20, 10))  
plot\_tree(gini\_model, filled=True, feature\_names=X\_train.columns, class\_names=gini\_model.classes\_, rounded=True)  
plt.title("Decision Tree Visualization (Gini Model : No Depth Specified)")  
plt.show()



# Run this block for model evaluation  
print("Model Gini impurity model")  
print("Accuracy:", metrics.accuracy\_score(y\_test,y\_pred))  
print("Balanced accuracy:", metrics.balanced\_accuracy\_score(y\_test,y\_pred))  
print('Precision score' , metrics.precision\_score(y\_test,y\_pred, pos\_label = "YES"))  
print('Recall score' , metrics.recall\_score(y\_test,y\_pred, pos\_label = "NO"))

Model Gini impurity model  
Accuracy: 0.9915966386554622  
Balanced accuracy: 0.9878048780487805  
Precision score 0.9873417721518988  
Recall score 0.975609756097561

How do the results here compare to the previous model? Write your judgements here: Previous Model:

Model Entropy - no max depth Accuracy: 0.9915966386554622 Balanced accuracy: 0.9878048780487805 Precision score for "Yes" 0.9873417721518988 Precision score for "No" 1.0 Recall score for "Yes" 1.0 Recall score for "No" 0.975609756097561

The Gini model was more accurate at .99. The balanced accuracy and Precision score and Recall score were the same essentially.

## Model 3: Entropy model - max depth 3

We're going to try to limit the depth of our decision tree, using entropy first.

As you know, we need to strike a balance with tree depth.

Insufficiently deep, and we're not giving the tree the opportunity to spot the right patterns in the training data.

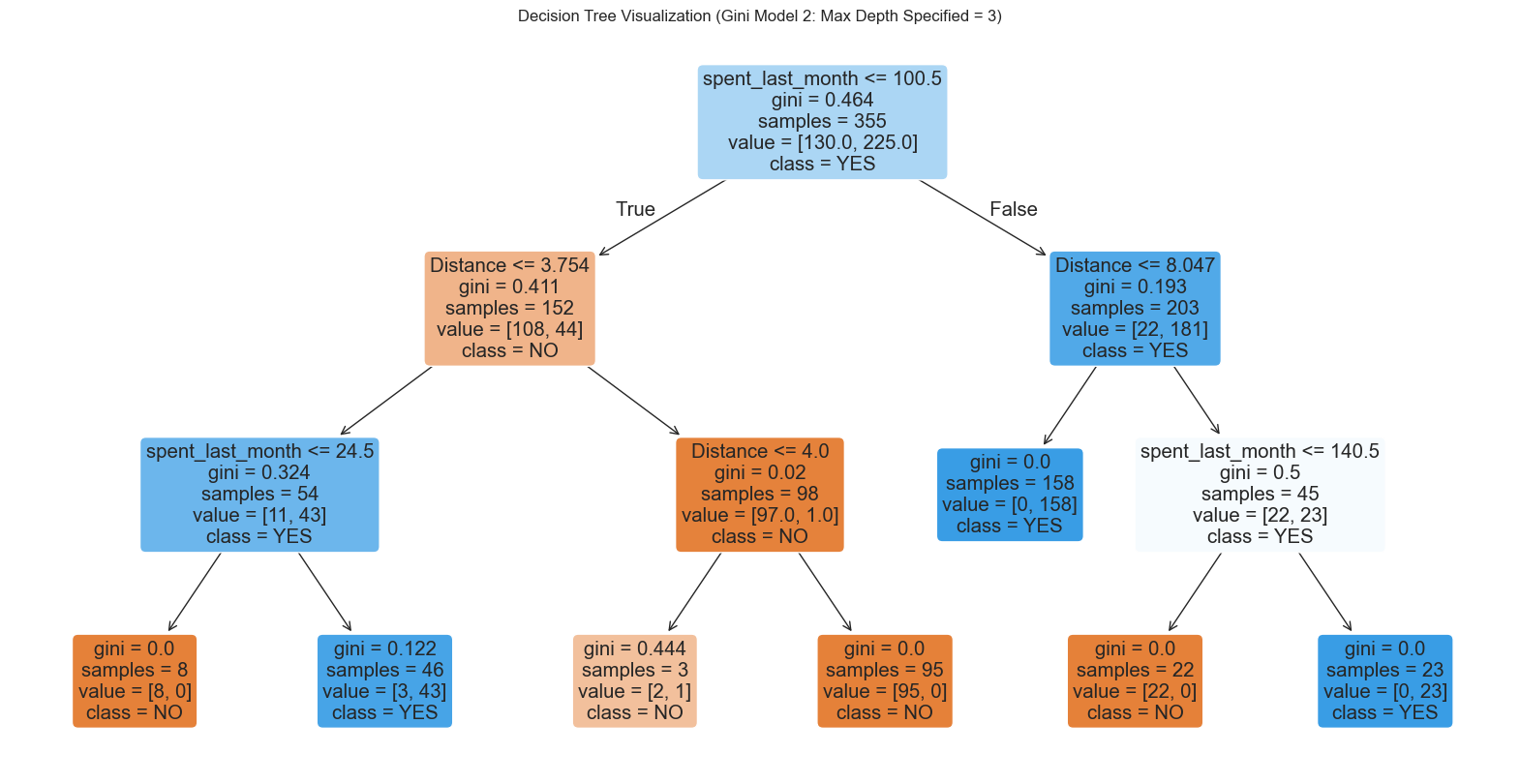
Excessively deep, and we're probably going to make a tree that overfits to the training data, at the cost of very high error on the (hitherto unseen) test data.

