Simeon Sukinder (sprem)

We highly recommend you work as a group!

Classmates you worked with today:

- Anthony Mele
- Ryan S
- Arjun M

Submission instruction: Please create a pdf via File -> Print (or cmd + P on mac), and upload it to Gradescope. No autograder since there are many correct ways to approach this question. You are not required to finish the entire worksheet provided that you were actively engaged during the entire class period. As long as you've collaborated, put in good effort and made reasonable progress during the lab period, you can expect to get full credit. So even if you decide to finish it at home, please submit what you have by the end of 50 minutes to make sure you get credit!

Part A: Preparing data

Our goal today is to figure out what the defining characteristics of each penguin species are such that if we encounter a new penguin in the wild, we can predict their species. Start by answering the following questions:

- Is this more of a **classification** problem or a **regression** problem? Choose one closest answer.
- What are the **predictor variables** (or features or X)?
- What are the **target variables** (or labels or y)?

If you are working on the penguin track for the final project, you are welcome to use anything from this lab for the project.

1. This is a classification problem.
2. Predictor variables include things like culmen length, flipper length, culmen depth, mass, sex, and other variables.
3. Target variable is the species of the penguin.

Run the following cell to load the penguin dataset as a `pandas` `DataFrame` called

`penguins`. I've also supplied code to shorten the penguins species name for convenient exploration and plotting.

In [2]:
```
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
pd.set_option("future.no_silent_downcasting", True)

penguins = pd.read_csv("palmer_penguins.csv")

# shorten the species name
penguins["Species"] = penguins["Species"].str.split().str.get(0)
```

For today's exercise, keep only the following columns: `'Species'`, `'Island'`, `'Culmen Length (mm)'`, `'Culmen Depth (mm)'`, `'Flipper Length (mm)'`, `'Body Mass (g)'`, `'Sex'`. Calling `penguins.filter([...])` with the column names inside should make this happen. Reassign this table to `penguins`. The updated `penguins` table should have 344 rows and 7 columns.

In [3]:
```
penguins = penguins.filter(['Species', 'Island', 'Culmen Length (mm)', 'Culm
penguins
```

Out[3]:

	Species	Island	Culmen Length (mm)	Culmen Depth (mm)	Flipper Length (mm)	Body Mass (g)	Sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE
...	...	...	...	...	...	...	...
339	Gentoo	Biscoe	NaN	NaN	NaN	NaN	NaN
340	Gentoo	Biscoe	46.8	14.3	215.0	4850.0	FEMALE
341	Gentoo	Biscoe	50.4	15.7	222.0	5750.0	MALE
342	Gentoo	Biscoe	45.2	14.8	212.0	5200.0	FEMALE
343	Gentoo	Biscoe	49.9	16.1	213.0	5400.0	MALE

344 rows × 7 columns

You might have noticed that your table contains rows with `NaN` values. Calling `penguins.dropna()` will remove these rows. Do this below, and reassign the result back to `penguins`. Your updated `penguins` table should have 334 rows and 7 columns.

In [4]:
```
penguins = penguins.dropna()
penguins
```

Out[4]:

	Species	Island	Culmen Length (mm)	Culmen Depth (mm)	Flipper Length (mm)	Body Mass (g)	Sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0	MALE
...	...	...	...	...	...	...	...
338	Gentoo	Biscoe	47.2	13.7	214.0	4925.0	FEMALE
340	Gentoo	Biscoe	46.8	14.3	215.0	4850.0	FEMALE
341	Gentoo	Biscoe	50.4	15.7	222.0	5750.0	MALE
342	Gentoo	Biscoe	45.2	14.8	212.0	5200.0	FEMALE
343	Gentoo	Biscoe	49.9	16.1	213.0	5400.0	MALE

334 rows × 7 columns

In an ideal world, we would train our model on the entire dataset, collect data from new penguins, then test it on the new data. However, this is obviously not feasible in this case. In cases like this, most people randomly split the existing samples into train and test, and "pretend" like the samples in the test set are actually coming from penguins they haven't met yet.

