A quick introduction to machine learning

Author: Leonardo Uieda

This notebook is meant as a very brief hands-on introduction to machine learning. It will cover some of the common nomenclature, principles, and applications. It's designed to be taught as a 1-2 hour session with live-coding.

📖 • What is ML?

Some features of machine learning (from my personal point of view):

- Focus on practical problems
- Learning from data and making predictions
- Overlap with statistics and optimization
- Computational approach

Oversimplified summary: Fit a mathematical model to data and use it to make predictions.

🗺️ • Glossary

model
: Mathematical formula used to approximate the data

parameters
: Variables that define the model and control its behavior

labels/classes
: Quantity/category that we want to predict

features
: Measurements (information) used as predictors of labels/classes

training
: Using features and known labels/classes to fit the model (estimate its parameters)

hyper-parameters
: Variables that influence the training and the model but are not estimated during training

unsupervised learning
: Extract information and structure from the data without "training". Examples: clustering, principal component analysis.

supervised learning
: Fit a model using data to "train" it for making predictions. Examples: regression, classification, spam detection, recommendation systems

Disclaimer: I'm not an ML researcher. Don't quote me on this.

🐍 • Libraries

In Python, the main tool used for machine learning is scikit-learn. We'll use it and some of the other scientific Python *stack* to play with some data as we work through the core principles of machine

learning.

In [1]:
```
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import (
    preprocessing,
    neighbors,
    decomposition,
    cluster,
    metrics,
    model_selection,
    pipeline,
)
```

🐧 • Data

For this tutorial, we'll use one of the sample datasets from the seaborn library. It contains measurements of anatomical features of 3 different species of penguin collected Dr. Kristen Gorman and the Palmer Station, Antarctica LTER, a member of the Long Term Ecological Research Network.

The data are distributed under a CC-0 license from the GitHub repository allisonhorst/palmerpenguins. The data were originally published in Gorman et al. (2014).

In [2]:
```
data = sns.load_dataset("penguins")
data
```

Out[2]:

	species	island	bill_length_mm	bill_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

The seaborn function returns the data in a `pandas.DataFrame`, which would be standard way to load data stored in CSV files or Excell spreadsheets. Each row contains information about an individual and each column is a type of observation.

> **Example of good practice:** Notice how each numerical column name includes the unit! This is a great example and your future self will be very grateful if you replicate this for

> your own data. As a bonus, also use all lowercase characters and underscores `_` instead of spaces or dashes `-`.

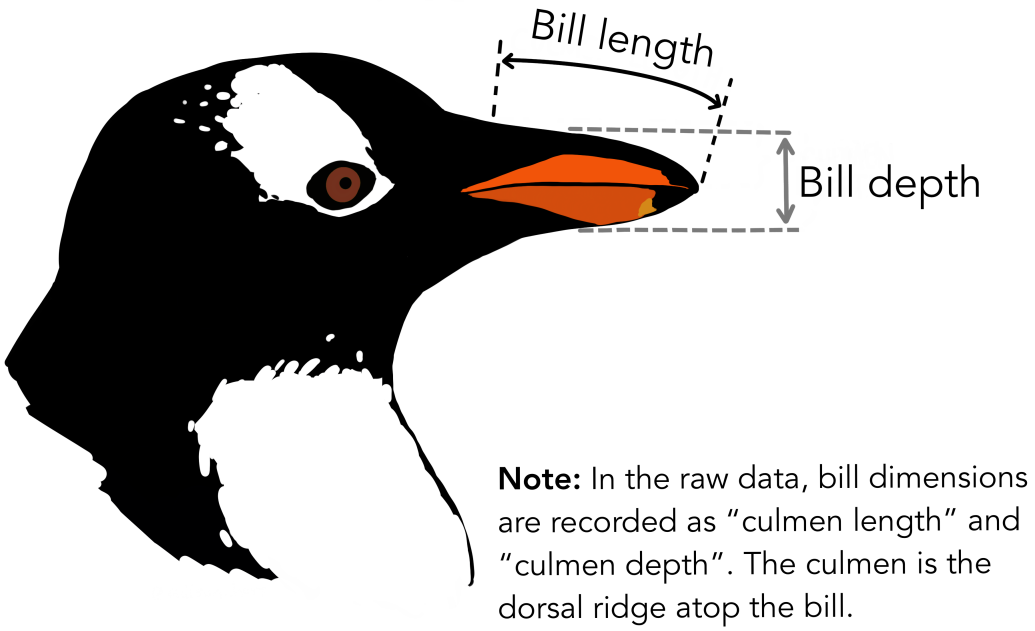


Figure: Illustration of the penguin beak measurements. Artwork by @allison_horst.

Goal

Give the data that we have, let's set ourselves the following goal: **predict the species from the anatomical measurements.**

Since the species is a category and not a continuous variable, we're dealing with a **classification problem** and not a regression problem. This is important to know since it will guide the models and methods that we use and research.

Cleaning

Data cleaning and inspection is probably the most crucial part of a machine learning workflow. Without first having sufficient, good, standardized, and cleaned data there is no algorithm that will solve your problems.

The first thing to note is that our data has missing values (`NaN` or "Not a Number"). Some of the rows don't have all of the information for all features. This can cause a lot of problems in our processing and machine learning pipelines. So let's first see how many there are and which values are missing.