Sophisticated data scientists use methods like random search with cross-validation to systematically find a good depth for their tree. We'll start with picking 3, and see how that goes.

# Made a model as before, but call it entr\_model2, and make the max\_depth parameter equal to 3.   
# Execute the fitting, predicting, and Series operations as before  
# Make a variable called gini\_model2, and assign it exactly what you assigned entr\_model with above, but with the  
# criterion changed to 'gini'  
gini\_model2 = DecisionTreeClassifier(criterion='gini', random\_state=1234, max\_depth=3)  
  
# Call fit() on the gini\_model as you did with the entr\_model  
gini\_model2.fit(X\_train, y\_train)  
  
# Call predict() on the gini\_model as you did with the entr\_model   
y\_pred\_gini = gini\_model2.predict(X\_test)  
  
# Turn y\_pred into a series, as before  
y\_pred\_gini\_series = pd.Series(y\_pred\_gini)   
  
# Check out gini\_model  
gini\_model2

DecisionTreeClassifier(max\_depth=3, random\_state=1234)

# As before, we need to visualize the tree to grasp its nature  
# Visualize the decision tree  
   
plt.figure(figsize=(20, 10))  
plot\_tree(gini\_model2, filled=True, feature\_names=X\_train.columns, class\_names=gini\_model2.classes\_, rounded=True)  
plt.title("Decision Tree Visualization (Gini Model 2: Max Depth Specified = 3)")  
plt.show()



# Run this block for model evaluation   
print("Model Entropy model max depth 3")  
print("Accuracy:", metrics.accuracy\_score(y\_test,y\_pred))  
print("Balanced accuracy:", metrics.balanced\_accuracy\_score(y\_test,y\_pred))  
print('Precision score for "Yes"' , metrics.precision\_score(y\_test,y\_pred, pos\_label = "YES"))  
print('Recall score for "No"' , metrics.recall\_score(y\_test,y\_pred, pos\_label = "NO"))

Model Entropy model max depth 3  
Accuracy: 0.9915966386554622  
Balanced accuracy: 0.9878048780487805  
Precision score for "Yes" 0.9873417721518988  
Recall score for "No" 0.975609756097561

So our accuracy decreased, but is this certainly an inferior tree to the max depth original tree we did with Model 1? Write your conclusions here: The accuracy decreased slightly, but with a simpler pruned tree, which is easier to analyze. It may not generalize as well to new data.