Fill in the blank such that this sentence describes what the code does:

```
We will randomly put 80% of the ___1____ into the __2___ set,
and put the remaining __1_____ into the ____2___ set.
```

- Options for 1: rows or columns

- Options for 2: train or test

The corresponding code:

```
train = penguins.sample(frac=0.8)
test = penguins.drop(index=train.index)
print(train.shape, test.shape)
```

We will randomly put 80% of the rows into the train set, and put the remaining rows into the test set.

In [5]:
```
train = penguins.sample(frac=0.8)
test = penguins.drop(index=train.index)
print(train.shape, test.shape)
```

```
(267, 7) (67, 7)
```

Part B: Manual decision tree

We'll first approach this problem manually, meaning that you'll be the one designing the prediction algorithm, not your computer.

Calculate the mean of each numeric variable in the table PER penguin species in your `train` data.

In [6]:
```
penguin_means = penguins.groupby("Species")["Culmen Length (mm)"].mean()
penguin_means1 = penguins.groupby("Species")["Culmen Depth (mm)"].mean()
penguin_means2 = penguins.groupby("Species")["Flipper Length (mm)"].mean()
penguin_means3 = penguins.groupby("Species")["Body Mass (g)"].mean()

print(penguin_means)
print(penguin_means1)
print(penguin_means2)
print(penguin_means3)
```

```
Species
Adelie       38.823973
Chinstrap    48.833824
Gentoo       47.542500
Name: Culmen Length (mm), dtype: float64
Species
Adelie       18.347260
Chinstrap    18.420588
Gentoo       15.002500
Name: Culmen Depth (mm), dtype: float64
Species
Adelie       190.102740
Chinstrap    195.823529
Gentoo       217.233333
Name: Flipper Length (mm), dtype: float64
Species
Adelie       3706.164384
Chinstrap    3733.088235
Gentoo       5090.625000
Name: Body Mass (g), dtype: float64
```

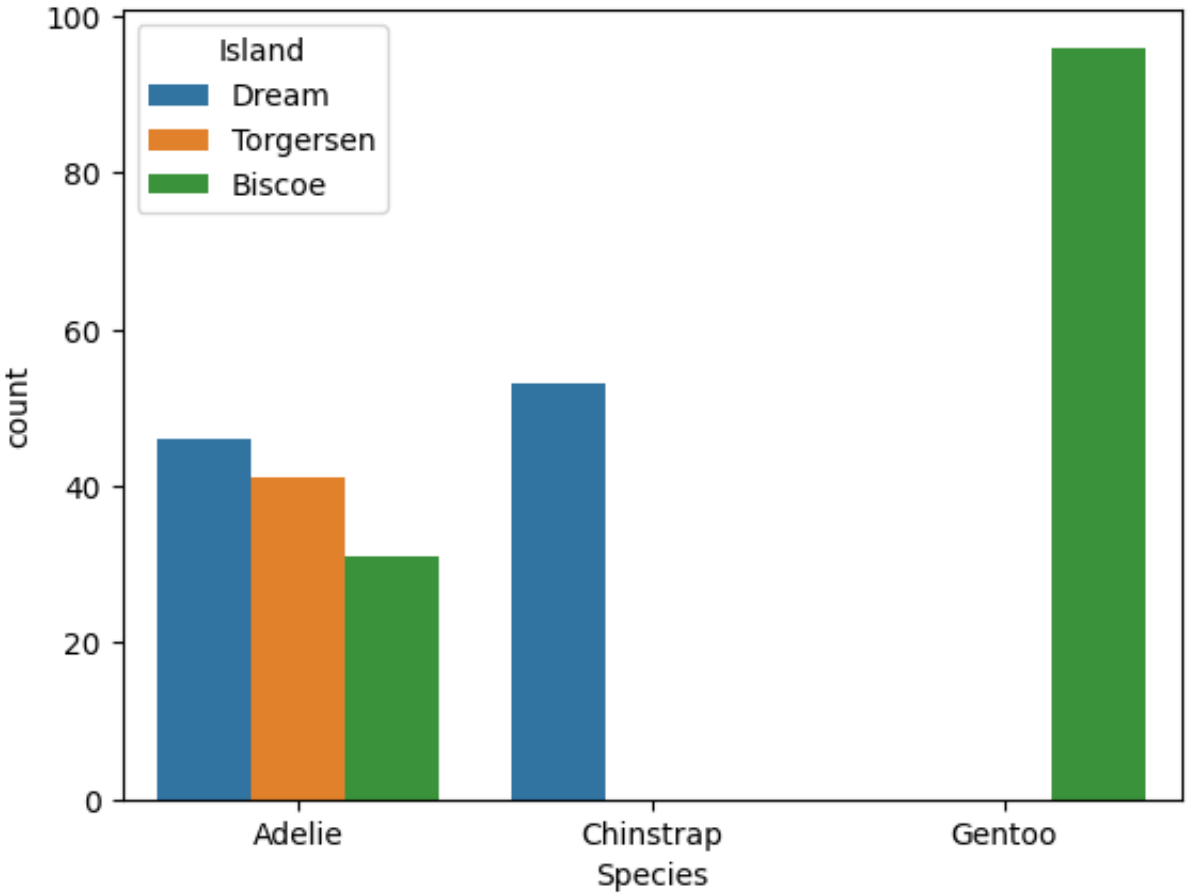
For the categorical variable `Island`, I'll give you the code. Copy and run following code:

```
island_counts =
pd.DataFrame(train.groupby('Species').Island.value_counts())

sns.barplot(island_counts, x='Species', hue='Island', y='count')

island_counts
```