In [3]:
```
data[np.any(data.isna(), axis=1)]
```

Out[3]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
8	Adelie	Torgersen	34.1	18.1	193.0	3475.0	NaN
9	Adelie	Torgersen	42.0	20.2	190.0	4250.0	NaN
10	Adelie	Torgersen	37.8	17.1	186.0	3300.0	NaN

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
11	Adelie	Torgersen	37.8	17.3	180.0	3700.0	NaN
47	Adelie	Dream	37.5	18.9	179.0	2975.0	NaN
246	Gentoo	Biscoe	44.5	14.3	216.0	4100.0	NaN
286	Gentoo	Biscoe	46.2	14.4	214.0	4650.0	NaN
324	Gentoo	Biscoe	47.3	13.8	216.0	4725.0	NaN
336	Gentoo	Biscoe	44.5	15.7	217.0	4875.0	NaN
339	Gentoo	Biscoe	NaN	NaN	NaN	NaN	NaN

It's not that many and most are in the `sex` column, which we won't use here. So let's first drop that entire column from our dataset.

In [4]:
```
data.drop("sex", axis=1, inplace=True)
data
```

Out[4]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0
3	Adelie	Torgersen	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0
...	...	...	...	...	...	...
339	Gentoo	Biscoe	NaN	NaN	NaN	NaN
340	Gentoo	Biscoe	46.8	14.3	215.0	4850.0
341	Gentoo	Biscoe	50.4	15.7	222.0	5750.0
342	Gentoo	Biscoe	45.2	14.8	212.0	5200.0
343	Gentoo	Biscoe	49.9	16.1	213.0	5400.0

344 rows × 6 columns

And then we drop every row that has at least 1 missing value.

In [5]:
```
data.dropna(axis=0, inplace=True)
data
```

Out[5]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0
...	...	...	...	...	...	...
338	Gentoo	Biscoe	47.2	13.7	214.0	4925.0

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
340	Gentoo	Biscoe	46.8	14.3	215.0	4850.0
341	Gentoo	Biscoe	50.4	15.7	222.0	5750.0
342	Gentoo	Biscoe	45.2	14.8	212.0	5200.0
343	Gentoo	Biscoe	49.9	16.1	213.0	5400.0

342 rows × 6 columns

Now the problem is that the index doesn't match the number of rows in our table (notice that the last row is 343 when we only have 342 rows). This happens because `pandas` deletes rows but keeps the original index numbers intact (which is a sensible default).

Let's re-align the index with the number of rows.

In [6]:
```
data.reset_index(drop=True, inplace=True)
data
```

Out[6]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0
3	Adelie	Torgersen	36.7	19.3	193.0	3450.0
4	Adelie	Torgersen	39.3	20.6	190.0	3650.0
...	...	...	...	...	...	...
337	Gentoo	Biscoe	47.2	13.7	214.0	4925.0
338	Gentoo	Biscoe	46.8	14.3	215.0	4850.0
339	Gentoo	Biscoe	50.4	15.7	222.0	5750.0
340	Gentoo	Biscoe	45.2	14.8	212.0	5200.0
341	Gentoo	Biscoe	49.9	16.1	213.0	5400.0

342 rows × 6 columns

Now we have data that has no missing values and we're ready to start.