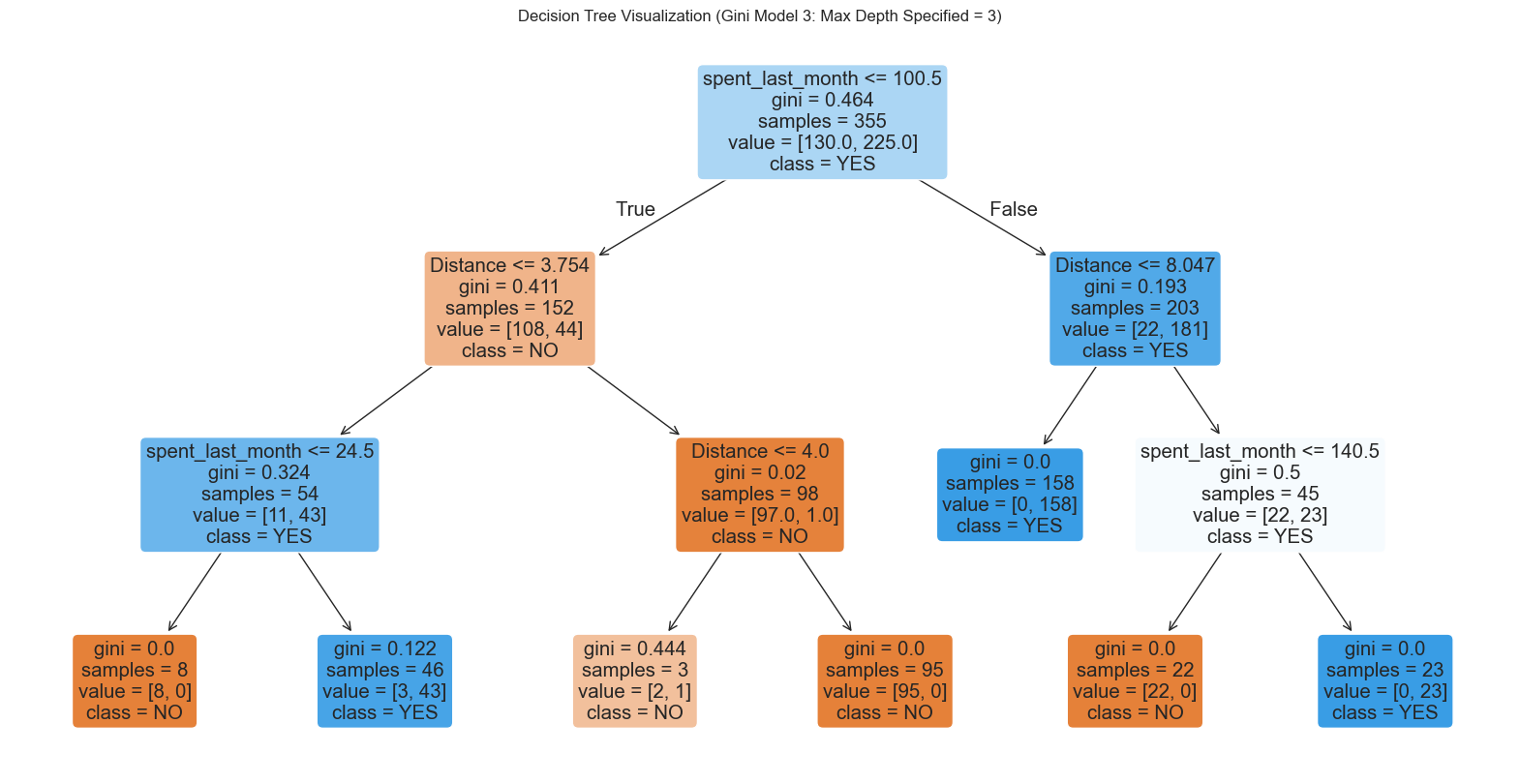
Gini Model No depth: Model Gini impurity model Accuracy: 0.9915966386554622 Balanced accuracy: 0.9878048780487805 Precision score 0.9873417721518988 Recall score 0.975609756097561

## Model 4: Gini impurity model - max depth 3

We're now going to try the same with the Gini impurity model.