In [7]:
```
island_counts = pd.DataFrame(train.groupby('Species').Island.value_counts())
sns.barplot(island_counts, x='Species', hue='Island', y='count')
island_counts
```

Out[7]:

		count
Species	**Island**	
Adelie	**Dream**	46
	Torgersen	41
	Biscoe	31
Chinstrap	**Dream**	53
Gentoo	**Biscoe**	96



Based on your findings from these tables and barplot, propose a miniature decision tree to help distinguish between the penguin species. Your decision tree might have rules like the following:

1. First, check the island on which the penguin was found.
    A. If Torgersen, then check the body mass.
        a. If the body mass is over 4,000g, then guess Adelie.
        b. Otherwise, guess Chinstrap
    B. If Biscoe, then check the sex of the penguin.
        a. If female, guess Gentoo
        b. Otherwise, guess Chinstrap
    C. If Dream, then guess Adelie.

Your decision tree should operate using **no more than three columns** from the data.

Below your decision tree, write an explanation of how you came up with it and how the tables that you created above informed your choices.

If on Dream Island, check culmen length. If culmen length > 45 mm, guess Chinstrap.

Else, guess Adelie. If on Biscoe Island, check body mass. If body mass > 4500 g, guess Gentoo. Else, guess Adelie. If on Torgersen Island, guess Adelie.

If you'd like (**not required**), you may write your decision tree directly as a Python function. This example algorithm would look like this:

```
def decision_tree(island, mass, sex):
    if island == "Torgersen":
        if mass > 4000:
            return "Adelie"
        else:
            return "Chinstrap"
    elif island == "Biscoe":
        if sex == "FEMALE":
            return "Gentoo"
        else:
            return "Chinstrap"
    else:
        return "Adelie"

decision_tree("Biscoe", 5000, "MALE")
```

Part C: Automated decision tree

Now let's see what the automated version looks like.