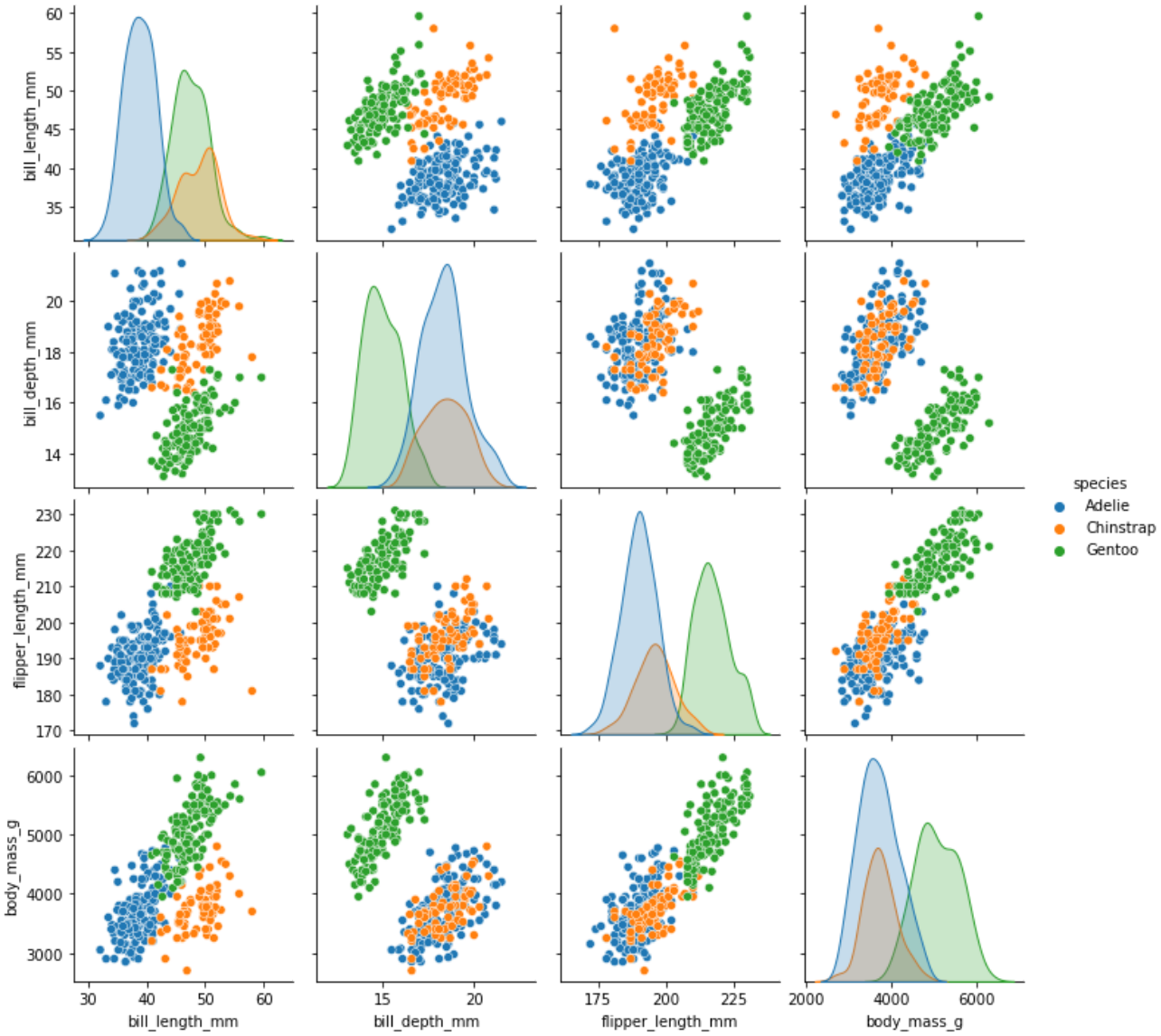
Visualize

Now that we have clean and ordered data, it's important to visualize the data so that we know what we're dealing with. For this dataset, we have few enough features (columns) and samples (rows) to visualize it straight away. Otherwise, we should apply some sort of dimensionality reduction first.

A standard way to plot the different features and the pair-wise relationship between then is with a "pair plot". Thankfully, we don't have to create these ourselves with `matplotlib` since `seaborn` has a function just for this.

In [7]:
```
sns.pairplot(data, hue="species")
```

Out[7]:
```
<seaborn.axisgrid.PairGrid at 0x7f18a05dd4c0>
```



In the figure above, each graph is a cross-plot of 2 different numerical columns in our dataset. The diagonal in the middle is the estimated distribution of that numerical column. Each plot is colored by the respective species.

What we can see from this is that some features are highly correlated for some species. For example, body mass and flipper length or flipper length and and bill depth (for Chinstrap and Adelie). Notice also how the distributions overlap heavily but the overlap isn't always the same. For example, the body mass of Adelie and Chinstrap penguins overlap but not their bill length.

If all features were correlated and overlapping, we'd have very little chance of differentiating between the different classes (species) using them and our machine learning pipeline would halt here. It would indicated that we probably need to gather data from a different feature.

Since that's not the case, let's move on and see what we can do.

Format

Scikit-learn and a lot of other Python libraries for machine learning (which largely try to emulate scikit-learn), require the data to be formatted in a certain way before it can be used.

Features: The observations that we will use as predictors of the species. These need to be stored in a 2D array (basically a matrix). Each **row** of this matrix is a **sample** (in our case the measurements of

an individual) and each **column** corresponds to a different **feature**. This matrix is often called `X` in the scikit-learn documentation.

We can make the feature matrix `X` from our `pandas.DataFrame` by selecting the columns and getting the `.values` attribute (which returns the data in a `DataFrame` as a 2D array).

In [8]:

```
feature_columns = ["bill_length_mm", "bill_depth_mm", "flipper_length_mm", "body_m
X = data[feature_columns].values
X
```

Out[8]:

```
array([[  39.1,   18.7,  181. , 3750. ],
       [  39.5,   17.4,  186. , 3800. ],
       [  40.3,   18. ,  195. , 3250. ],
       ...,
       [  50.4,   15.7,  222. , 5750. ],
       [  45.2,   14.8,  212. , 5200. ],
       [  49.9,   16.1,  213. , 5400. ]])
```

This is another example of why getting the data sorted into a standard format early is important. Making the feature matrix would be difficult to do if we hadn't done that previously.

Labels/classes: The thing that we are trying to predict from our features. In our case, we have labels for each sample in our dataset (the `species` column). The labels must be stored in a 1D array (can be 2D depending on the application). In the scikit-learn documentation this is often called `y` .

In [9]:

```
y = data.species.values
y
```

Out[9]:

```
array(['Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
```

```
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo'], dtype=object)
```

That we have our `X` and `y` variables, we can start trying to learn from the data with scikit-learn.

🔭 • Unsupervised learning: Finding patterns in the data

Sometimes we may have a bunch of features but no labels already present in the data. Without them, we can't train our models to predict the labels from the features. But that doesn't mean that there is nothing we can do.

Unsupervised learning methods are all about finding patterns in the data without first training a model with given labels. An example of this is **clustering**, which tries to categorize the data based on how much they group around each other.

From our pair-plot above, we can see that the 3 species tend to be clustered around certain ranges of values depending on the features. There is more or less overlap between species depending on the features used so we can expect that clustering methods may struggle a bit to separate them.

Let's see what the simplest form of clustering (K-Means) can do. We need to tell it the number of clusters to look. Thankfully, we already know that there are only 3 possible species.

In [10]:

```
kmeans = cluster.KMeans(n_clusters=3, random_state=0)
kmeans.fit(X)
cluster_labels = kmeans.predict(X)
cluster_labels
```

Out[10]:

```
array([0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 2, 0, 2, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0,
       2, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0,
       2, 0, 2, 0, 0, 0, 2, 0, 2, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 0, 0, 0,
       0, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 0, 0, 0, 0, 2, 0,
       2, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 0,
```

```
2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0,
0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0,
2, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 2, 0,
2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 2,
1, 2, 1, 1, 2, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 1, 1, 2,
1, 1, 1, 2, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 0, 1, 2, 2, 1,
2, 2, 1, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
1, 2, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
1, 1, 1, 2, 1, 2, 1, 2, 2, 2, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 1, 1,
1, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 1], dtype=int32)
```

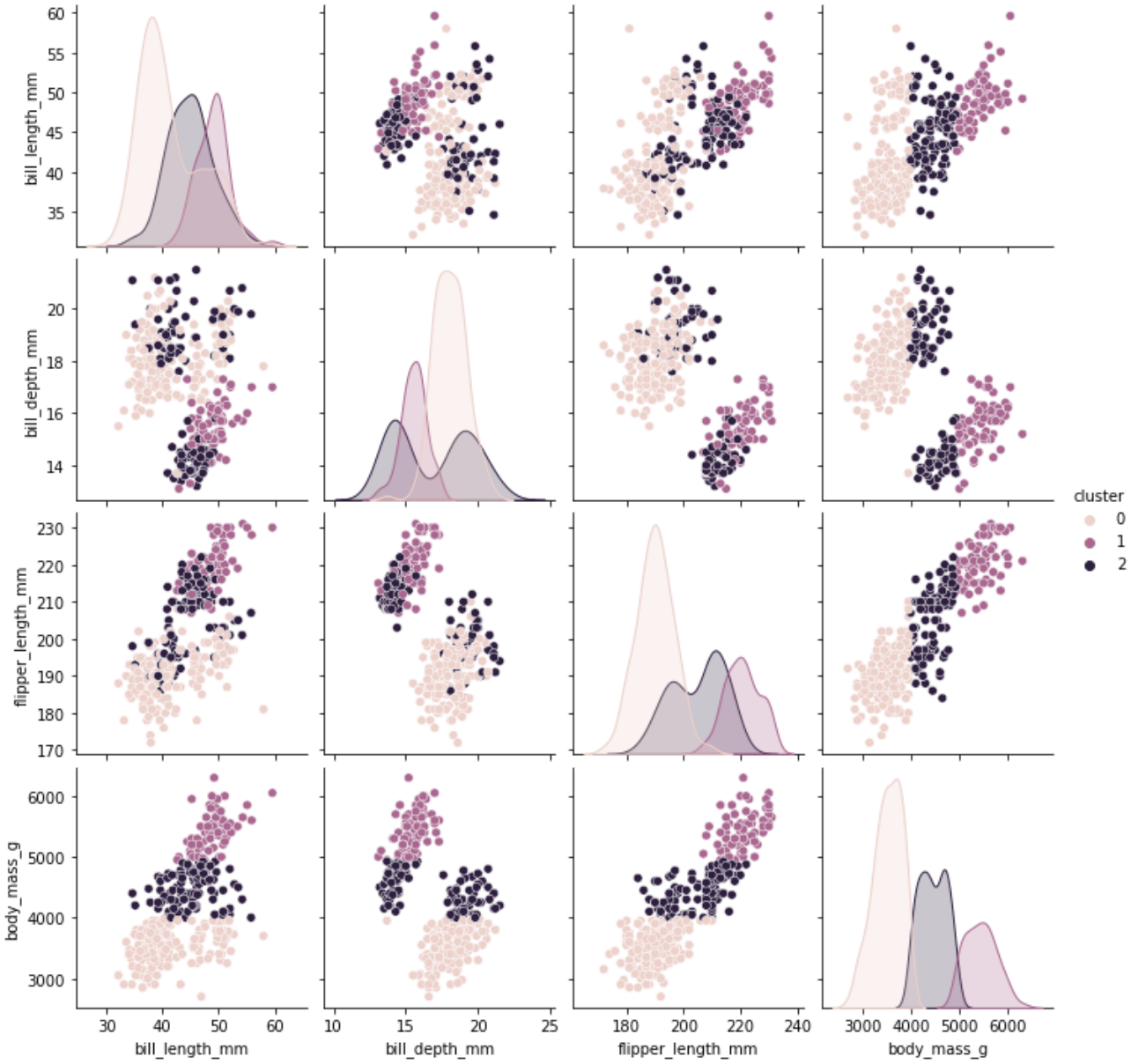
The prediction output is a value of 0, 1, or 2 indicating to which cluster each data sample belongs. We don't know which one corresponds to which species (if any) but we can do a visual inspection to check the performance.

To do so, we can assign the k-means labels as a column in our `data` variable and use it to create a new pair-plot.

In [11]:
```
data = data.assign(cluster=cluster_labels)

sns.pairplot(data, hue="cluster")
```

Out[11]:
```
<seaborn.axisgrid.PairGrid at 0x7f18a05dd6d0>
```



> **Pro tip:** Open a "New view for notebook" in the JupyterLab "File" menu to place the original pair-plot and our k-means prediction side by side for comparison.