# As before, make a variable, but call it gini\_model3, and ensure the max\_depth parameter is set to 3  
# Train the Gini-based Decision Tree model with max\_depth=3  
gini\_model3 = DecisionTreeClassifier(criterion='gini', random\_state=1234, max\_depth=3)  
  
# Do the fit, predict, and series transformations as before.   
gini\_model3.fit(X\_train, y\_train)  
y\_pred\_gini3 = gini\_model3.predict(X\_test)

# Visualize the decision tree  
   
plt.figure(figsize=(20, 10))  
plot\_tree(gini\_model3, filled=True, feature\_names=X\_train.columns, class\_names=gini\_model3.classes\_, rounded=True)  
plt.title("Decision Tree Visualization (Gini Model 3: Max Depth Specified = 3)")  
plt.show()



print("Gini impurity model - max depth 3")  
print("Accuracy:", metrics.accuracy\_score(y\_test,y\_pred))  
print("Balanced accuracy:", metrics.balanced\_accuracy\_score(y\_test,y\_pred))  
print('Precision score' , metrics.precision\_score(y\_test,y\_pred, pos\_label = "YES"))  
print('Recall score' , metrics.recall\_score(y\_test,y\_pred, pos\_label = "NO"))

Gini impurity model - max depth 3  
Accuracy: 0.9915966386554622  
Balanced accuracy: 0.9878048780487805  
Precision score 0.9873417721518988  
Recall score 0.975609756097561

Now this is an elegant tree. Its accuracy might not be the highest, but it's still the best model we've produced so far. Why is that? Write your answer here: It provides a simpler and more interpretable model without sacrificing much accuracy. A smaller tree is less prone to overfitting and generalizes better to new data.

# 4. Evaluating and concluding

## 4a. How many customers will buy Hidden Farm coffee?

Let's first ascertain how many loyal customers claimed, in the survey, that they will purchase the Hidden Farm coffee.

# Call value\_counts() on the 'Decision' column of the original coffeeData  
# Count the occurrences of each decision in the original dataset  
decision\_counts = coffeeData["Decision"].value\_counts()  
  
decision\_counts

Decision  
YES 303  
NO 171  
Name: count, dtype: int64

Let's now determine the number of people that, according to the model, will be willing to buy the Hidden Farm coffee.

1. First we subset the Prediction dataset into new\_X considering all the variables except Decision
2. Use that dataset to predict a new variable called potential\_buyers

# Feature selection  
# Make a variable called feature\_cols, and assign it a list containing all the column names except 'Decision'  
feature\_cols = [col for col in coffeeData.columns if col != "Decision"]  
  
# Make a variable called new\_X, and assign it the subset of Prediction, containing just the feature\_cols   
new\_X = Prediction[feature\_cols]

# Call get\_dummies() on the Pandas object pd, with new\_X plugged in, to one-hot encode all features in the training set  
new\_X = pd.get\_dummies(new\_X)  
  
# Make a variable called potential\_buyers, and assign it the result of calling predict() on a model of your choice;   
# don't forget to pass new\_X to predict()  
potential\_buyers = gini\_model2.predict(new\_X)

# Let's get the numbers of YES's and NO's in the potential buyers   
# Call unique() on np, and pass potential\_buyers and return\_counts=True   
np.unique(potential\_buyers, return\_counts=True)

(array(['NO', 'YES'], dtype=object), array([ 45, 183], dtype=int64))

The total number of potential buyers is 303 + 183 = 486

# Print the total number of surveyed people   
print("The total number of surveyed people was", coffeeData.salary.count())

The total number of surveyed people was 702

# Let's calculate the proportion of buyers  
486/702

0.6923076923076923

# Print the percentage of people who want to buy the Hidden Farm coffee, by our model   
print("Only ", round((486/702)\*100, 2), "% of people want to buy the Hidden Farm coffee." )

Only 69.23 % of people want to buy the Hidden Farm coffee.

## 4b. Decision

Remember how you thought at the start: if more than 70% of the interviewed customers are likely to buy the Hidden Farm coffee, you will strike the deal with the local Hidden Farm farmers and sell the coffee. Otherwise, you won't strike the deal and the Hidden Farm coffee will remain in legends only. Well now's crunch time. Are you going to go ahead with that idea? If so, you won't be striking the deal with the Chinese farmers.

They're called decision trees, aren't they? So where's the decision? What should you do? (Cue existential cat emoji).

Ultimately, though, we can't write an algorithm to actually *make the business decision* for us. This is because such decisions depend on our values, what risks we are willing to take, the stakes of our decisions, and how important it us for us to *know* that we will succeed. What are you going to do with the models you've made? Are you going to risk everything, strike the deal with the *Hidden Farm* farmers, and sell the coffee?

The philosopher of language Jason Stanley once wrote that the number of doubts our evidence has to rule out in order for us to know a given proposition depends on our stakes: the higher our stakes, the more doubts our evidence has to rule out, and therefore the harder it is for us to know things. We can end up paralyzed in predicaments; sometimes, we can act to better our situation only if we already know certain things, which we can only if our stakes were lower and we'd *already* bettered our situation.

Data science and machine learning can't solve such problems. But what it can do is help us make great use of our data to help *inform* our decisions.

## 5. Random Forest

You might have noticed an important fact about decision trees. Each time we run a given decision tree algorithm to make a prediction (such as whether customers will buy the Hidden Farm coffee) we will actually get a slightly different result. This might seem weird, but it has a simple explanation: machine learning algorithms are by definition ***stochastic***, in that their output is at least partly determined by randomness.

To account for this variability and ensure that we get the most accurate prediction, we might want to actually make lots of decision trees, and get a value that captures the centre or average of the outputs of those trees. Luckily, there's a method for this, known as the ***Random Forest***.

Essentially, Random Forest involves making lots of trees with similar properties, and then performing summary statistics on the outputs of those trees to reach that central value. Random forests are hugely powerful classifers, and they can improve predictive accuracy and control over-fitting.

Why not try to inform your decision with random forest? You'll need to make use of the RandomForestClassifier function within the sklearn.ensemble module, found [here](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html).

### 5a. Import necessary modules

from sklearn.ensemble import RandomForestClassifier  
from sklearn.datasets import make\_classification

### 5b. Model

You'll use your X\_train and y\_train variables just as before.

You'll then need to make a variable (call it firstRFModel) to store your new Random Forest model. You'll assign this variable the result of calling RandomForestClassifier().