Once again, these `scikit-learn` functions don't know how to handle text variables like `Island` and `Sex`, so we'll have to turn them into numbers for them. You can use boolean indexing like we did in lecture, but to save time, I'll give you code that does this quickly.

```
train['Species'] = train.Species.replace({'Adelie': 0, 'Chinstrap':
1, 'Gentoo': 2})
train['Island'] = train.Island.replace({'Dream': 0, 'Biscoe': 1,
'Torgersen': 2})
train['Sex'] = train.Sex.replace({'MALE': 0, 'FEMALE': 1, '.' : 2})

test['Species'] = test.Species.replace({'Adelie': 0, 'Chinstrap':
1, 'Gentoo': 2})
test['Island'] = test.Island.replace({'Dream': 0, 'Biscoe': 1,
'Torgersen': 2})
test['Sex'] = test.Sex.replace({'MALE': 0, 'FEMALE': 1, '.' : 2})

train = train.astype(float)
test = test.astype(float)
```

In [8]:
```
train['Species'] = train.Species.replace({'Adelie': 0, 'Chinstrap': 1, 'Gent
train['Island'] = train.Island.replace({'Dream': 0, 'Biscoe': 1, 'Torgersen'
train['Sex'] = train.Sex.replace({'MALE': 0, 'FEMALE': 1, '.' : 2})

test['Species'] = test.Species.replace({'Adelie': 0, 'Chinstrap': 1, 'Gentoo
test['Island'] = test.Island.replace({'Dream': 0, 'Biscoe': 1, 'Torgersen':
test['Sex'] = test.Sex.replace({'MALE': 0, 'FEMALE': 1, '.' : 2})

train = train.astype(float)
test = test.astype(float)
```

Each of your table needs to be split into two parts (`X` and `y`) for the automated algorithm to understand. Remember that `X` corresponds to a table where each column is a feature or a predictor variable, and `y` corresponds to an array with the target variable or the labels.

I've given you partial code that creates four new variables `y_train` , `X_train` , `X_test` , `y_test` . **Fill in the missing parts marked with ...**, then copy and run the code. The answer is a single column name that is the same in all four places. Pause and make sure you understand what is going on.

```
y_train = train[...] # select column with target variable
X_train = train.drop(columns=[...]) # keep all other columns with
predictor variables
print(X_train.shape, y_train.shape)

y_test = test[...]
X_test = test.drop(columns=[...])
print(X_test.shape, y_test.shape)
```

In [9]:
```
y_train = train["Species"] # select column with target variable
X_train = train.drop(columns=["Island", "Species", "Sex"]) # keep all other
print(X_train.shape, y_train.shape)

y_test = test["Species"]
X_test = test.drop(columns=["Island", "Species", "Sex"])
print(X_test.shape, y_test.shape)
```

```
(267, 4) (267,)
(67, 4) (67,)
```