The clustering doesn't seem to have performed very well in the prediction of the labels. For one, it seems to have divided the body mass 3 ways into low, mid, and high values.

I guess this means that k-means is not able to predict the species properly so we should move on to more complex and sophisticated models. 🤷

The importance of scaling

🛑 Stop! 🛑

Before moving on, we should always try to **understand why the model failed** and **check if there is a problem with our data**. If our data are faulty or need some more cleaning, there is no amount of cleverness in model choice that will overcome that. As the saying goes:

> **Garbage in, garbage out.** 🗑️

Going back to our data, let's have a look at the statistics for each feature.

In [12]:
```
data[feature_columns].mean(axis=0)
```

Out[12]:
```
bill_length_mm         43.921930
bill_depth_mm          17.151170
flipper_length_mm     200.915205
body_mass_g          4201.754386
dtype: float64
```

In [13]:
```
data[feature_columns].std(axis=0)
```

Out[13]:
```
bill_length_mm         5.459584
bill_depth_mm          1.974793
flipper_length_mm     14.061714
body_mass_g          801.954536
dtype: float64
```

Notice that the body mass is 2 orders of magnitude larger in both mean and standard deviation than the bill measurements. That should raise some red flags since our clustering seems to have divided the data based on body mass alone.

K-means calculates the clusters based on the distance between points and their "center", as defined by the mean of all their dimensions (features). And what we know about means is **large values will skew the mean towards them**. So it's no wonder that the clusters are being picked based on the body mass along.

What can we do to mitigate this? The standard practice, and an **underlying assumption for many machine learning methods**, is that all features have **zero mean and unit standard deviation** (in other words, are as close to normally distributed as possible).

The simplest possible method to resolve this is to subtract the mean and divide by the standard deviation for each feature. In scikit-learn, we can do this with the `StandardScaler`.

In [14]:
```
scaler = preprocessing.StandardScaler()
```

```
scaler.fit(X)
X_scaled = scaler.transform(X)
X_scaled
```

Out[14]:
```
array([[-0.88449874,  0.78544923, -1.41834665, -0.56414208],
       [-0.81112573,  0.1261879 , -1.06225022, -0.50170305],
       [-0.66437972,  0.43046236, -0.42127665, -1.18853234],
       ...,
       [ 1.18828874, -0.73592307,  1.50164406,  1.93341896],
       [ 0.23443963, -1.19233476,  0.7894512 ,  1.24658968],
       [ 1.09657248, -0.53307343,  0.86067049,  1.49634578]])
```

In [15]:
```
X_scaled.mean(axis=0)
```

Out[15]:
```
array([ 1.66208827e-16, -1.41277503e-15, -8.31044135e-16,  4.15522068e-17])
```

In [16]:
```
X_scaled.std(axis=0)
```

Out[16]:
```
array([1., 1., 1., 1.])
```

Now let's try the clustering again but with our scaled features.

In [17]:
```
kmeans.fit(X_scaled)
cluster_labels_scaled = kmeans.predict(X_scaled)
```

Instead of trying to visually match the cluster labels with the species, let's see if we can match them with code.

In [18]:
```
cluster_labels_scaled
```

Out[18]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,
       1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1,
       2, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2,
       2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

Since our data was originally sorted by species, we can tell from the prediction that:

- 0 = Gentoo
- 1 = Adelie
- 2 = Chinstrap

Let's make an array that replaces the cluster labels with the species name and add it to our `DataFrame`.

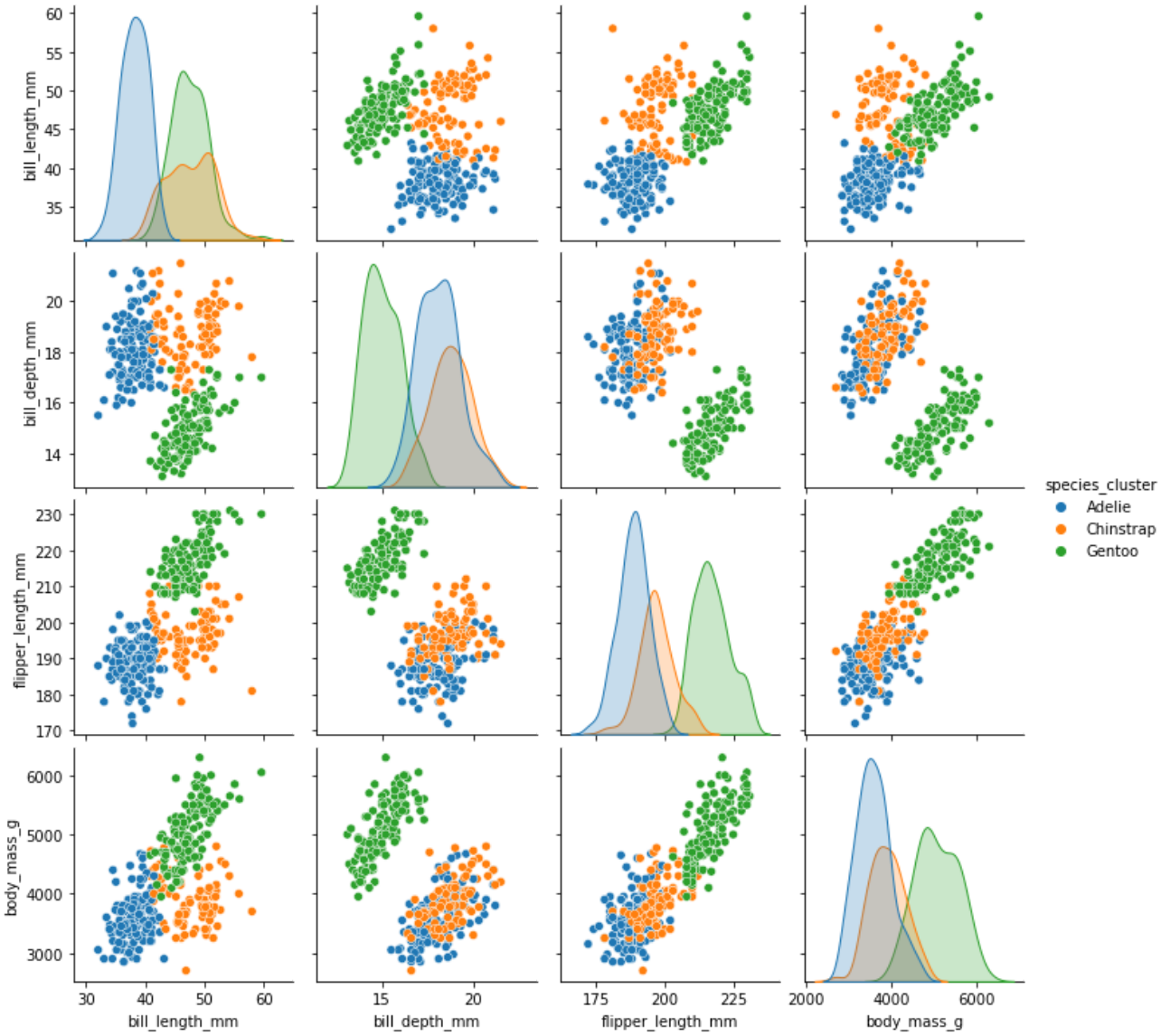
In [19]:
```
cluster_species_scaled = np.empty_like(data.species)
```

```
cluster_species_scaled[cluster_labels_scaled == 0] = "Gentoo"
cluster_species_scaled[cluster_labels_scaled == 1] = "Adelie"
cluster_species_scaled[cluster_labels_scaled == 2] = "Chinstrap"

data = data.assign(species_cluster=cluster_species_scaled)
```

In [20]:
```
sns.pairplot(data, hue="species_cluster", vars=feature_columns)
```

Out[20]:
```
<seaborn.axisgrid.PairGrid at 0x7f18974bd1c0>
```



Comparing these predicted cluster labels with our actual known species shows that **k-means is a decent method to predict species**. Of course, the clustering is not perfect and some samples are mislabeled. Let's see if we can do better with a supervised learning approach.

I can't stress this enough:

> **Always start with the simplest model and check your data before moving to more complex models!**

🤖 • Supervised learning: Training models for prediction

Since we have labels for our data (the species), we can train models to predict these labels based on the input features. Machine learning techniques that follow this procedure fall under the supervised learning category. Examples of supervised learning include linear regression, support vector machines, neural networks (including deep learning), and more. Once we have trained our model, we can then use it to predict the labels of new data that comes in without labels (new penguin measurements by someone who doesn't know the species).

The simplest possible supervised learning model for our classification problems is a k-nearest-neighbors classifier (kNN). The general idea is to assign labels based on the labels of a sample's *k* nearest neighbors (the label with a majority is used). This method is fast and scales well to larger datasets so it's a good choice for a first pass.

In [21]:
```
knn = neighbors.KNeighborsClassifier()
knn.fit(X, y)
species_prediction = knn.predict(X)
species_prediction
```

Out[21]:
```
array(['Adelie', 'Adelie', 'Chinstrap', 'Adelie', 'Chinstrap', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Gentoo', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Chinstrap',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Gentoo', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Gentoo', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Chinstrap', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Chinstrap', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Gentoo', 'Adelie', 'Gentoo', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Gentoo', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Gentoo', 'Adelie', 'Gentoo', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Chinstrap', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Chinstrap', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Chinstrap', 'Adelie', 'Adelie', 'Adelie', 'Adelie', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Adelie', 'Adelie',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Adelie', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Adelie', 'Chinstrap', 'Adelie',
       'Adelie', 'Adelie', 'Chinstrap', 'Adelie', 'Chinstrap', 'Adelie',
       'Adelie', 'Chinstrap', 'Chinstrap', 'Adelie', 'Gentoo', 'Adelie',
       'Adelie', 'Adelie', 'Adelie', 'Chinstrap', 'Adelie', 'Chinstrap',
       'Gentoo', 'Adelie', 'Adelie', 'Chinstrap', 'Chinstrap', 'Adelie',
       'Adelie', 'Chinstrap', 'Adelie', 'Chinstrap', 'Adelie',
       'Chinstrap', 'Chinstrap', 'Adelie', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Adelie', 'Adelie', 'Chinstrap', 'Chinstrap',
       'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap', 'Chinstrap',
       'Adelie', 'Chinstrap', 'Chinstrap', 'Adelie', 'Chinstrap',
       'Adelie', 'Gentoo', 'Adelie', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Adelie', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
```

```
       'Gentoo', 'Gentoo', 'Adelie', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Chinstrap', 'Gentoo',
       'Adelie', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Adelie', 'Gentoo', 'Adelie', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo', 'Gentoo',
       'Gentoo', 'Gentoo', 'Gentoo'], dtype=object)
```

An advantage of the supervised model is that is can give us the predicted species directly, without us having to guess which label corresponds to which.

Instead of looking at the pair-plot for this again, let's try a more quantitative approach. We can calculate something called the accuracy score (read the reference for the maths behind it). Basically, it tells us how close two non-numerical arrays are by counting the number of entries that are equal and dividing by the number of samples. If all entries are the same, the score will be 1. If none of them are, the score will be 0.

In [22]:

```
metrics.accuracy_score(y, species_prediction)
```

Out[22]:

```
0.8391812865497076
```

That's not a bad score! Particularly when we consider that we did nothing to tune the model or configure it in any way.

One thing that may have sparked something in your mind is that the kNN is calculating *distances* between samples, just like our K-means clustering was. And so, it's very likely suffering from the same scaling issue we faces there. Let's see if using the scaled version of the features improves our accuracy score.

In [23]:

```
knn = neighbors.KNeighborsClassifier()
knn.fit(X_scaled, y)
species_prediction_scaled = knn.predict(X_scaled)

metrics.accuracy_score(y, species_prediction_scaled)
```

Out[23]:

```
0.9912280701754386
```

That's a big improvement! The prediction is almost perfect now.

For good measure, let's create our pair plot again with the kNN predictions this time.

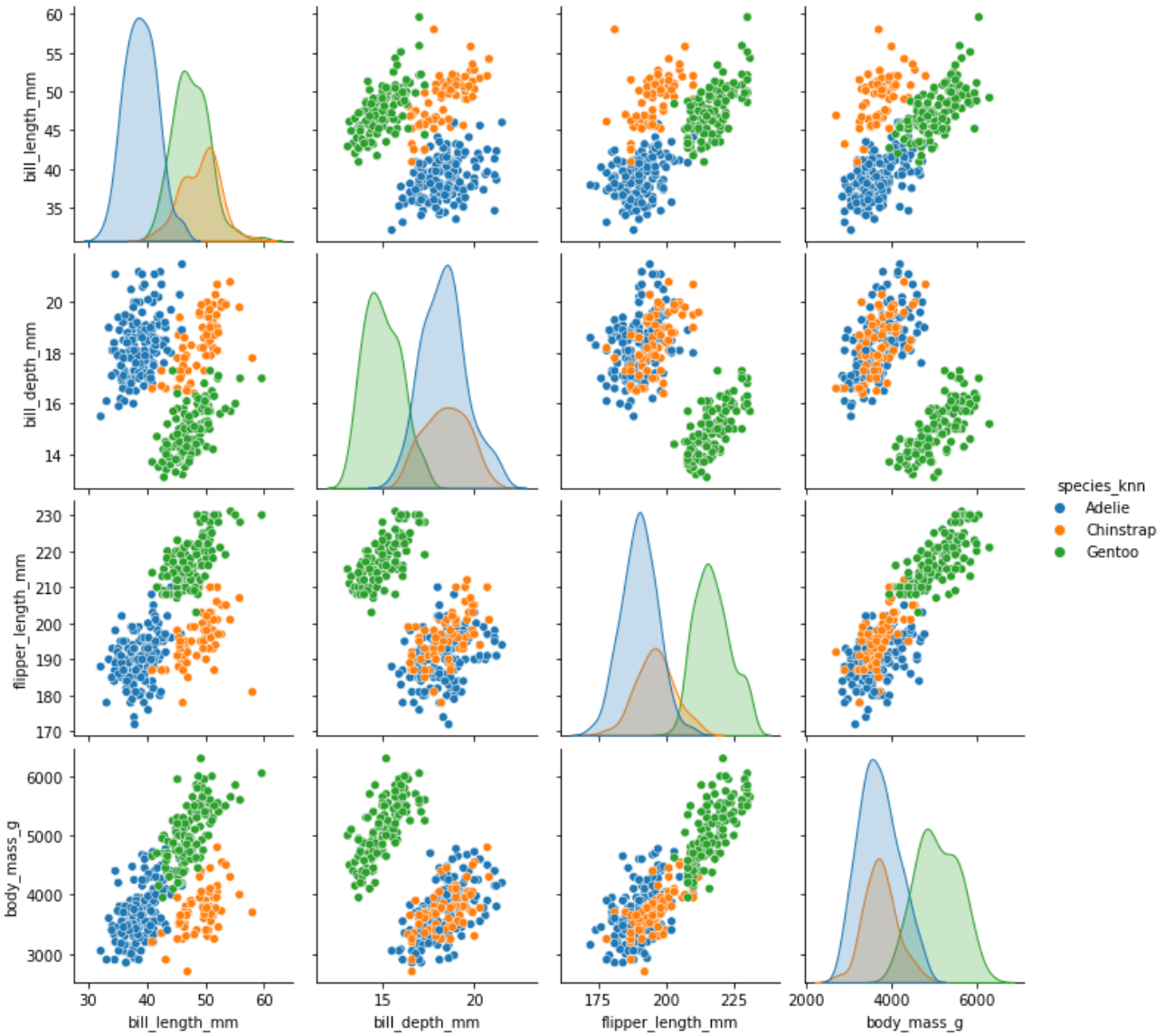
In [24]:

```
data = data.assign(species_knn=species_prediction_scaled)

sns.pairplot(data, hue="species_knn", vars=feature_columns)
```

Out[24]:

```
<seaborn.axisgrid.PairGrid at 0x7f188c0ea7c0>
```



Since scaling is such an important thing, we can make it a permanent part of our machine learning workflow for this problem by using a scikit-learn `Pipeline` .

In [25]:
```
classifier = pipeline.make_pipeline(
    preprocessing.StandardScaler(), neighbors.KNeighborsClassifier(),
)
classifier.fit(X, y)
species_prediction_pipeline = classifier.predict(X)

metrics.accuracy_score(y, species_prediction_pipeline)
```

Out[25]:
```
0.9912280701754386
```

What is this high score actually telling us? It's calculated by comparing the known labels that we used to train our model against the labels predicted by the model. So it indicates how well our model fits the data. That's a good thing to know. But our **data has noise** (for example, misidentified species) so we probably don't want to fit the data perfectly, otherwise we'd be replicating noise in the predictions. This is often called **over-fitting** in machine learning and is a major pitfall to avoid!

What we actually want to know is not how well we can fit the training data, but **how well will our model perform when given new data**. In other words, what is the predictive power of our model? The accuracy scores above don't tell us anything about that.

🕵🏽 • Validation: How good are our predictions?

If we want to answer the question *"how well does our model perform on new data"*, we have to **keep back some data** from the training process. We can then use this data to **test our model's performance** on unseen data. This is known as **cross-validation**, where we validate the model using the data itself.

With scikit-learn, the easiest way to do this is with the `train_test_split` function. By default, it splits the input data randomly into 75% training data and 25% testing data.

In [26]:
```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y)
X_train.shape, X_test.shape
```

Out[26]:
```
((256, 4), (86, 4))
```

The procedure now is to **fit** our model pipeline (scaling + kNN) on the **training data**.

In [27]:
```
classifier.fit(X_train, y_train)
```

Out[27]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('kneighborsclassifier', KNeighborsClassifier())])
```

And then **score** the trained model prediction against the **testing data**.

In [28]:
```
metrics.accuracy_score(y_test, classifier.predict(X_test))
```

Out[28]:
```
1.0
```

Doing that we get a lower score that is more representative of the accuracy we should expect when applying our model in the real world. It's still a very good score, though!

A limitation of this method is that the score is dependent on how the data were split. If you run the entire code above (split, train, test) multiple times, you will get different scores depending on the roll of the dice (give it a try!).

We can do better by using a more exhaustive form of cross-validation known as K-Fold. This involves splitting the data multiple times systematically and alternating which part is used to training and testing. This way, all of our data has been used for both purposes by the end.

Scikit-learn provides a useful function for performing cross-validation that does the splitting, training, testing loop for us and returns the score from each iteration.

In [29]:
```
scores = model_selection.cross_val_score(classifier, X, y)
scores
```

Out[29]:
```
array([0.98550725, 0.98550725, 0.98529412, 0.98529412, 1.        ])
```

Now we can calculate the mean score, which will be more stable across multiple runs and a better representation of our model accuracy.

In [30]:
```
scores.mean()
```

```
0.9883205456095482
```

Out[30]:

Again, that's a very good score!

Let's compare this against the accuracy of our unsupervised K-means clustering.

In [31]:
```
metrics.accuracy_score(y, data.species_cluster)
```

Out[31]:
```
0.9152046783625731
```

That's a good improvement in the prediction accuracy!

A few things to keep in mind:

- Our data are ideal since they contain roughly the same number of samples for each of the 3 categories. This is very likely not going to the case in other scenarios and there are special techniques for dealing with "class imbalance", as it's called.
- K-fold splitting for categorical data is not optimal since it can create class imbalance depending on the split. Other forms of cross-validation exist that take this into account.

🔖 • Summary

The main take home messages of this tutorial are:

- There are 2 main types of machine learning: **supervised** (when you have labels to train the model) and **unsupervised** (when you don't) learning.
- **Always** start by thoroughly inspecting, cleaning, and formatting your data. This includes standardization, principle component analysis, encoding, and more.
- Start with the **simplest possible model** (for example, `KMeans` for clustering, `LinearModel` for regression, or `KNeighborsClassifier` for classification) **before** reaching for more complex models (looking at you deep learning 👺). Your goal with other models is always to be at least better than these simple models.
- Understanding **how methods work is essential** for diagnosing problems. Treating them as black boxes will only take you so far.
- Always **split your data and cross-validate** to assess model performance and avoid over-fitting.
  - OK: Single random split into training and testing (`train_test_split`)
  - Good: K-Fold splits into training and testing (`cross_val_score`)
  - Great: Single random split into training and *validation* and then K-Fold split training and testing (`train_test_split` + `cross_val_score`)
  - Best: Nested cross-validation (2 x `cross_val_score`)

👩🏾‍🎓 • Further reading

What to look into for the future.

- The scikit-learn documentation is full of examples, theory, and tutorials. This part on model tuning is a must-read!
- The Software Underground has some video tutorials on machine learning for geoscience as part of their Transform events:
  - Tutorial: Big data lithology prediction with machine learning

    - Tutorial: Back to basics with data and statistical concepts for data analysis
    - Tons of other tutorials on their YouTube channel.
- Jake VanderPlas has an excellent video tutorial.

⚖️ • License

The original material for this tutorial can be found at leouieda/ml-intro. Comments, corrections, and additions are welcome.