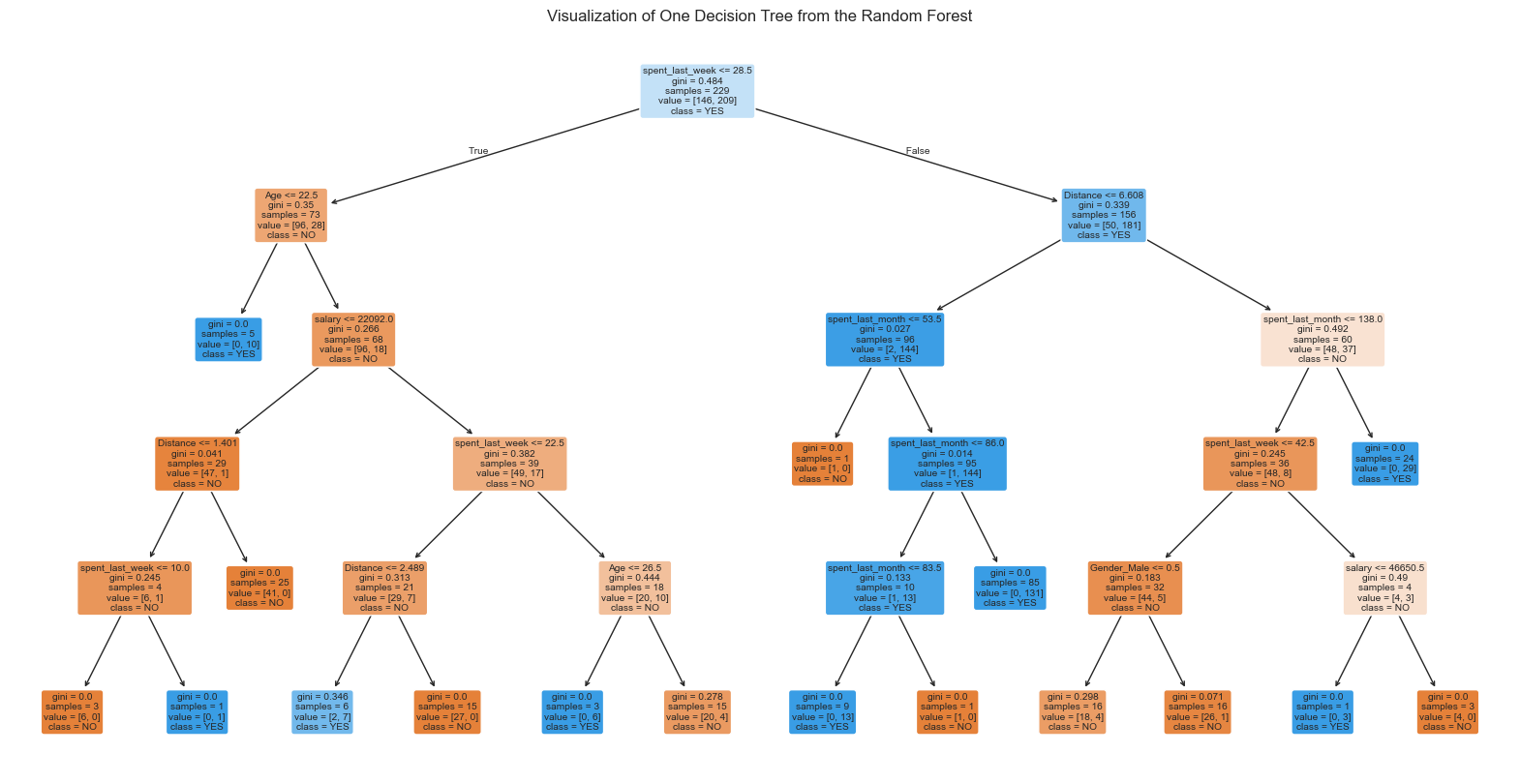
Then, just as before, you'll call fit() on that firstRFModel variable, and plug in X\_train and y\_train.

Finally, you should make a variable called y\_pred, and assign it the result of calling the predict() method on your new firstRFModel, with the X\_test data passed to it.

# Plug in appropriate max\_depth and random\_state parameters   
  
# Initialize the Random Forest model  
firstRFModel = RandomForestClassifier(random\_state=1234, max\_depth=5)  
  
# Train the model on the training data  
firstRFModel.fit(X\_train, y\_train)  
  
# Make predictions on the test set  
y\_pred = firstRFModel.predict(X\_test)  
  
# Display the trained model object  
firstRFModel

RandomForestClassifier(max\_depth=5, random\_state=1234)

# Visualize one tree from the Random Forest  
plt.figure(figsize=(20, 10))  
plot\_tree(firstRFModel.estimators\_[0], filled=True, feature\_names=X\_train.columns, class\_names=firstRFModel.classes\_, rounded=True)  
plt.title("Visualization of One Decision Tree from the Random Forest")  
plt.show()



potential\_buyers = firstRFModel.predict(new\_X)  
np.unique(potential\_buyers, return\_counts=True)

(array(['NO', 'YES'], dtype=object), array([ 41, 187], dtype=int64))

# Calculate the proportion of buyers  
187/702

0.26638176638176636

# Print the percentage of people who want to buy the Hidden Farm coffee, by our model   
print("Only ", round((187/702)\*100, 2), "% of people want to buy the Hidden Farm coffee." )

Only 26.64 % of people want to buy the Hidden Farm coffee.

### 5c. Revise conclusion

Has your conclusion changed? Or is the result of executing random forest the same as your best model reached by a single decision tree?

## Yes. Now with random forest decision tree analysis the percent of predicted buyers is much lower than the threshold of 70%, above which we might have been inclined to buy and market the new coffee. It is 26.64%.