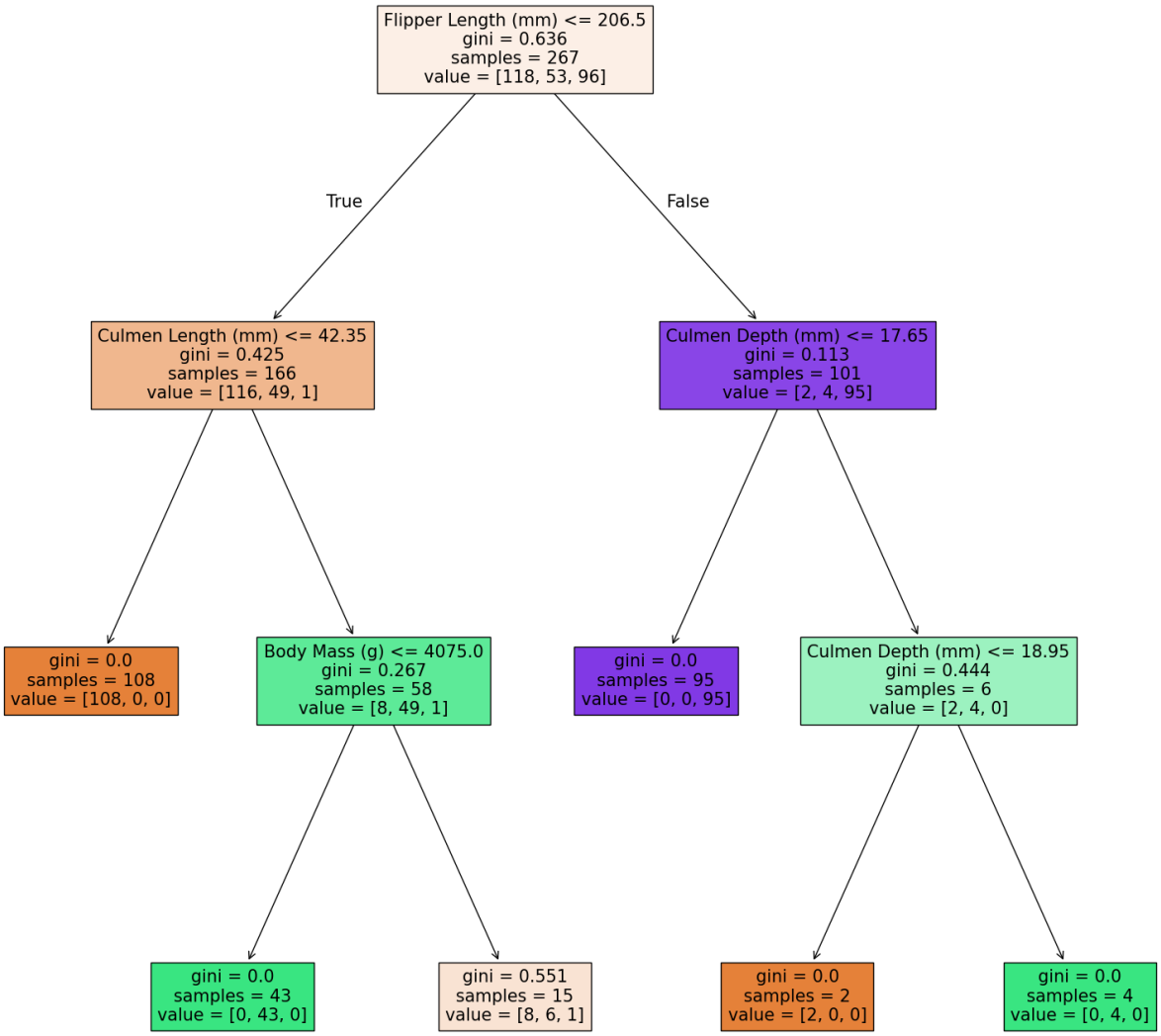
We are almost done!

I've also given you mostly finished code that will automatically create a decision tree classifier. Put in `X_train` , `y_train` , `X_test` , `y_test` in appropriate places, then copy and run the code.

```
from sklearn.tree import DecisionTreeClassifier, plot_tree

T = DecisionTreeClassifier(max_depth=3)
T.fit(..., ...) # train the model

print('Score on train:', T.score(..., ...)) # evaluate on train
data
print('Score on test:', T.score(..., ...)) # evaluate on test data

fig, ax = plt.subplots(1, figsize = (20, 20))
p = plot_tree(T, filled = True, feature_names = X_train.columns)
```

In [10]:
```
from sklearn.tree import DecisionTreeClassifier, plot_tree
T = DecisionTreeClassifier(max_depth=3)
T.fit(X_train, y_train)
print('Score on train:', T.score(X_train, y_train))
print('Score on test:', T.score(X_test, y_test))
fig, ax = plt.subplots(1, figsize = (20, 20))
p = plot_tree(T, filled = True, feature_names = X_train.columns)
```

```
Score on train: 0.9737827715355806
Score on test: 0.9253731343283582
```



What do you think about this tree? Do you think this does a good job at classifying penguin species? Did your computer create a similar algorithm to your manual one, or something very different?

The tree does a good job and seems pretty accurate on both train and test data. It's not exactly like my manual one but it used similar features like body mass and culmen length. Overall, it makes sense and works well.

In [ ]:

In [ ]:

Final Check-In

As we wrap up this lab, we'd love to hear how things are going with your project planning.

- How are things going with your team so far?
- Anything you'd like us to know as your team moves forward with the project?

Our group is very cohesive and works well together. We had a solid plan going into the project, and everything has been going smoothly so far. We don't have anything specific to share at the moment.

In [ ]: