GUIDE TO DATA ANALYSIS

Nikola Živković

RUBIK'S CODE
BUILDING SMART APPS

Editor: Ivana Milićev

Cover design and book layout: Slavica Aj

Preface

The crucial part of any such application is data. You might have heard the term "garbage in – garbage out", which is often used by more experienced data scientists. This is referring to the situation in which data is not prepared for processing. That is why we utilize different techniques of data analysis and data visualization to clean up the data and remove the "garbage".
One of the main tasks of **data scientists** is to visualize and analyze data. This can happen during two **phases** of developing a solution. First, when we start working on a project, we need to understand the data. Before we can use some fancy machine learning or **deep learning** model, we need to understand the data we are dealing with. This is done through **exploratory data analysis** and **feature engineering**.

During this process, we are looking into data and we are trying to clean it and reshape it, so it can be processed by machine learning algorithms. We usually create a lot of graphs and plots that can help us see how data is distributed and what is going on. You know what they say – a **picture** is worth a thousand words. Also, at the end of the project, data scientists usually have to explain to the client **why** their solution is good and how it will impact the client's business. Here, it is also important to use data visualizations that are in general more **understandable** than complicated mathematical formulas.

That is why this document is organized in three sections:

- Data visualization
- Exploratory Data Analysis
- Feature Engineering

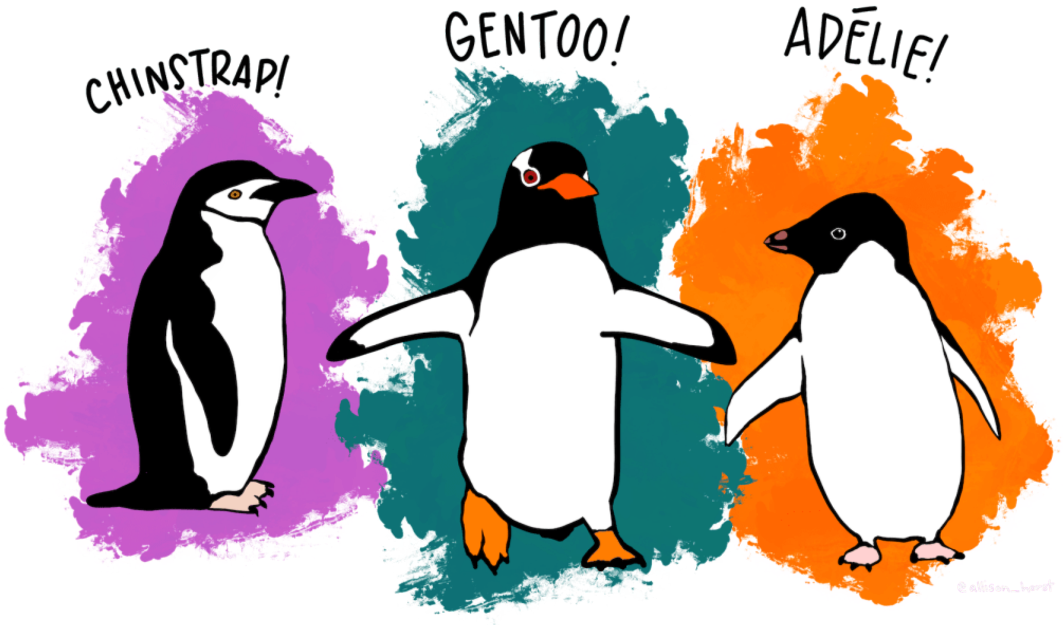
Data visualization is an integral part of every project and it is used for Exploratory Data Analysis and Feature Engineering. That is why it got a section for itself. During Exploratory Data Analysis, we explore the nature of each feature in the dataset and their relationships. With Feature Engineering, we prepare data for machine learning processing. However, before both of those sections, we have some Data Analysis intuition.

Prerequisites

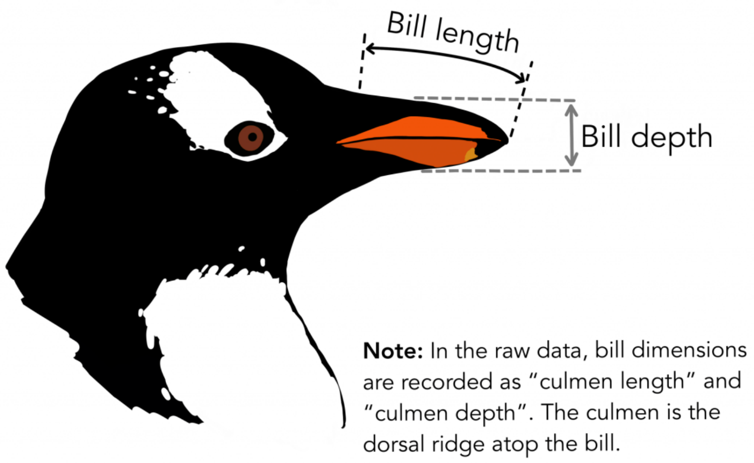
First, let's **import** all the necessary libraries that we use in this guide. Apart from the mentioned visualization libraries, we import *Pandas* and *NumPy* for data **importing** and **handling**:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline
```

The data that we use in this guide is from **PalmerPenguins** Dataset. This dataset has been recently introduced as an alternative to the famous *Iris dataset.* It is created by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER. You can obtain this dataset **here**, or via Kaggle.



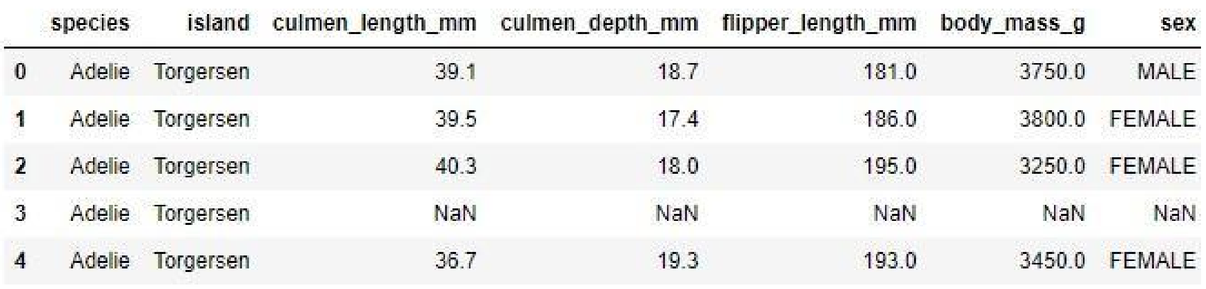
This dataset is essentially composed of two datasets, each containing the data of 344 penguins. Just like in the Iris *dataset*, there are 3 different **species** of penguins coming from 3 **islands** in the Palmer Archipelago. Also, these datasets contain culmen dimensions for each species. The culmen is the upper ridge of a bird's bill. In the simplified penguin's data, culmen length and depth are renamed as variables *culmen_length_mm* and *culmen_depth_mm*.


Bill length
Bill depth
Note: In the raw data, bill dimensions are recorded as "culmen length" and "culmen depth". The culmen is the dorsal ridge atop the bill.

Let's load the data and see what it looks like:

```
data = pd.read_csv('./data/penguins_size.csv')
data.head()
```

Note that the dataset is located in the *data* folder of our *Jupyter* notebook. Also, we are loading a simpler dataset. Here is the output:



```
   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
```

1. The Basics

Before we go into details of each step of the analysis, let's step back and define some terms that we already mentioned. First of all, what is data and in which form do we "consume" it? Data are **records** of information about an object organized into **variables** or **features**. A feature represents a certain characteristic of a record. If you have a software development background, a record is an object and a feature is the property of that object.

Data usually comes in **tabular** form, where each row represents a single record or sample and columns represent features. There are two types of features that we explore:

- **Categorical Features** – these features have discrete values that are mutually exclusive. You can observe them as enumerations.
- **Quantitative Features** – these features have continuous, numeral values. In general, they represent some sort of measurement.

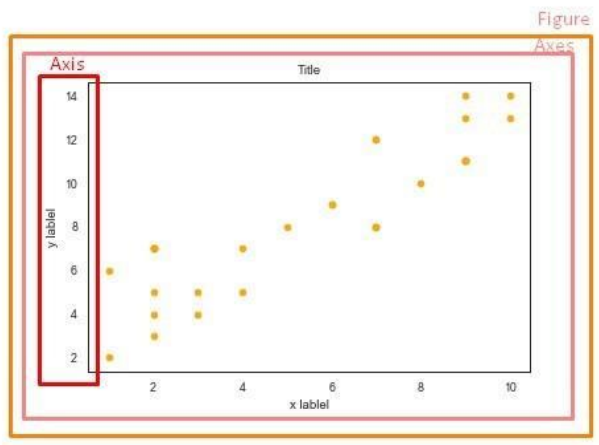
2. Data Visualization

The Python Package Index has many libraries for data visualization. In this guide, we focus on the two most popular libraries – **Matplotlib** and **Seaborn**. Matplotlib was created back in 2003 by the late John D. Hunter. His main idea was to simulate data visualization that existed in *MATLAB*. You can watch Mr. Hunter's full speech about the evolution of *Matplotlib* at the SciPy Conference **here**. As he tragically passed away in 2012, *Matplotlib* became a **community** effort making it one huge library. At the moment, we are writing this guide and it has more than *70000* lines of code. *Seaborn* is a library that is built on top of *Matplotlib* for making statistical graphics in *Python*. Apart from that, it is closely integrated with *Pandas* **data structures**.

2.1 Object Hierarchy in Matplotlib

In general, the idea behind *Matplotlib* is **twofold**. On the one hand, this library supports **general** potting actions like 'contour this 2D array'. On the other hand, it supports **specific** plotting actions, like 'make this line orange'. This makes *Matplotlib* such a cool library; you can use it in its general form most of the time and yet you are able to use specific commands when needed. This makes it a somewhat hard library for understanding and usage. That is why we start from the understanding of the **hierarchy** of the objects in this library.

At the top of the hierarchy, we can find the *pyplot* module. This module contains *Matplotlib's* "state-machine environment" and at this level, **simple** methods are used to plot data in figures and axes. One step below, we can find the first level of the **object-oriented** interface. Here the *pyplot* abstractions are used only for a few functions (i.e. figure creation). This means that the user **explicitly** creates and keeps track of the figure and axes objects. At the lowest level, using which the user has the most control, *the pyplot* module is not used at all, and only an object-oriented approach is used. In this guide, most of the time, we use the highest level of the hierarchy, meaning we rely on the *pyplot and Seaborn* modules.



Every single plot is composed of several important parts that you can find in the image above:

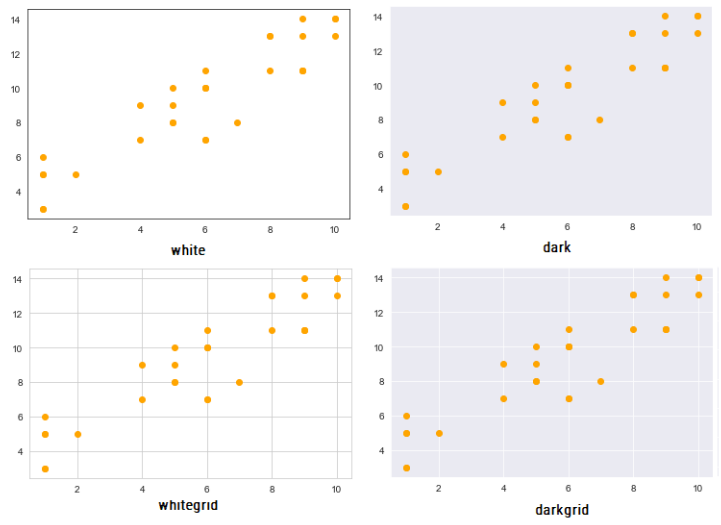
- **Figure** – this is the base object and it contains the whole figure, ie. the whole plot. It is a 'parent object' to *Axes, Canvas* and other smaller objects like *titles* and *legends* (also known as artists).
- **Axes** – when we think of the term 'plot', this is what we think of. In general, this is the part of the image with the data space and it controls data limits. It is the entry point for working with the object-oriented interface of *Matplotlib.*
- **Axis** – note the difference between Axes and Axis objects. They are the number-line-like objects and they take care of setting the graph limits and generating the ticks on each axis.
- **Artist** – this term refers to everything you see on a figure (including the *Figure* object itself).

In this guide, we will use *Matplotlib* as a base. As we mentioned, it is hard to utilize everything that exists in this library in an easy manner. That is why most of the time, we use the *Seaborn* library that is built on top of *Matplotlib.* The **combination** of these two libraries gives us big visualization power.

2.2 Seaborn Styles

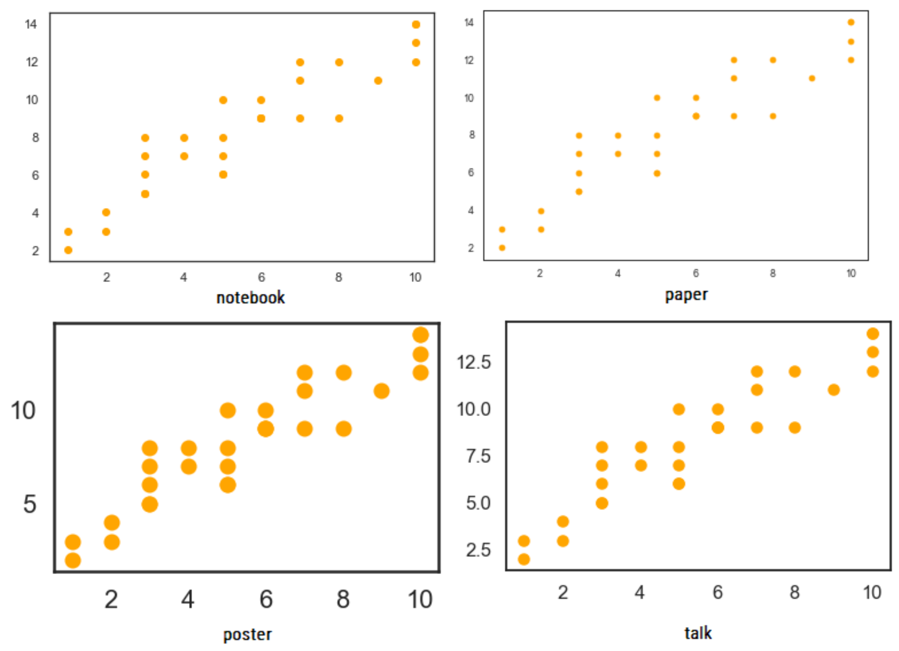
Before we dive into different types of plots that we can provide with these libraries, there is just one more thing we need to mention. When we start working with some data, it is useful to pick a **color scheme** and the **style** of the graphs. The human psyche reacts differently to different colors. Also, if you need to present data to a client, it is useful to use colors that they are comfortable with. We can set this up using *Seaborn* styles, context and color palette. In a nutshell, this is the purpose of *Seaborn* – to make our plots look dashing. To set the style of the plots, we use ***set_stlyle*** function. There are four styles that we can use: *white, dark, whitegrid, darkgrid.*

```
sb.set_stlye(name_of_the_stlye)
```



Seaborn contexts are used to define how the plot looks. These are 'pre-build packages' and they affect the size of the labels, lines, and other elements of the plot, but not the overall style, which, as we saw, is controlled by the *set_style* function. Context has four options as well: *notebook, paper, poster* and *talk.*

```
sb.set_context(name_of_the_context)
```



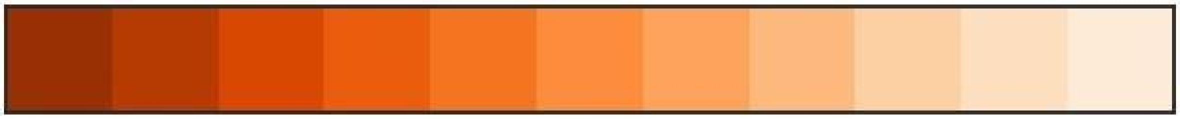
Finally, the **Seaborn palette** is used to control the colors of the charts. You can set it using the function *set_palette* and it has many options:

```
Accent, Accent_r, Blues, Blues_r, BrBG, BrBG_r, BuGn, BuGn_r, BuPu, BuPu_r, CMRmap, CMRmap_r,
Dark2, Dark2_r, GnBu, GnBu_r, Greens, Greens_r, Greys, Greys_r, OrRd, OrRd_r, Oranges,
Oranges_r, PRGn, PRGn_r, Paired, Paired_r, Pastel1, Pastel1_r, Pastel2, Pastel2_r, PiYG,
PiYG_r, PuBu, PuBuGn, PuBuGn_r, PuBu_r, PuOr, PuOr_r, PuRd, PuRd_r, Purples, Purples_r, RdBu,
RdBu_r, RdGy, RdGy_r, RdPu, RdPu_r, RdYlBu, RdYlBu_r, RdYlGn, RdYlGn_r, Reds, Reds_r, Set1,
Set1_r, Set2, Set2_r, Set3, Set3_r, Spectral, Spectral_r, Wistia, Wistia_r, YlGn, YlGnBu,
YlGnBu_r, YlGn_r, YlOrBr, YlOrBr_r, YlOrRd, YlOrRd_r, afmhot, afmhot_r, autumn, autumn_r,
binary, binary_r, bone, bone_r, brg, brg_r, bwr, bwr_r, cividis, cividis_r, cool, cool_r,
coolwarm, coolwarm_r, copper, copper_r, cubehelix, cubehelix_r, flag, flag_r, gist_earth,
gist_earth_r, gist_gray, gist_gray_r, gist_heat, gist_heat_r, gist_ncar, gist_ncar_r,
gist_rainbow, gist_rainbow_r, gist_stern, gist_stern_r, gist_yarg, gist_yarg_r, gnuplot,
gnuplot2, gnuplot2_r, gnuplot_r, gray, gray_r, hot, hot_r, hsv, hsv_r, icefire, icefire_r,
inferno, inferno_r, jet, jet_r, magma, magma_r, mako, mako_r, nipy_spectral, nipy_spectral_r,
ocean, ocean_r, pink, pink_r, plasma, plasma_r, prism, prism_r, rainbow, rainbow_r, rocket,
rocket_r, seismic, seismic_r, spring, spring_r, summer, summer_r, tab10, tab10_r, tab20,
tab20_r, tab20b, tab20b_r, tab20c, tab20c_r, terrain, terrain_r, twilight, twilight_r,
twilight_shifted, twilight_shifted_r, viridis, viridis_r, vlag, vlag_r, winter, winter_r
```

```
sb.set_palette(name_of_the_palette)
```

If you are not sure if the palette you've picked is suitable for you, you can always print colors with *palplot* function. For example:

```
sb.palplot(sb.color_palette('Oranges_r', 11))
```



Ok, now we know what the major components of one *Matplotlib* graph are, and we know how to control its style using *Seaborn*. Now, let's start plotting.

2.3 Plots

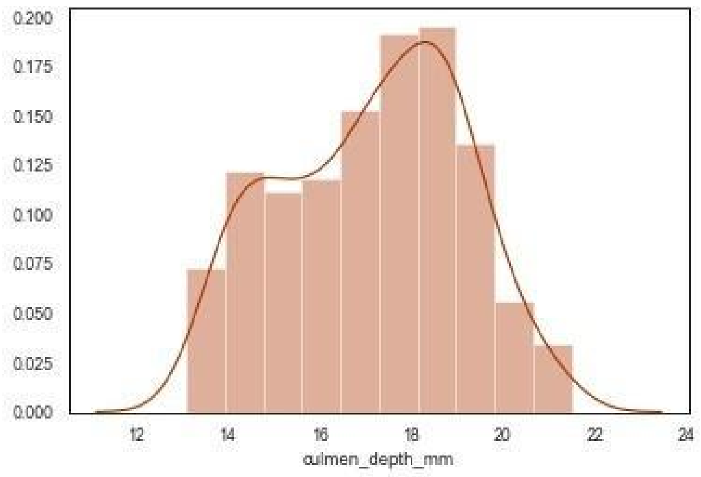
Among these libraries, we have many options for plotting data. However, in order to have a more systematic approach, we divide plots into five groups. Each group has a specific purpose.

2.3.1 Distribution Plots

This type of plot is used to show the distribution of the data, meaning it shows a list of all the possible values of the **data.** They are often used for univariate data analysis when we observe one variable and its nature. *Seaborn* also has an option for 2D distribution plots, which we can use to observe the distribution of two variables simultaneously.

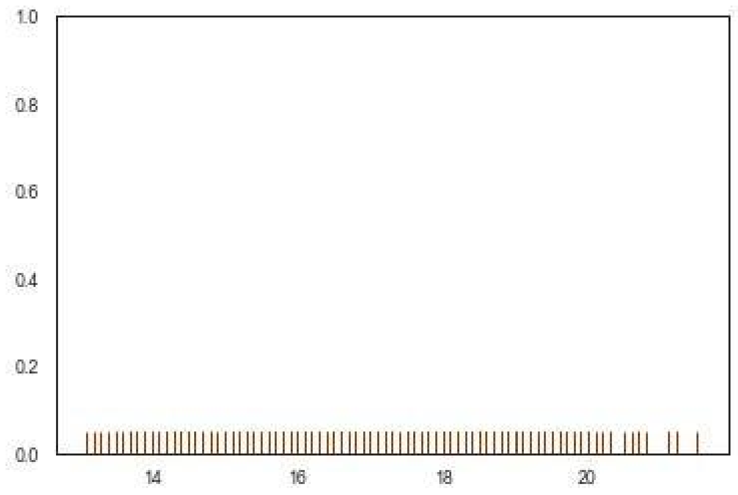
The first distribution plot that we explore is ***distplot***. It plots a univariate distribution of a variable and it is good for plotting histograms. Let's check the distribution *culmen_depth_mm* variable in the *PalmerPenguins* dataset.

```
sb.distplot(data['culmen_depth_mm'])
```



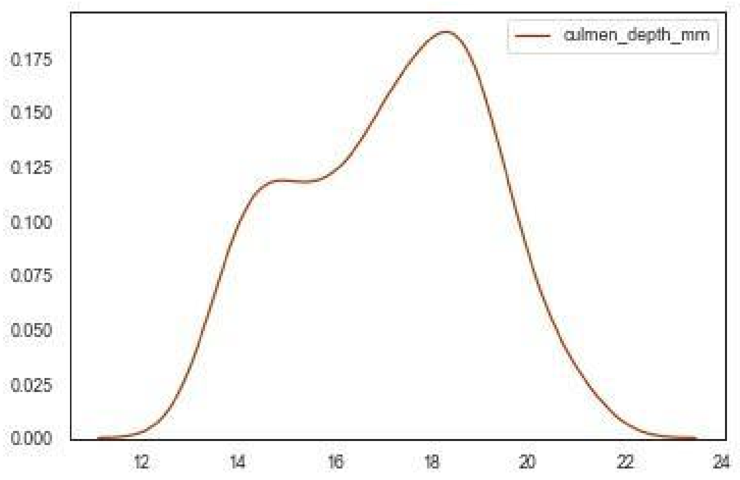
Also, we can use ***rugplot*** to plot data points in an array as sticks on an axis.

```
sb.rugplot(data['culmen_depth_mm'])
```



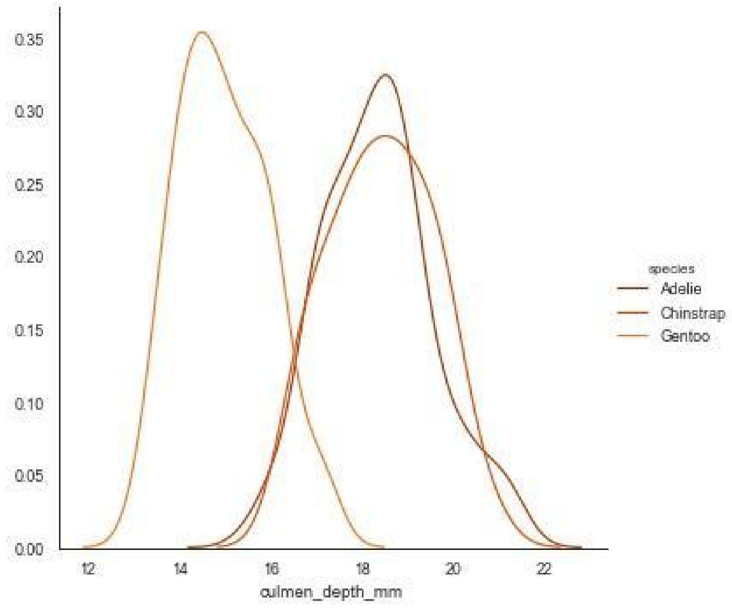
Using *Seaborn*, we can also plot the KDE plot using the ***kdeplot*** function. The kernel density estimation (or **KDE**) is a way to estimate the probability density function of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination. The KDE plot is already included in the *distplot*, but we may want to use it separately

```
sb.kdeplot(data['culmen_depth_mm'])
```



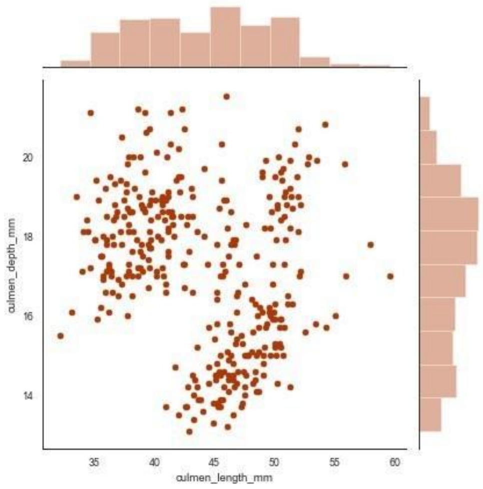
Apart from that, we can use *FacetGrid* from *Seaborn* for plotting conditional relationships, for which we may pick the KDE plot. For example, let's plot the relationship between *species* and *culmen_depth_mm* variables from the dataset:

```
sb.FacetGrid(data,hue="species",height=5)\
              .map(sb.kdeplot,"culmen_depth_mm")\
              .add_legend()
plt.ioff()
```



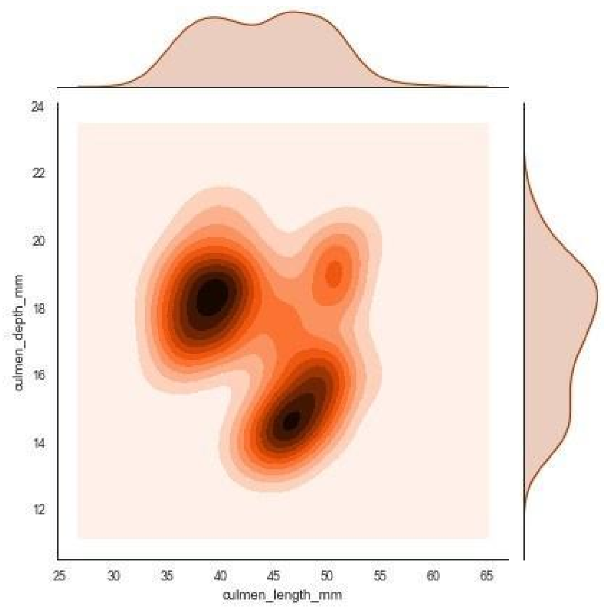
When we talk about distributions and relationships, we need to mention ***Jointplot***. This type of plot is used to visualize and analyze the relationship between two variables but also to display the individual distribution of each variable on the same plot.

```
sb.jointplot(x="culmen_length_mm",y="culmen_depth_mm", data=data)
```



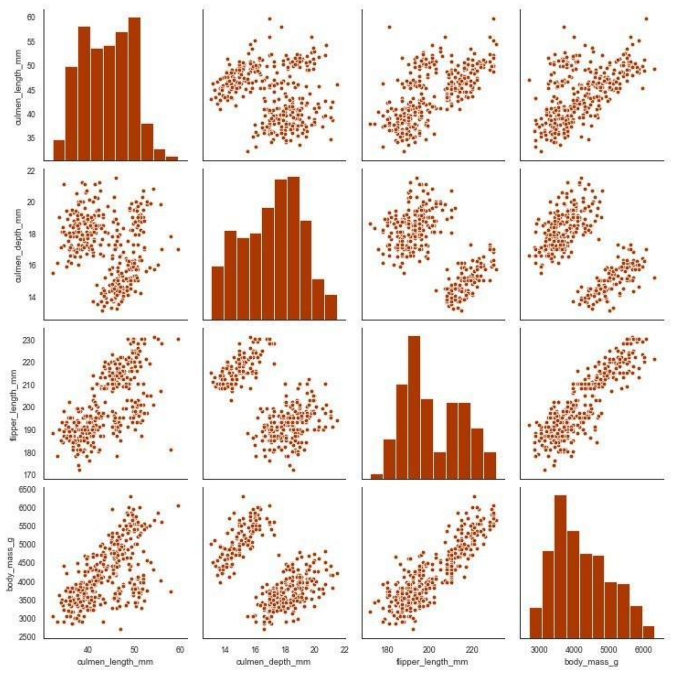
This can also be extended with the KDE plot; all we have to do is use the *kind* parameter of the *jointplot* function.

```
sb.jointplot(x="culmen_length_mm",y="culmen_depth_mm", data=data,
kind='kde')
```



In the end, we may want to print the relationships of all variables. This can be done with the ***pairplot*** function. It is one very useful trick for exploratory data analysis.

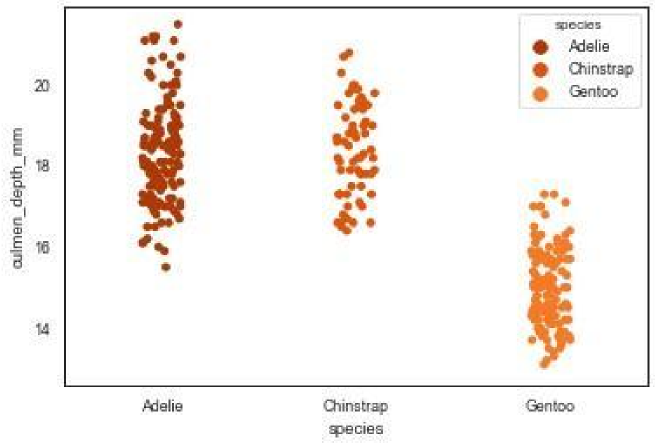
```
sb.pairplot(data)
```



2.4 Continuous & Categorical Variables Relationships

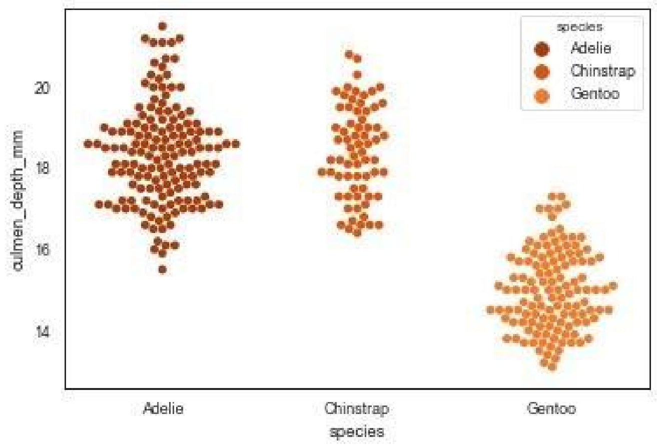
This group of plots is all about the relationship between continuous and categorical variables. For example, if we want to get more information about the distribution of the *cullmen_depth_mm* variable in relationship with the *species* variable, we would use one of these plots. Let's start with the stripplot function. This *Seaborn* method draws a scatterplot with one categorical variable.

```
sb.stripplot(x="species",y="culmen_depth_mm",data=data,hue="species")
```



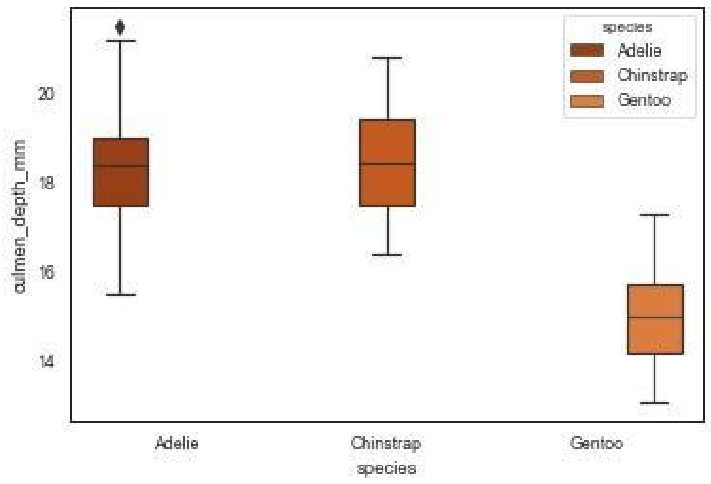
Similar to the *stripplot* function, *swarmplot (*also called *beeswarm)* draws scatterplot but without overlapping points. It gives a better representation of the distribution of values.

```
sb.swarmplot(x="species",y="culmen_depth_mm",data=data,hue="species")
```



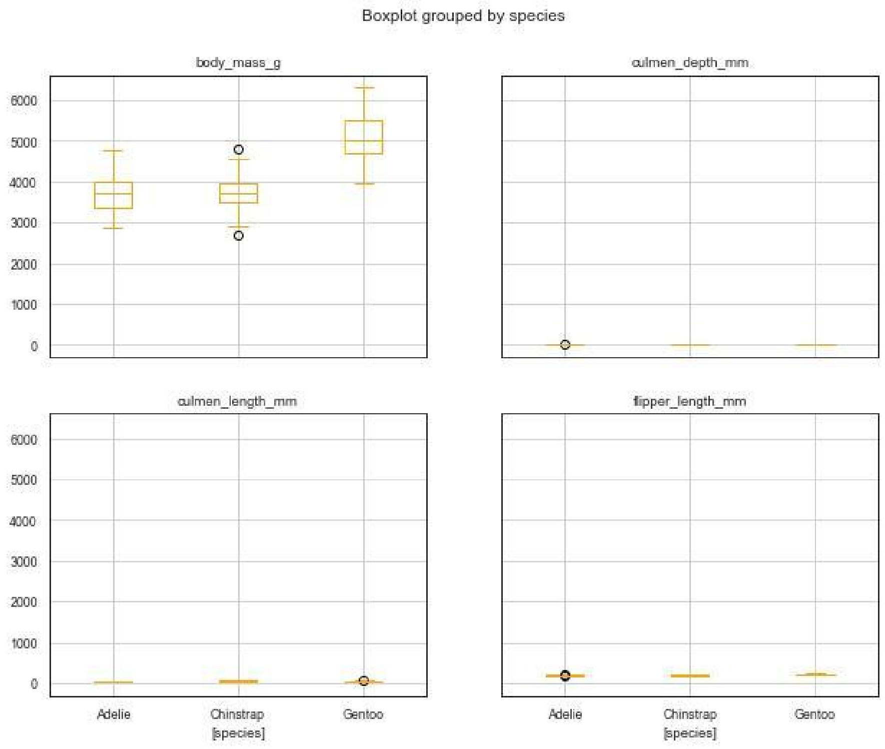
Probably the most commonly used plot for this purpose is the ***boxplot***. This plot draws five important distribution points, ie. it gives a statistical summary of the variable. The minimum and the maximum, 1st quartile (25th percent), the median and 3rd quartile (75th percent) of the variable are included in the graph. Also, using this plot, you can detect outliers. The main problem of this plot is that it has a tendency to hide irregular distributions.

```
sb.boxplot(x="species",y="culmen_depth_mm",data=data,hue="species")
```



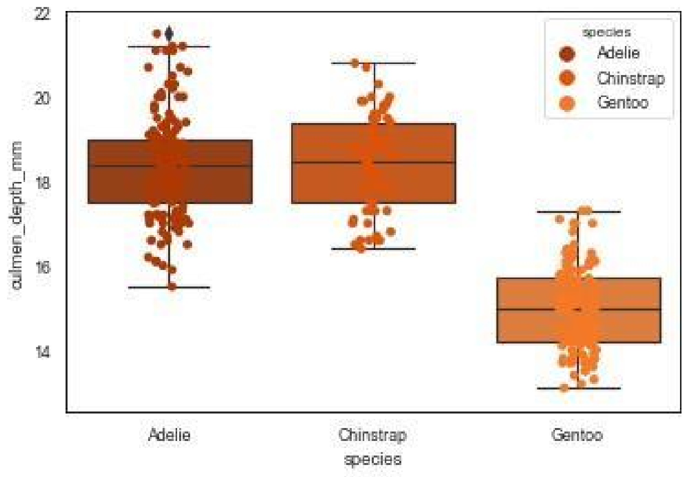
Sometimes it is useful to use *boxplot* for each variable; something like this:

```
data.boxplot(by="species",figsize=(10,8), color='orange')
```



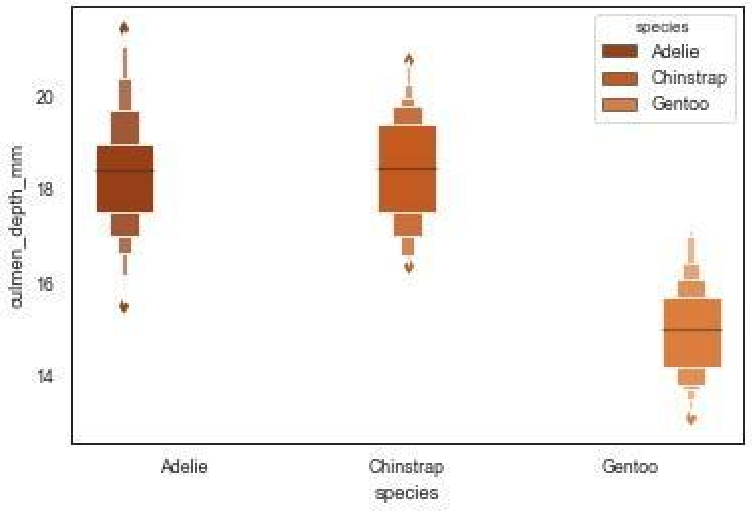
Another neat trick is to use the *boxplot* with a stripplot.

```
box_strip_combo_fig=sb.boxplot(x="species",y="culmen_depth_mm",data=data)
box_strip_combo_fig=sb.stripplot(x="species",y="culmen_depth_mm",data=data,
hue="species")
```



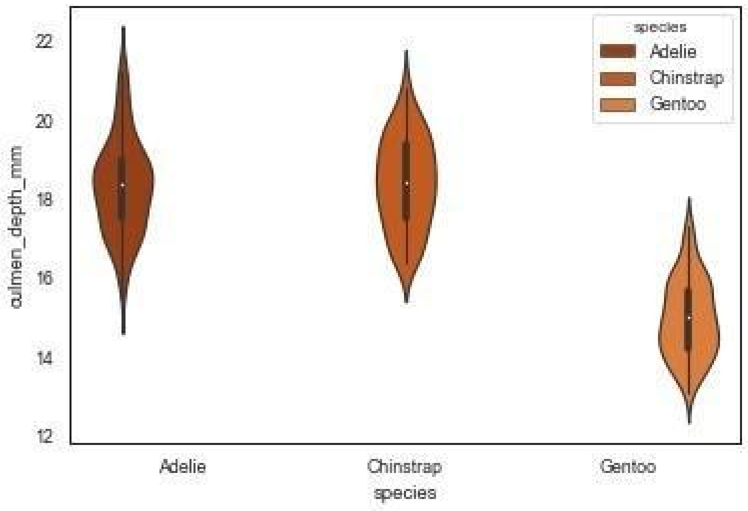
In order to use the *boxplot* benefits on large datasets, sometimes we can use the ***boxenplot***, which is basically an extended *boxplot.*

```
sb.boxenplot(x="species",y="culmen_depth_mm",data=data,hue="species")
```



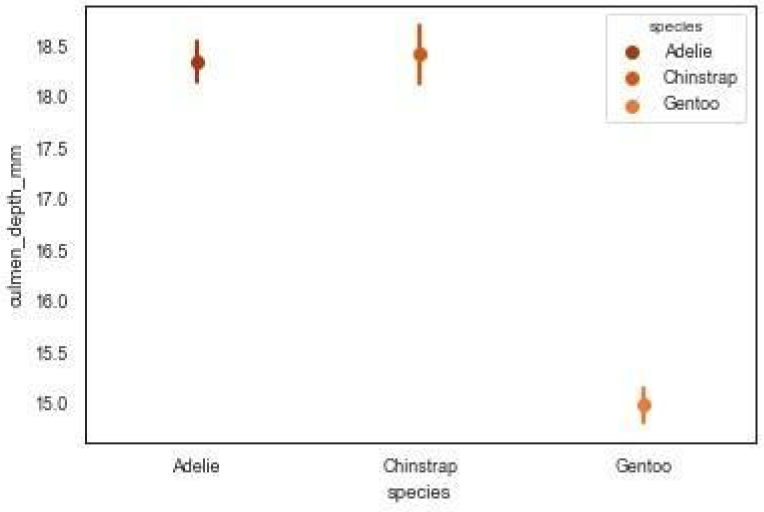
A good alternative to *boxplot* is ***violinplot.*** It plots the distribution of the variable along with its probability distribution. The interquartile range, the 95% confidence intervals, and the median of the variable is displayed in this chart. The biggest flaw of this plot is that it tends to hide how values in the variable itself are distributed.

```
sb.violinplot(x="species",y="culmen_depth_mm",data=data,hue="species")
```



In some special cases, we may want to use ***pointplot*****.** This plot gives one point that represents the corresponding variable. It is useful for comparing continuous numerical variables.

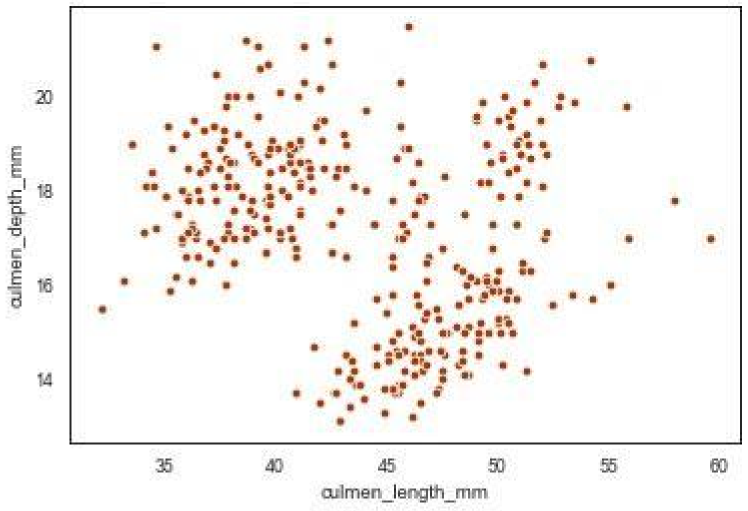
```
sb.pointplot(x="species",y="culmen_depth_mm",data=data,hue="species")
```



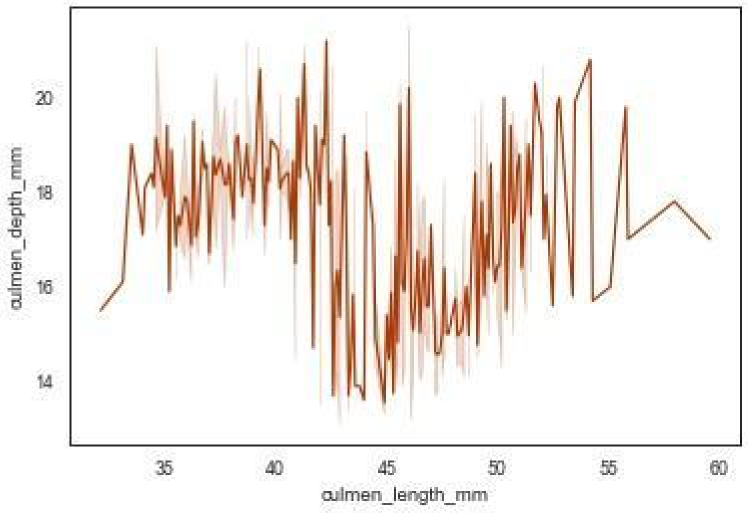
2.5 Continuous Variables Relationships

During exploratory data analysis, sometimes we want to visualize relationships of two continuous variables. We can do so with two plots: ***scatterplot*** and ***lineplot***. Let's use them to represent the relationship between *culmen_length_mm* and *culment_depth_mm* features.

```
sb.scatterplot(x="culmen_length_mm",y="culmen_depth_mm", data=data)
```



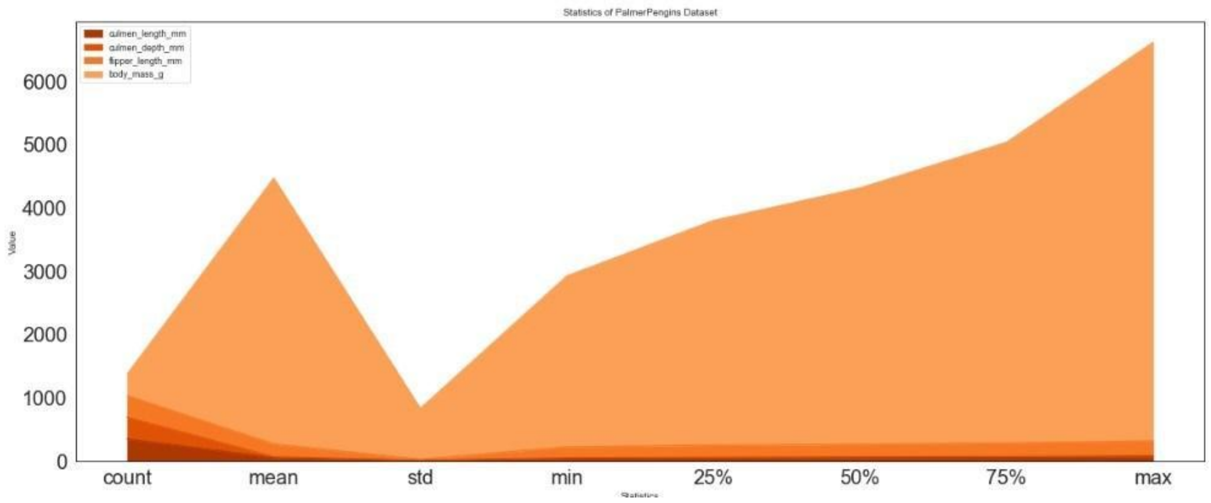
```
sb.lineplot(x="culmen_length_mm",y="culmen_depth_mm", data=data)
```



2.6 Statistical models

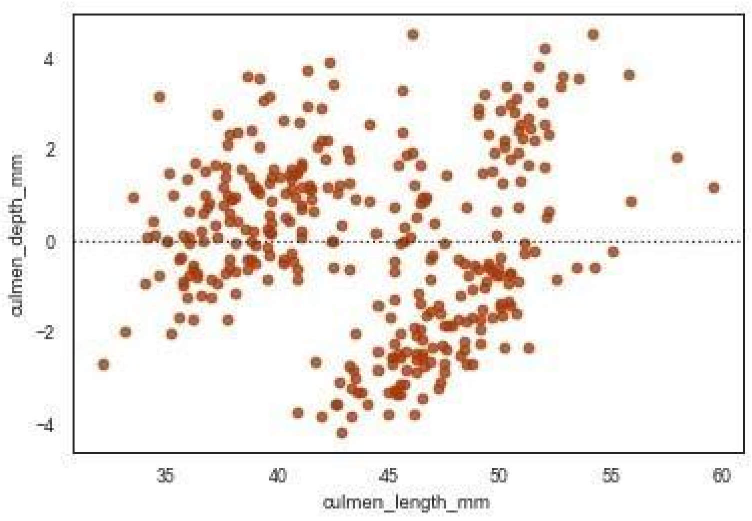
In general, we use these plots to display the statistical nature of the data. We can do so on the complete dataset just to get familiar with it, or we can use ***residplot*** and ***lmplot.*** We utilize *Pandas* and *Seaborn* interoperability for the first task:

```
data.describe().plot(kind="area",fontsize=20, figsize=(20,8),table=False)
plt.xlabel('Statistics')
plt.ylabel('Value')
plt.title("Statistics of PalmerPengins Dataset")
```



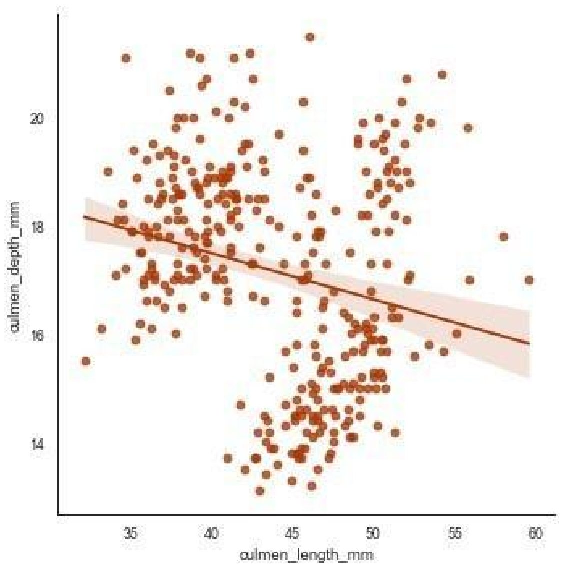
The *residplot* draws residuals of linear regression, meaning it displays how far each data point was off the linear regression fit.

```
sb.residplot(x="culmen_length_mm",y="culmen_depth_mm", data=data)
```



The *lmplot* does the same thing, but it also displays confidence intervals. It has many parameters that can help you customize this plot. One of the most useful ones is the *logistic* parameter. When set to *true*, this parameter will indicate that the *y*-variable is binary and will use a Logistic Regression model.

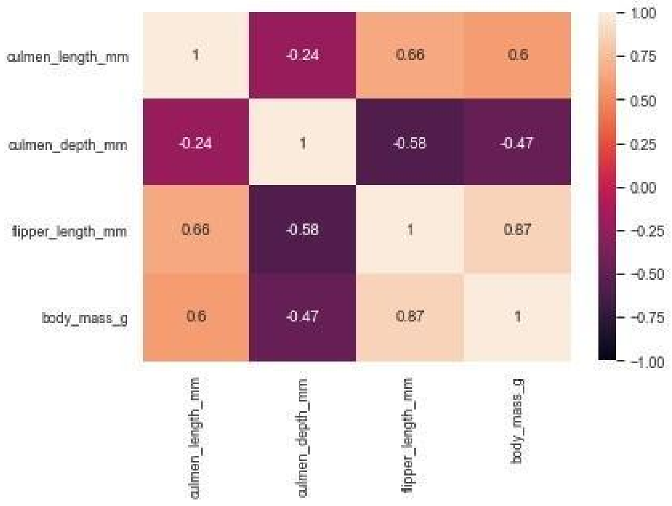
```
sb.lmplot(x="culmen_length_mm",y="culmen_depth_mm", data=data)
```



2.7 Heatmaps

The final type of plot that we investigate is the so-called ***heatmap.*** The heatmap displays any type of matrix by painting higher values with more intense color. It is often used for correlation analysis when we need to decide which features from the dataset we want to pick for machine learning.

```
sb.heatmap(data.corr(),vmin=-1,vmax=1,annot=True)
```



3. Exploratory Data Analysis

When working on one machine learning model, the first step is data analysis or to be more specific **exploratory data analysis**. In this step, we are trying to figure out the nature of each feature that exists in our data, as well as its distribution and relation to other features. We aim to clean up all the unnecessary information that could potentially confuse our algorithm.

After this step, we run an algorithm and evaluate it. If we are not satisfied with the results, we go back to data analysis or apply a different algorithm. As you can see, exploratory data analysis is a **crucial** element of the machine learning process. Without it, our smart algorithms would give too optimistic results (**overfitting**) or plain wrong results.

3.1 Univariate Analysis

In this first step of data analysis, we are trying to determine **the nature** of each feature. Sometimes categorical variables can have the wrong type, or some quantitative data is out of scale. These things are investigated during univariate analysis. We observe each feature individually, but we try to grasp the image of the dataset as a whole. A good approach is to get the shape of the dataset and observe the number of samples. We can also print several samples and try to get some information from them.

```
print(data.shape)
data.head()
```

```
(344, 7)
```

Pandas function *head* gives us the first five records from the loaded data. We can show more data by giving any number as a parameter. Here is the output of the code snippet from above:

	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

From this output, we can see that we have 344 samples or records. Apart from that, we can notice that features have values on **different scales**. For example, record 0 has the *culmen_depth_mm* feature value of 18.7, while feature *body_mass_g* has the value 3750. If we are working on a machine learning solution, this is a situation we need to **address** and put these features on the same scale. If we don't do that before we start the training process, the machine learning model will "think" that the *body_mass_g* feature is more important than the *culmen_depth_mm* feature. We will see how this is handled in the next chapter; however the important bit is that we detected that here.

3.1.1 Remove unnecessary features

Another thing we can notice is that some features are not carrying information and are **useless**. For example, we can conclude that the *sex* feature is not working for us.
So we can remove it:

```
drop_features = {"sex"}
data = data.drop(columns=drop_features)
data.head()
```

	species	island	culmen_length_mm	culmen_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

3.1.2 Handling Types of Features

The next thing we can do in this analysis is check the datatypes of each feature. Again, we use *Pandas* for this:

```
data.dtypes
```

The output of this call looks like this:

```
species               object
island                object
culmen_length_mm     float64
culmen_depth_mm      float64
flipper_length_mm    float64
body_mass_g          float64
dtype: object
```

The important thing to notice here is that we have **categorical** variables like *species and island*. However, these features in the loaded data have type *objects*, meaning machine learning and deep learning models will observe them as **quantitative** features. We have to change this and change their **type**. We can do that like this:

```
categorical_features = {"species", "island"}
for feature in categorical_features:
data[feature] = data[feature].astype("category")
```

Now, when we call *data.dtypes* we get this output:

```
species                  category
island                   category
culmen_length_mm          float64
culmen_depth_mm           float64
flipper_length_mm         float64
body_mass_g               float64
dtype: object
```

Great, now our categorical features really have a type *category.*

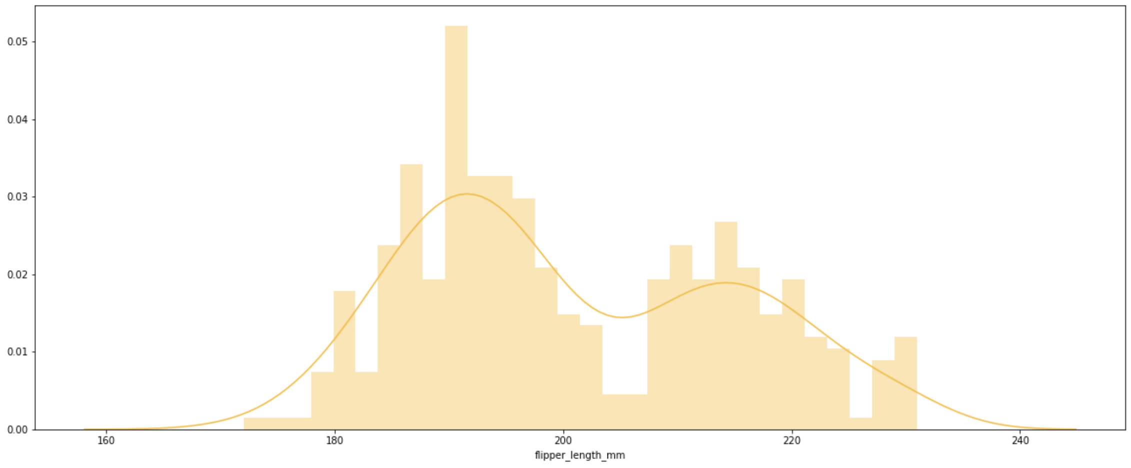
3.2 Distribution

The important characteristic of features we need to explore is **distribution**. This is especially important for features with **quantitative** nature. Mathematically speaking, the distribution of a feature is a function showing all the possible values (or intervals) of the data and their **frequency** of occurrence. When we talk about the distribution of categorical data, we can see the number of samples in each category. On the other hand, when we observe a distribution of numerical data, values are ordered from smallest to largest and sliced into reasonably sized groups.

The distribution of the data is usually represented with a **histogram**. Basically, we split the complete range of possible values into intervals and count how many samples fall into each interval. To do this in *Python*, we use the *Seaborn* module:

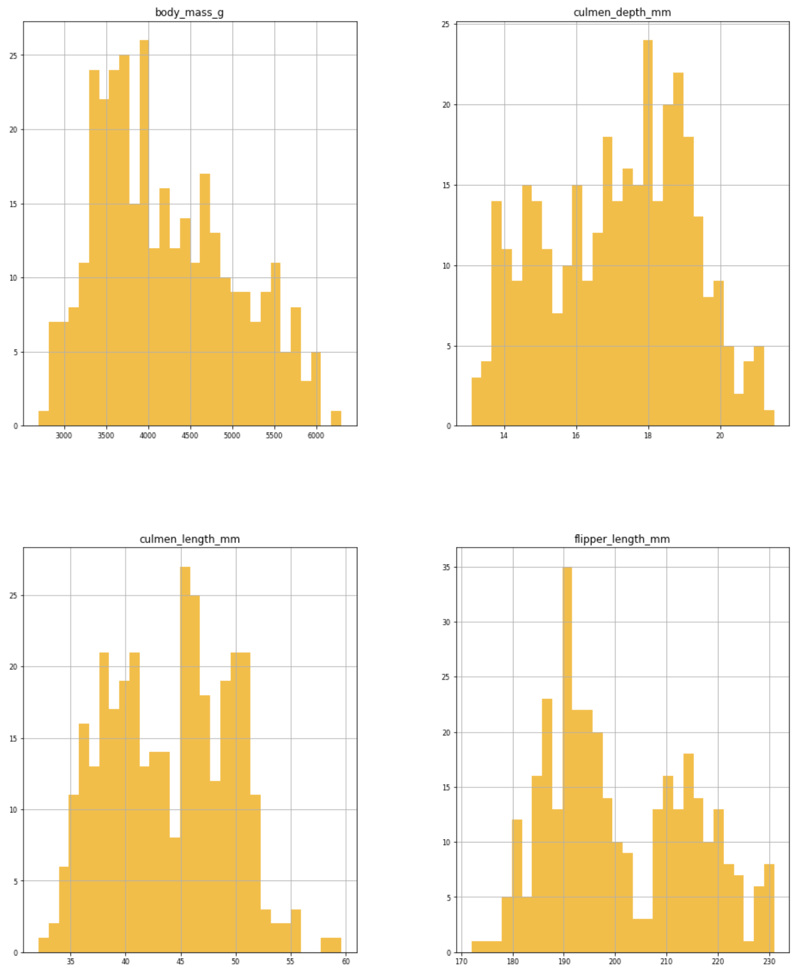
```
plt.figure(figsize=(20, 8))
sb.distplot(data['flipper_length_mm'], color=ORANGE, bins=30,
hist_kws={'alpha': 0.4});
```

In this particular case, we are using the *flipper_length_mm* feature, with an interval of 30 samples. Here is what that histogram looks like:



Apart from this, we can do this for every feature in the dataset:

```
numerical_fetures = data.select_dtypes(include = ['float64', 'int64'])
numerical_fetures.hist(figsize=(16, 20), color = ORANGE, bins=30, 
xlabelsize=8, ylabelsize=8)
```



When we observe the distribution of the data, we want to describe certain characteristics like its center, shape, spread, amount of variability, etc. To do so, we use several measures.

3.2.1 Measures of Center

For describing the center of the distribution we use:

- **Mean** – this value is the average of a set of samples. This means that it represents the sum of the values of the feature for each sample divided by the number of samples.
- **Mode** –the mode is identified as the “peak” of the histogram. Distributions can have one mode (unimodal distributions) or two modes (bimodal distributions).
- **Median** – this value represents the midpoint of the distribution. It is such a number that half of the observations fall above, and half fall below.

To get these values, we can use *Panda’s* functions *mean*, *mode* and *median*:

```
data.median()
```

```
culmen_length_mm       44.45
culmen_depth_mm        17.30
flipper_length_mm     197.00
body_mass_g          4050.00
dtype: float64
```

```
data.mode()
```

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g
0	Adelie	Biscoe	41.1	17.0	190.0	3800.0

```
data.mean()
```

```
culmen_length_mm       43.921930
culmen_depth_mm        17.151170
flipper_length_mm     200.915205
body_mass_g          4201.754386
dtype: float64
```

We can apply these functions for each feature.

3.2.2 Measures of Spread

To describe the spread, we most commonly use the following measures:

- **Range** – this measure represents the distance between the smallest data point and the largest one.
- **Inter-quartile range (IQR)** – while range covers the whole data, IQR indicates where the middle 50 percent of data is located. When we look for this value, we first look for the median M, since it splits data into half. Then, we locate the median of the lower end of the data (denoted as Q1) and the median of the higher end of the data (denoted as Q3). The data between Q1 and Q3 is the IQR.
- **Standard deviation** – this measure gives the average distance between data points and the mean. Essentially, it quantifies the spread of a distribution.

To get all these measures, we use the function of Pandas. This function will give back the rest of the measures that we used for the center as well. Here is what that looks like.

```
data['flipper_length_mm'].describe()
```

```
count    342.000000
mean     200.915205
std       14.061714
min      172.000000
25%      190.000000
50%      197.000000
75%      213.000000
max      231.000000
Name: flipper_length_mm, dtype: float64
```

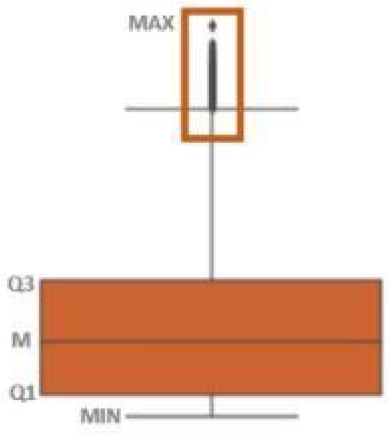
Similar to the previous functions, we can call it over the whole dataset and get these statistics for every feature:

	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g
count	342.000000	342.000000	342.000000	342.000000
mean	43.921930	17.151170	200.915205	4201.754386
std	5.459584	1.974793	14.061714	801.954536
min	32.100000	13.100000	172.000000	2700.000000
25%	39.225000	15.600000	190.000000	3550.000000
50%	44.450000	17.300000	197.000000	4050.000000
75%	48.500000	18.700000	213.000000	4750.000000
max	59.600000	21.500000	231.000000	6300.000000

3.3 Detecting and Handling Outliers

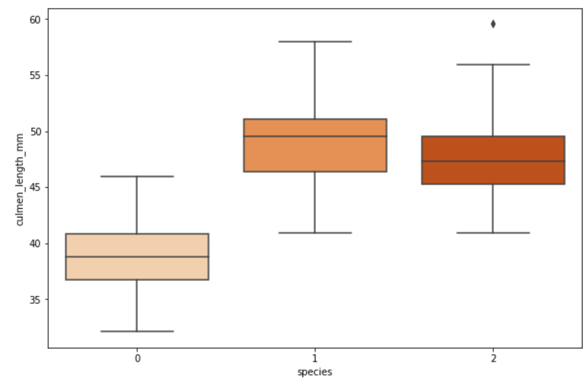
Outliers are values that deviate from the whole **distribution** of the data. Sometimes these values are mistakes and wrong measurements and should be removed from datasets, but sometimes they are valuable **edge-case** information. This means that sometimes we want to leave these values in the dataset since they may carry some important information, while other times we want to remove those samples because of the wrong information.

In a nutshell, we can use the **Interquartile range** to detect these points. As we mentioned in the previous chapter, the Inter-quartile range or *IQR* indicates where 50 percent of data is located. When we look for this value, we first look for the median since it splits data in half. Then we locate the **median** of the lower end of the data (*Q1*) and the median of the higher end of the data (*Q3*). The data between *Q1* and *Q3* is the *IQR*. Outliers are defined as samples that fall below *Q1 – 1.5(IQR)* or above *Q3 + 1.5(IQR)*. We can do this using a **boxplot**. The purpose of the boxplot is to visualize the distribution. In essence, it includes important points: max value, min value, median, and two IQR points (Q1, Q3). Here is what one example of a boxplot looks like:

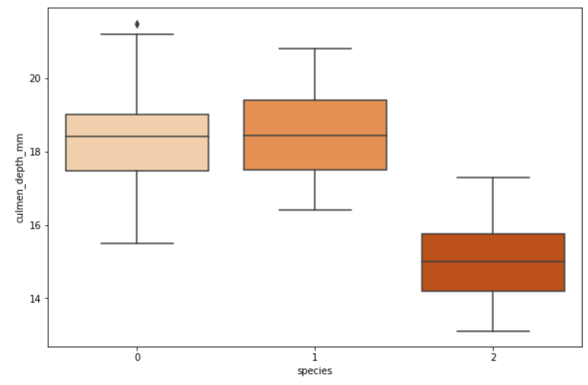


Let's apply it to *PalmerPenguins* dataset:

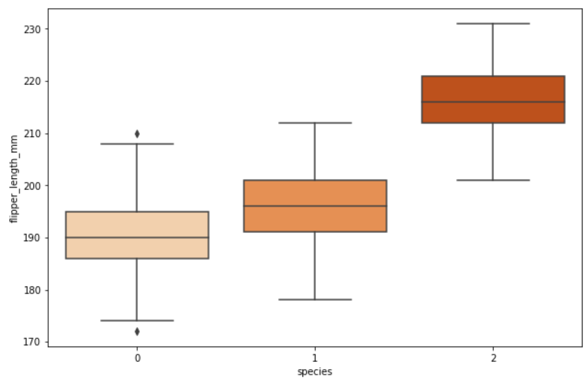
```
fig, axes = plt.subplots(nrows=4,ncols=1)
fig.set_size_inches(10, 30)
sb.boxplot(data=data,y="culmen_length_mm",x="species",orient="v",ax=axes[0]
, palette="Oranges")
sb.boxplot(data=data,y="culmen_depth_mm",x="species",orient="v",ax=axes[1],
palette="Oranges")
sb.boxplot(data=data,y="flipper_length_mm",x="species",orient="v",ax=axes[2
], palette="Oranges")
sb.boxplot(data=data,y="body_mass_g",x="species",orient="v",ax=axes[3],
palette="Oranges")
```



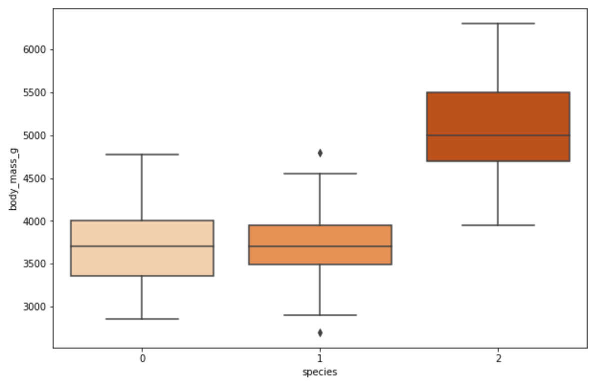
culmen_length_mm
60
55
50
45
40
35
0
1
2
species



culmen_depth_mm
20
18
16
14
0
1
2
species



flipper_length_mm
230
220
210
200
190
180
170
0
1
2
species

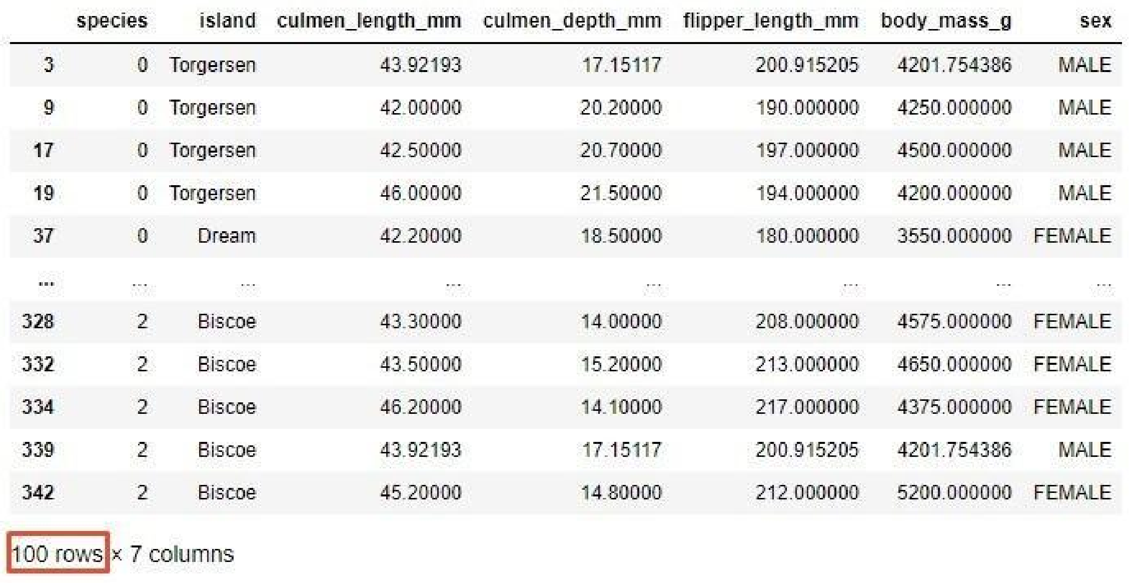


body_mass_g
6000
5500
5000
4500
4000
3500
3000
0
1
2
species

The other way for detecting and removing outliers would be by using standard deviation.

```
factor = 2
upper_lim = data['culmen_length_mm'].mean () + data['culmen_length_mm'].std
() * factor
lower_lim = data['culmen_length_mm'].mean () - data['culmen_length_mm'].std
() * factor

no_outliers = data[(data['culmen_length_mm'] < upper_lim) &
(data['culmen_length_mm'] > lower_lim)]
no_outliers
```



	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
3	0	Torgersen	43.92193	17.15117	200.915205	4201.754386	MALE
9	0	Torgersen	42.00000	20.20000	190.000000	4250.000000	MALE
17	0	Torgersen	42.50000	20.70000	197.000000	4500.000000	MALE
19	0	Torgersen	46.00000	21.50000	194.000000	4200.000000	MALE
37	0	Dream	42.20000	18.50000	180.000000	3550.000000	FEMALE
...	...	...	...	...	...	...	...
328	2	Biscoe	43.30000	14.00000	208.000000	4575.000000	FEMALE
332	2	Biscoe	43.50000	15.20000	213.000000	4650.000000	FEMALE
334	2	Biscoe	46.20000	14.10000	217.000000	4375.000000	FEMALE
339	2	Biscoe	43.92193	17.15117	200.915205	4201.754386	MALE
342	2	Biscoe	45.20000	14.80000	212.000000	5200.000000	FEMALE

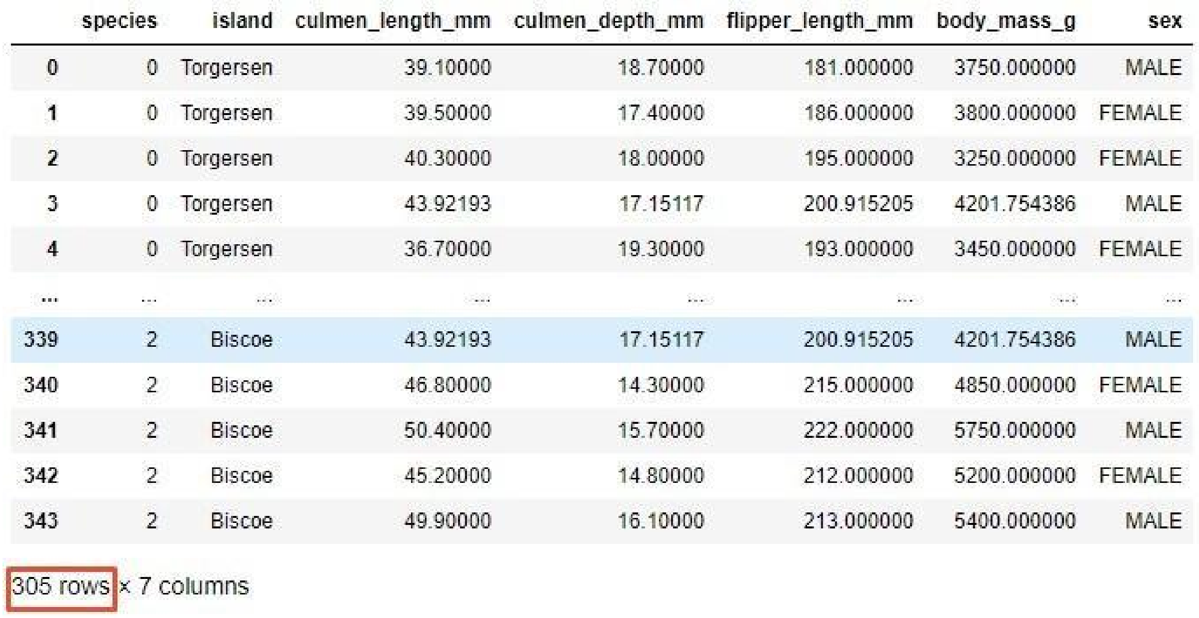
100 rows × 7 columns

Note that we now have only 100 samples left after this operation. Here we need to define the **factor** by which we multiply the standard deviation. Usually, we use values between 2 and 4 for this purpose.

Finally, we can use a method to detect outliers, which is to use **percentiles**. We can assume a certain percentage of the value from the top or the bottom as an outlier. Again, the value for the percentiles we use as outliers border depends on the distribution of the data. Here is what we can do on the *PalmerPenguins* dataset:

```
upper_lim = data['culmen_length_mm'].quantile(.95)
lower_lim = data['culmen_length_mm'].quantile(.05)

no_outliers = data[(data['culmen_length_mm'] < upper_lim) &
(data['culmen_length_mm'] > lower_lim)]
no_outliers
```



	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	0	Torgersen	39.10000	18.70000	181.000000	3750.000000	MALE
1	0	Torgersen	39.50000	17.40000	186.000000	3800.000000	FEMALE
2	0	Torgersen	40.30000	18.00000	195.000000	3250.000000	FEMALE
3	0	Torgersen	43.92193	17.15117	200.915205	4201.754386	MALE
4	0	Torgersen	36.70000	19.30000	193.000000	3450.000000	FEMALE
...	...	...	...	...	...	...	...
339	2	Biscoe	43.92193	17.15117	200.915205	4201.754386	MALE
340	2	Biscoe	46.80000	14.30000	215.000000	4850.000000	FEMALE
341	2	Biscoe	50.40000	15.70000	222.000000	5750.000000	MALE
342	2	Biscoe	45.20000	14.80000	212.000000	5200.000000	FEMALE
343	2	Biscoe	49.90000	16.10000	213.000000	5400.000000	MALE

305 rows × 7 columns

After this operation, we have 305 samples in our dataset. With this approach, we need to be extremely careful since it reduces the dataset size and highly depends on the data distribution.

3.4 Correlation Matrix

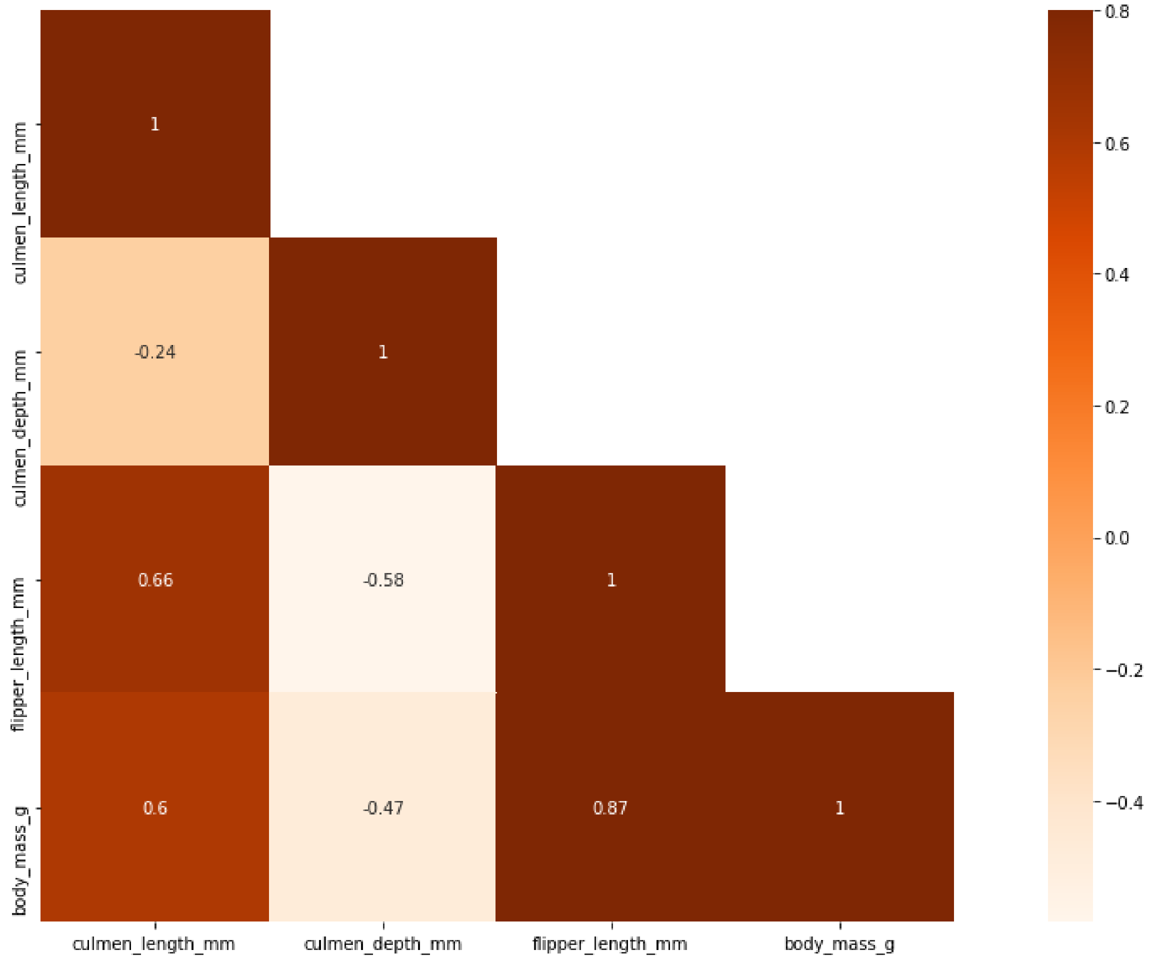
So far, we observed features individually and the relationship between quantitative and categorical features. However, in order to prepare data for our fancy algorithms, it is always important to analyze the **relationships** of one quantitative feature to another. The goal of this section of the analysis is to detect features that are affecting output too much, or features that are carrying the **same** information. In general, this is usually happening when explored features have a **linear relationship**. Meaning, we can model this relationship in the form $y = kx + n$, where y and x are explored features (variables), while k and n are scalar values.

There are two types of linear relationships, positive and negative. **Positive linear relationship** means that an increase in one of the features results in an increase in the other feature. On the other hand, a **negative linear relationship** means that an increase in one of the features results in a decrease in the other feature. Another characteristic is the **strength** of this relationship. Basically, if data points are far away from the modeled function, the relationship is weaker. From the image above, we can determine that the relationship in the first graph is stronger than in the other one (but it is not that obvious).

In order to determine what kind of a relationship we have, we use visualization tools like **Scatterplot** and **Correlation Matrix**. A scatterplot is a useful tool when it comes to displaying features. The **Correlation matrix** consists of correlation coefficients for each feature relationship. The **correlation coefficient** is a measure that gives us information about the **strength** and **direction** of a linear relationship between two quantitative features. This coefficient can have values from the range -1 to 1. If this coefficient is negative, the examined linear relationship is negative; otherwise, it is positive. If the value is closer to -1 or 1, the relationship is stronger. To get this information, we use a combination of *Pandas* and *Seaborn* modules. Here is how:

```
corrMatt = data.corr()
mask = np.array(corrMatt)
mask[np.tril_indices_from(mask)] = False
fig,ax= plt.subplots()
fig.set_size_inches(20,10)
sb.heatmap(corrMatt, cmap="Oranges", mask=mask,vmax=.8,
square=True,annot=True)
```

The output of this code snippet, looks like this:



As you can see, There is a high correlation between *feature_length_mm* and *body_mass_g*. We may choose to **remove** some of those features.

4. Feature Engineering

Feature engineering is not there just to optimize models. Sometimes, we need to apply these techniques, so our data is **compatible** with the machine learning algorithm. Machine learning algorithms sometimes expect data **formatted** in a certain way, and that is where feature engineering can help us. Apart from that, it is important to note that data scientists and engineers spend most of their time on data preprocessing. That is why it is important to master these techniques.

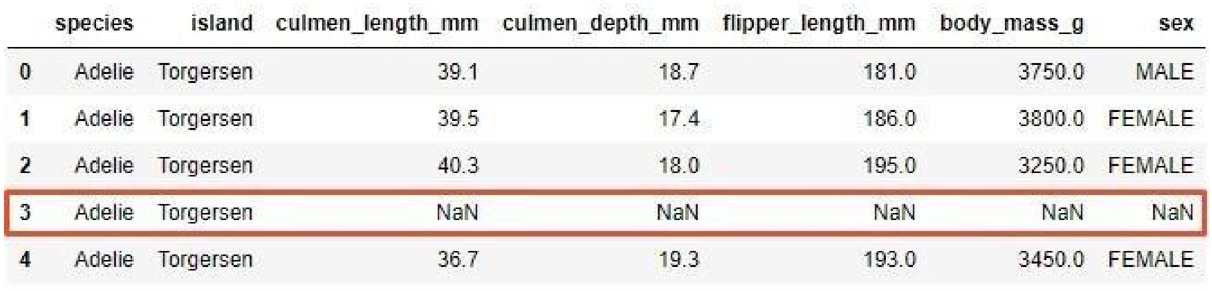
4.1 Imputation

Data that we get from clients can come in all shapes and forms. Often it is **sparse**, meaning some samples may **miss** data for some features. We need to detect those instances and remove those samples or replace empty values with something. Depending on the rest of the dataset, we may apply different strategies for replacing those missing values. For example, we may fill these empty slots with average feature value, or maximal feature value. However, let's first detect missing data. For that we can use *Pandas*:

```
print(data.isnull().sum())
```

```
species                0
island                 0
culmen_length_mm       2
culmen_depth_mm        2
flipper_length_mm      2
body_mass_g            2
sex                   10
```

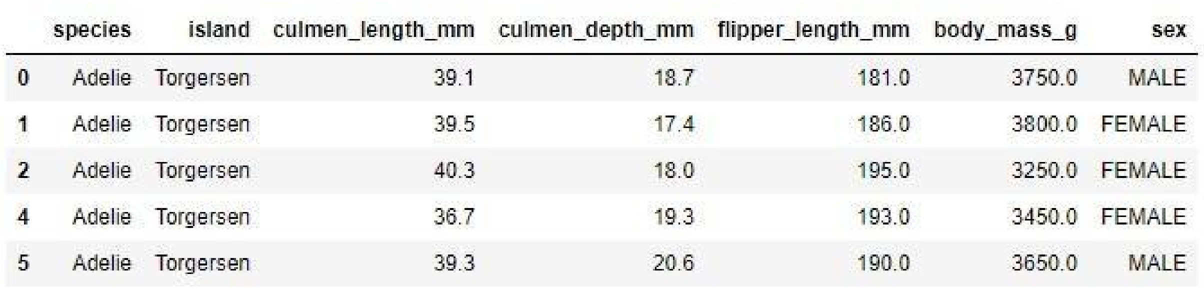
This means that there are instances in our dataset that are **missing** values in some of the features. There are two instances that are missing the *culmen_length_mm* feature value and 10 instances that are missing the *sex* feature. We were able to see that even in the first couple of samples (NaN means Not a Number, meaning missing value):



	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

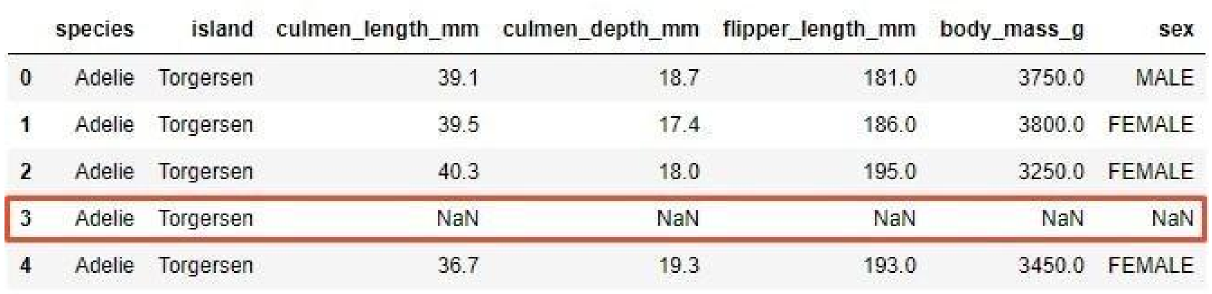
The easiest deal with missing values is to **drop** samples with missing values from the dataset; in fact, some machine learning platforms automatically do that for you. However, this may reduce the performance of the dataset because of the reduced dataset. The easy way to do it is, again, using *Pandas*:

```
data = pd.read_csv('./data/penguins_size.csv')
data = data.dropna()
data.head()
```



	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

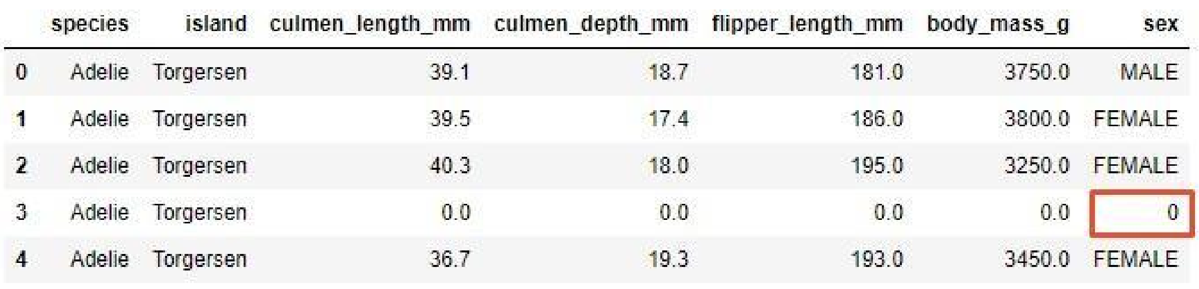
Note that the third sample with missing values is removed from the dataset. This is not optimal, but sometimes it is necessary since most of the machine learning algorithms don't work with sparse data. The other way is to use imputation, meaning to **replace** missing values. To do so, we can pick a value, or use the **mean** value of the feature or an **average** value of the feature, etc. Still, we need to be careful. Observe missing value at the row with index 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

If we just replace it with simple value, we apply the same value for categorical and for numerical features:

```
data = data.fillna(0)
```



	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	0.0	0.0	0.0	0.0	0
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

This is not good. So, here is the proper way. We detected missing data in numerical features *culmen_length_mm*, *culmen_depth_mm*, *flipper_length_mm* and *body_mass_g*. For the imputation value of these features, we will use the **mean** value of the feature. For the categorical feature '*sex*', we use the **most frequent value**. Here is how we do it:

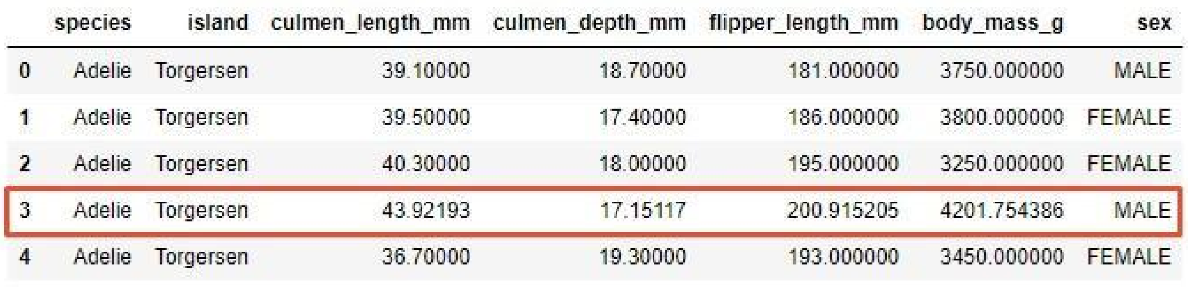
```
data = pd.read_csv('./data/penguins_size.csv')

data['culmen_length_mm'].fillna((data['culmen_length_mm'].mean()),
inplace=True)
data['culmen_depth_mm'].fillna((data['culmen_depth_mm'].mean()),
inplace=True)
data['flipper_length_mm'].fillna((data['flipper_length_mm'].mean()),
inplace=True)
data['body_mass_g'].fillna((data['body_mass_g'].mean()), inplace=True)

data['sex'].fillna((data['sex'].value_counts().index[0]), inplace=True)

data.reset_index()
data.head()
```

Observe what the mentioned third sample looks like now:

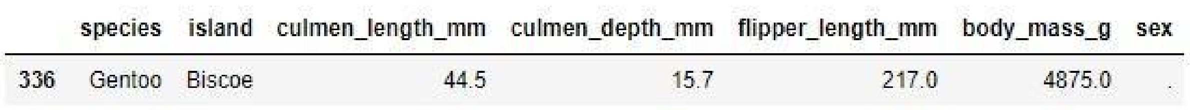


	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.10000	18.70000	181.000000	3750.000000	MALE
1	Adelie	Torgersen	39.50000	17.40000	186.000000	3800.000000	FEMALE
2	Adelie	Torgersen	40.30000	18.00000	195.000000	3250.000000	FEMALE
3	Adelie	Torgersen	43.92193	17.15117	200.915205	4201.754386	MALE
4	Adelie	Torgersen	36.70000	19.30000	193.000000	3450.000000	FEMALE

Often, data is not missing, but it has an **invalid** value. For example, we know that for the '*sex*' feature we can have two values: FEMALE and MALE.

We can check if we have values **other** than this:

```
data.loc[(data['sex'] != 'FEMALE') & (data['sex'] != 'MALE')]
```



	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_g	sex
336	Gentoo	Biscoe	44.5	15.7	217.0	4875.0	.

As it turns out, we have one record that has value '.' for this feature, which is not correct. We can observe these instances as missing data and drop them or replace them:

```
data = data.drop([336])
data.reset_index()
```

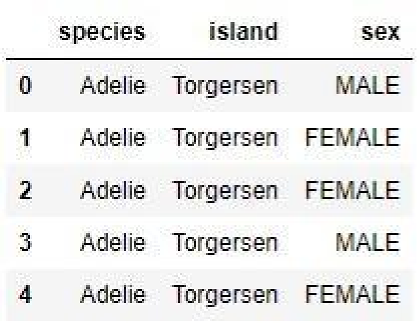
4.2 Categorical Encoding

One way to improve your predictions is by applying clever ways when working with **categorical variables**. These variables, as the name suggests, have discrete values and represent some sort of category or class. For example, color can be a categorical variable (*'red'*, *'blue*', *'green*'). The challenge is including these variables in data analysis and using them with machine learning algorithms. Some machine learning algorithms support categorical variables without further manipulation, but some don't. That is why we use **categorical encoding**. In this tutorial, we cover several types of categorical encoding, but before we continue, let's **extract** those variables from our dataset into a separate variable and mark them as categorical type:

```
data["species"] = data["species"].astype('category')
data["island"] = data["island"].astype('category')
data["sex"] = data["sex"].astype('category')
data.dtypes
```

```
species              category
island               category
culmen_length_mm      float64
culmen_depth_mm       float64
flipper_length_mm     float64
body_mass_g           float64
sex                  category
```

```
categorical_data = data.drop(['culmen_length_mm', 'culmen_depth_mm',
'flipper_length_mm', \
                              'body_mass_g'], axis=1)
categorical_data.head()
```



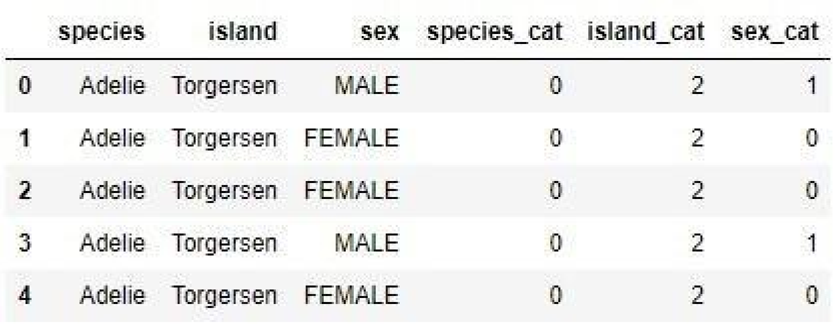
	species	island	sex
0	Adelie	Torgersen	MALE
1	Adelie	Torgersen	FEMALE
2	Adelie	Torgersen	FEMALE
3	Adelie	Torgersen	MALE
4	Adelie	Torgersen	FEMALE

Ok, now we are ready to roll. We start with the simplest form of encoding, Label Encoding.

4.2.1 Label Encoding

Label encoding is **converting** each categorical value into a number. For example, the '*species*' feature contains 3 categories. We can assign value 0 to *Adelie*, 1 to *Gentoo* and 2 to *Chinstrap*. To perform this technique we can use Pandas:

```
categorical_data["species_cat"] = categorical_data["species"].cat.codes
categorical_data["island_cat"] = categorical_data["island"].cat.codes
categorical_data["sex_cat"] = categorical_data["sex"].cat.codes
categorical_data.head()
```



	species	island	sex	species_cat	island_cat	sex_cat
0	Adelie	Torgersen	MALE	0	2	1
1	Adelie	Torgersen	FEMALE	0	2	0
2	Adelie	Torgersen	FEMALE	0	2	0
3	Adelie	Torgersen	MALE	0	2	1
4	Adelie	Torgersen	FEMALE	0	2	0

As you can see, we added three new features, each containing encoded categorical features. From the first five instances, we can see that the *species* category *Adelie* is encoded with value *0,* the *island* category *Torgensesn* is encoded with value *2* and the *sex* categories *FEMALE* and *MALE* are encoded with values 0 and 1, respectively.

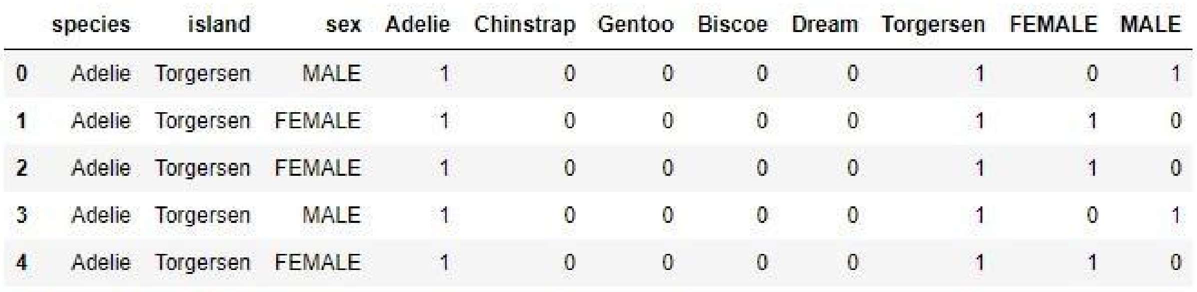
4.2.2 One-Hot Encoding

This is one of the most popular categorical encoding techniques. It spreads the values in a feature to **multiple** flag features and assigns values 0 or 1 to them. This binary value represents the **relationship** between non-encoded and encoded features.

For example, in our dataset, we have two possible values in the '*sex*' feature: *FEMALE* and *MALE*. This technique will create two separate features labeled let's say, '*sex_female*' and '*sex_male*'. If in the '*sex*' feature we have value '*FEMALE*' for some sample, the '*sex_female*' will be assigned value 1 and '*sex_male*' will be assigned value 0. In the same way, if in the '*sex*' feature we have the value '*MALE*' for some sample, the '*sex_male*' will be assigned value 1 and '*sex_female*' will be assigned value 0. Let's apply this technique to our categorical data and see what we get:

```
encoded_spicies = pd.get_dummies(categorical_data['species'])
encoded_island = pd.get_dummies(categorical_data['island'])
encoded_sex = pd.get_dummies(categorical_data['sex'])

categorical_data = categorical_data.join(encoded_spicies)
categorical_data = categorical_data.join(encoded_island)
categorical_data = categorical_data.join(encoded_sex)
```



	species	island	sex	Adelie	Chinstrap	Gentoo	Biscoe	Dream	Torgersen	FEMALE	MALE
0	Adelie	Torgersen	MALE	1	0	0	0	0	1	0	1
1	Adelie	Torgersen	FEMALE	1	0	0	0	0	1	1	0
2	Adelie	Torgersen	FEMALE	1	0	0	0	0	1	1	0
3	Adelie	Torgersen	MALE	1	0	0	0	0	1	0	1
4	Adelie	Torgersen	FEMALE	1	0	0	0	0	1	1	0

We have some new columns there. Essentially, every category in each feature got a separate column. Often, just one-hot encoded values are used as input to a machine learning algorithm.

4.2.3 Count Encoding

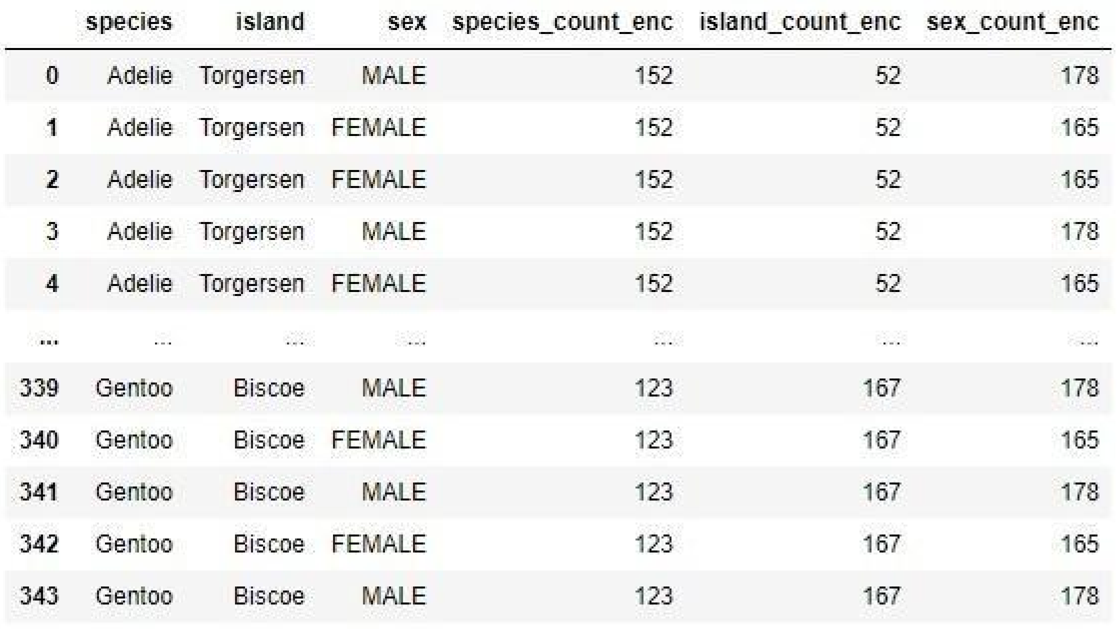
Count encoding is converting each categorical value to its frequency, i.e. the number of times it **appears** in the dataset. For example, if the '*species*' feature contains 6 occurrences of class *Adelie*, we will replace every *Adelie* value with the number 6. Here is how we do that in the code:

```
categorical_data = data.drop(['culmen_length_mm', 'culmen_depth_mm', \
                         'flipper_length_mm', 'body_mass_g'], axis=1)

species_count = categorical_data['species'].value_counts()
      island_count = categorical_data['island'].value_counts()
sex_count = categorical_data['sex'].value_counts()

categorical_data['species_count_enc'] =
categorical_data['species'].map(species_count)
categorical_data['island_count_enc'] =
categorical_data['island'].map(island_count)
categorical_data['sex_count_enc'] = categorical_data['sex'].map(sex_count)

categorical_data
```



	species	island	sex	species_count_enc	island_count_enc	sex_count_enc
0	Adelie	Torgersen	MALE	152	52	178
1	Adelie	Torgersen	FEMALE	152	52	165
2	Adelie	Torgersen	FEMALE	152	52	165
3	Adelie	Torgersen	MALE	152	52	178
4	Adelie	Torgersen	FEMALE	152	52	165
...	...	...	...	...	...	...
339	Gentoo	Biscoe	MALE	123	167	178
340	Gentoo	Biscoe	FEMALE	123	167	165
341	Gentoo	Biscoe	MALE	123	167	178
342	Gentoo	Biscoe	FEMALE	123	167	165
343	Gentoo	Biscoe	MALE	123	167	178

Notice how every category value is replaced with the number of occurrences.

4.2.4 Target Encoding

Unlike previous techniques, this one is a little bit more complicated. It replaces a categorical value with the **average** value of the output (i.e. target) for that value of the feature. Essentially, all you need to do is calculate the average output for all the rows with specific category value. Now, this is quite straightforward when the output value is numerical. If the output is categorical, like in our *PalmerPenguins* dataset, we need to apply some of the previous techniques to it.

Often, this average value is blended with the outcome **probability** over the entire dataset in order to reduce the variance of values with a few occurrences. It is important to note that since category values are calculated based on the output value, these calculations should be done on the training dataset and then **applied** to other datasets. Otherwise, we would face information **leakage**, meaning that we would include information about the output values from the test set inside of the training set. This would render our tests invalid or give us false confidence. Ok, let's see how we can do this in code:

```
categorical_data["species"] = categorical_data["species"].cat.codes

island_means = categorical_data.groupby('island')['species'].mean()
sex_means = categorical_data.groupby('sex')['species'].mean()
```

Here we used label encoding for output feature and then calculated mean values for categorical features '*island*' and '*sex*'. Here is what we get for the '*island*' feature:

```
island_means
```

```
island
Biscoe       1.473054
Dream        0.548387
Torgersen    0.000000
```

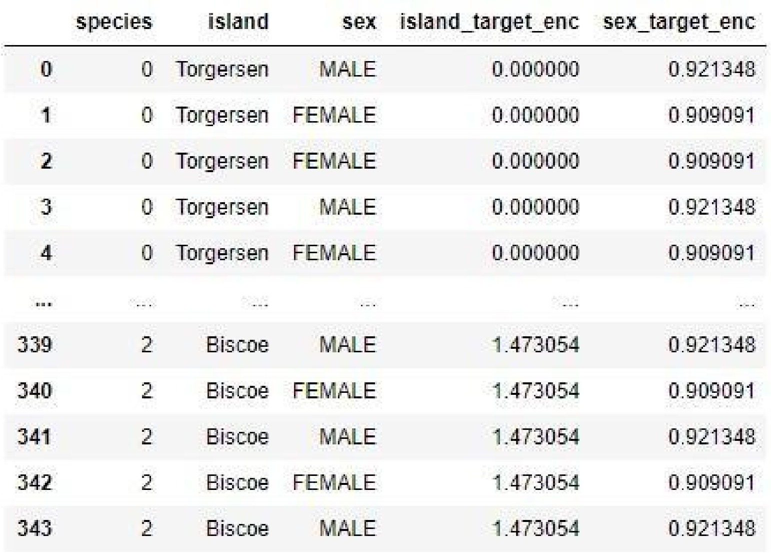
This means that values *Biscoe*, *Dream* and *Torgersen* will be replaced with values 1.473054, 0.548387 and 0 respectively. For the '*sex*' feature we have a similar situation:

```
sex_means
```

```
sex
FEMALE    0.909091
MALE      0.921348
```

This means that values *FEMALE* and *MALE* will be replaced with 0.909091 and 0.921348 respectively. Here is what that looks like in the dataset:

```
categorical_data['island_target_enc'] =
categorical_data['island'].map(island_means)
categorical_data['sex_target_enc'] = categorical_data['sex'].map(sex_means)
categorical_data
```



	species	island	sex	island_target_enc	sex_target_enc
0	0	Torgersen	MALE	0.000000	0.921348
1	0	Torgersen	FEMALE	0.000000	0.909091
2	0	Torgersen	FEMALE	0.000000	0.909091
3	0	Torgersen	MALE	0.000000	0.921348
4	0	Torgersen	FEMALE	0.000000	0.909091
...	...	...	...	...	...
339	2	Biscoe	MALE	1.473054	0.921348
340	2	Biscoe	FEMALE	1.473054	0.909091
341	2	Biscoe	MALE	1.473054	0.921348
342	2	Biscoe	FEMALE	1.473054	0.909091
343	2	Biscoe	MALE	1.473054	0.921348

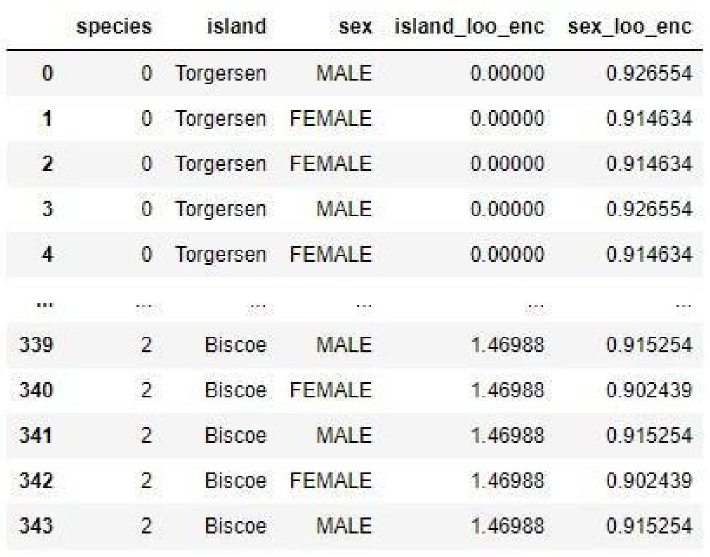
4.2.5 Leave One Out Target Encoding

The final type of encoding that we explore in this tutorial is built on top of Target Encoding. It works in the same way as Target encoding, with one difference. When we calculate the mean output value for the sample, we **exclude** that sample. Here is how it is done in the code. First, we define a function that does this:

```
def leave_one_out_mean(series):
      series = (series.sum() - series)/(len(series) - 1)
      return series
```

And then we apply it to categorical values in our dataset:

```
categorical_data['island_loo_enc'] =
categorical_data.groupby('island')['species'].apply(leave_one_out_mean)
categorical_data['sex_loo_enc'] =
categorical_data.groupby('sex')['species'].apply(leave_one_out_mean)
categorical_data
```



	species	island	sex	island_loo_enc	sex_loo_enc
0	0	Torgersen	MALE	0.00000	0.926554
1	0	Torgersen	FEMALE	0.00000	0.914634
2	0	Torgersen	FEMALE	0.00000	0.914634
3	0	Torgersen	MALE	0.00000	0.926554
4	0	Torgersen	FEMALE	0.00000	0.914634
...	...	...	...	...	...
339	2	Biscoe	MALE	1.46988	0.915254
340	2	Biscoe	FEMALE	1.46988	0.902439
341	2	Biscoe	MALE	1.46988	0.915254
342	2	Biscoe	FEMALE	1.46988	0.902439
343	2	Biscoe	MALE	1.46988	0.915254

4.3 Binning

Binning is a simple technique that **groups** different values into **bins**. For example, when we want to bin numerical features, that would look something like this:

- 0-10 – Low
- 10-50 – Medium
- 50-100 – High

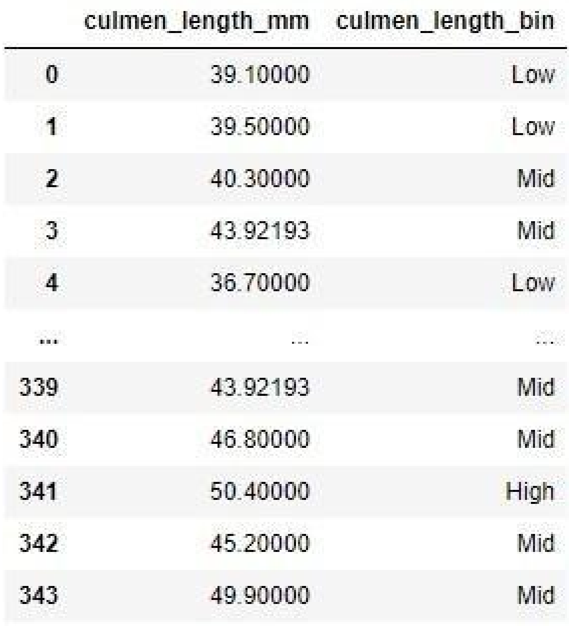
- In this particular case, we replace numerical features with categorical ones.

However, we can bin categorical values too. For example, we can bin countries by the continent they are on:

- Serbia – Europe
- Germany – Europe
- Japan – Asia
- China – Asia
- USA – North America
- Canada – North America

The problem with binning is that it can downgrade performance, but it can prevent overfitting and increase the robustness of the machine learning model. Here is what that looks like in the code:

```
bin_data = data[['culmen_length_mm']]
bin_data['culmen_length_bin'] = pd.cut(data['culmen_length_mm'], bins=[0, 
40, 50, 100], \
                                       labels=["Low", "Mid", "High"])
bin_data
```



	culmen_length_mm	culmen_length_bin
0	39.10000	Low
1	39.50000	Low
2	40.30000	Mid
3	43.92193	Mid
4	36.70000	Low
...	...	...
339	43.92193	Mid
340	46.80000	Mid
341	50.40000	High
342	45.20000	Mid
343	49.90000	Mid

4.4 Scaling

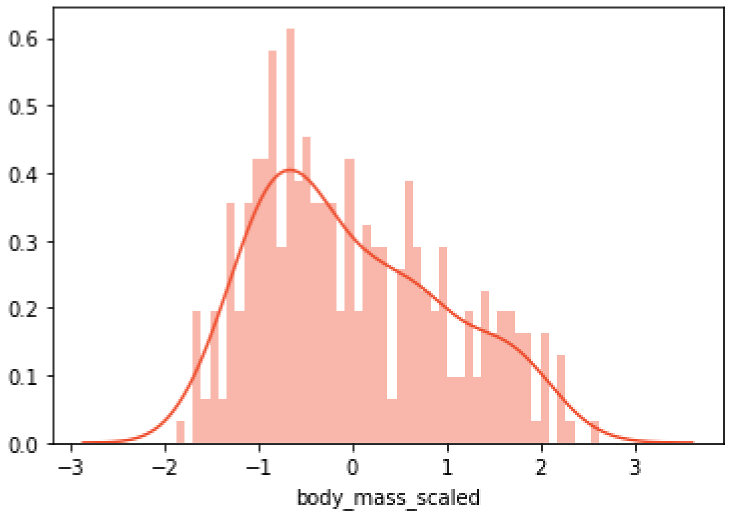
In the previous articles, we often had a chance to see how scaling helps machine learning models make better predictions. **Scaling** is done for one simple reason; if features are not in the same range, they will be treated **differently** by the machine learning algorithm. To put it in layman's terms, if we have one feature that has a range of values from 0-10 and another 0-100, a machine learning algorithm might **deduce** that the second feature is more important than the first one just because it has a higher value. As we already know, that is not always the case. On the other hand, it is unrealistic to expect that real data comes in the same range. That is why we use **scaling** to put our numerical features into the same **range**. This **standardization** of data is a common requirement for many machine learning algorithms. Some of them even require that features look like standard, normally distributed data. There are several ways we can scale and standardize the data, but before we go through them, let's observe one feature of PalmerPenguins dataset '*body_mass_g*'.

```
scaled_data = data[['body_mass_g']]

print('Mean:', scaled_data['body_mass_g'].mean())
print('Standard Deviation:', scaled_data['body_mass_g'].std())
```

```
Mean: 4199.791570763644
Standard Deviation: 799.9508688401579
```

Also, observe the distribution of this feature:



First, let's explore the scaling techniques that preserve distribution.

4.4.1 Standard Scaling

This type of scaling **removes** mean and scale data to unit variance. It is defined by the formula:

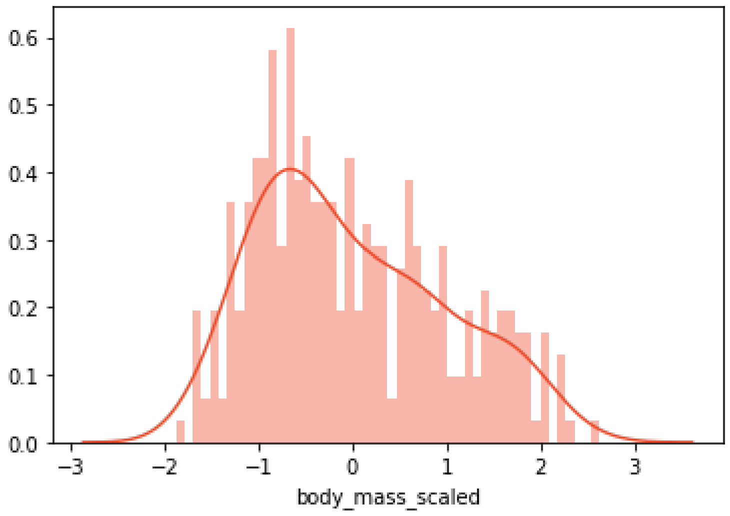
$$x_{scaled} = (x - mean)/std$$

where **mean** is the mean of the training samples, and **std** is the standard deviation of the training samples. The best way to understand it is to look at it in practice. For that we use *SciKit Learn* and *StandardScaler* class:

```
standard_scaler = StandardScaler()
scaled_data['body_mass_scaled'] =
standard_scaler.fit_transform(scaled_data[['body_mass_g']])

print('Mean:', scaled_data['body_mass_scaled'].mean())
print('Standard Deviation:', scaled_data['body_mass_scaled'].std())
```

```
Mean: -1.6313481178165566e-16
Standard Deviation: 1.0014609211587777
```



We can see that the original distribution of data is **preserved**. However, now data is in range -3 to 3.

4.4.2 Min-Max Scaling (Normalization)

The most popular scaling technique is **normalization** (also called *min-max normalization* and *min-max scaling*). It scales all data in the 0 to 1 range. This technique is defined by the formula:

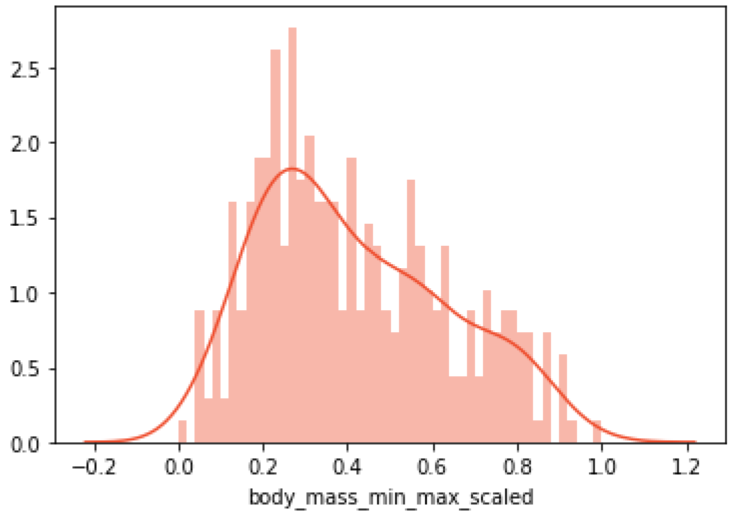
$$X_{std} = (X - X.min(axis=0))/(X.max(axis=0) - X.min(axis=0))$$
$$X_{scaled} = X_{std} * (max - min) + min$$

If we use MinMaxScaler from SciKit learn library:

```
minmax_scaler = MinMaxScaler()
scaled_data['body_mass_min_max_scaled'] =
minmax_scaler.fit_transform(scaled_data[['body_mass_g']])

print('Mean:', scaled_data['body_mass_min_max_scaled'].mean())
print('Standard Deviation:', scaled_data['body_mass_min_max_scaled'].std())
```

```
Mean: 0.4166087696565679
Standard Deviation: 0.2222085746778217
```



Distribution is preserved, but the data is now in range from 0 to 1.

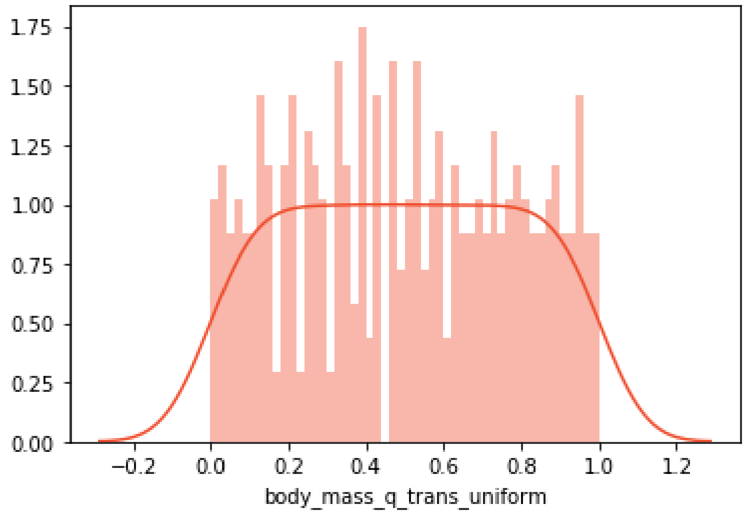
4.4.3 Quantile Transformation

As we mentioned, sometimes machine learning algorithms **require** that the distribution of our data is **uniform** or **normal**. We can achieve that using *QuantileTransformer* class from *SciKit Learn*. First, here is what it looks like when we transform our data to uniform distribution:

```
qtrans = QuantileTransformer()
scaled_data['body_mass_q_trans_uniform'] =
qtrans.fit_transform(scaled_data[['body_mass_g']])

print('Mean:', scaled_data['body_mass_q_trans_uniform'].mean())
print('Standard Deviation:',
scaled_data['body_mass_q_trans_uniform'].std())
```

```
Mean: 0.5002855778903038
Standard Deviation: 0.2899458384920982
```

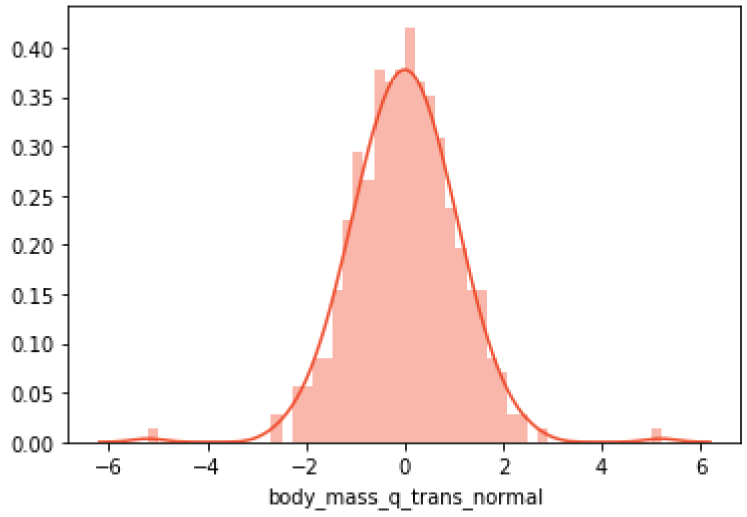


Here is the code that puts your data into normal distribution:

```
qtrans = QuantileTransformer(output_distribution='normal', random_state=0)
scaled_data['body_mass_q_trans_normal'] =
qtrans.fit_transform(scaled_data[['body_mass_g']])

print('Mean:', scaled_data['body_mass_q_trans_normal'].mean())
print('Standard Deviation:', scaled_data['body_mass_q_trans_normal'].std())
```

```
Mean: 0.0011584329410665568
Standard Deviation: 1.0603614567765762
```



Essentially, we use the *output_distribution* parameter in the constructor to define the type of distribution. Finally, we can observe scaled values of all features, with different types of scaling:

	body_mass_g	body_mass_scaled	body_mass_min_max_scaled	body_mass_q_trans_uniform	body_mass_q_trans_normal
0	3750.000000	-0.563095	0.291667	0.356725	-0.367226
1	3800.000000	-0.500500	0.305556	0.394737	-0.266994
2	3250.000000	-1.189047	0.152778	0.087719	-1.354934
3	4201.754386	0.002457	0.417154	0.565789	0.165664
4	3450.000000	-0.938666	0.208333	0.185673	-0.893957
...	...	...	...	...	...
339	4201.754386	0.002457	0.417154	0.565789	0.165664
340	4850.000000	0.813998	0.597222	0.773392	0.750064
341	5750.000000	1.940711	0.847222	0.967836	1.849903
342	5200.000000	1.252164	0.694444	0.853801	1.052876
343	5400.000000	1.502545	0.750000	0.894737	1.252120

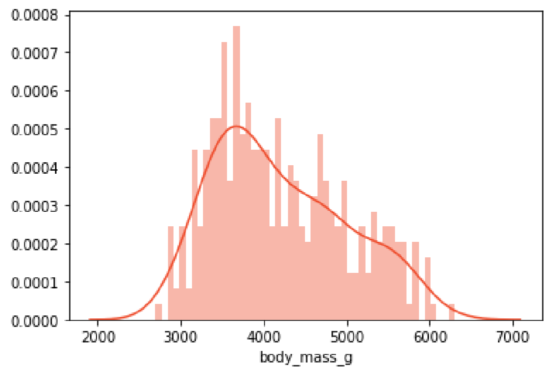
4.5 Log Transform

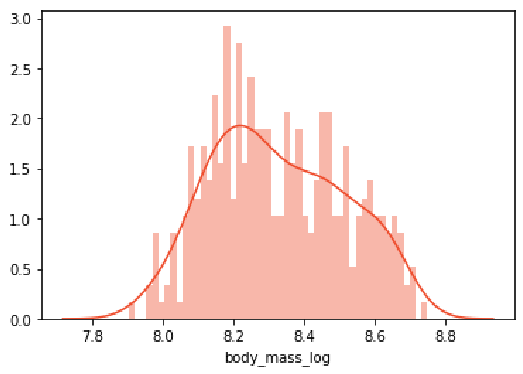
One of the most popular mathematical transformations of data is **logarithm transformation**. Essentially, we just apply the *log* function to the current values. It is important to note that data must be **positive**, so if you need a scale or normalize data beforehand, this transformation brings many **benefits**. One of them is that the distribution of the data becomes more **normal**. In turn, this helps us to handle **skewed** data and decreases the impact of the **outliers**. Here is what that looks like in the code:

```
log_data = data[['body_mass_g']]
log_data['body_mass_log'] = (data['body_mass_g'] + 1).transform(np.log)
log_data
```

	body_mass_g	body_mass_log
0	3750.000000	8.229778
1	3800.000000	8.243019
2	3250.000000	8.086718
3	4201.754386	8.343495
4	3450.000000	8.146419
...	...	...
339	4201.754386	8.343495
340	4850.000000	8.486940
341	5750.000000	8.657129
342	5200.000000	8.556606
343	5400.000000	8.594339

If we check the distribution of non-transformed data and transformed data, we can see that transformed data is closer to the normal distribution:





4.6 Feature Selection

Datasets coming from the client are often huge. We can have hundreds or even thousands of features, especially if we perform some of the techniques from above. A large number of features can lead to **overfitting**. Apart from that, optimizing hyperparameters and training algorithms, in general, will take longer. That is why we want to pick the most **relevant** features from the beginning.

There are several techniques when it comes to feature selection; however, in this tutorial, we cover only the simplest one (and the most often used) – **Univariate Feature Selection**. This method is based on univariate statistical tests. It calculates how strongly the output feature depends on each feature from the dataset using statistical tests (like *χ2*). In this example, we utilize *SelectKBest* which has several options when it comes to used statistical tests (the default, however, is *χ2* and we use that one in this example). Here is how we can do it:

```
feature_sel_data = data.drop(['species'], axis=1)

feature_sel_data["island"] = feature_sel_data["island"].cat.codes
feature_sel_data["sex"] = feature_sel_data["sex"].cat.codes

# Use 3 features
selector = SelectKBest(f_classif, k=3)

selected_data = selector.fit_transform(feature_sel_data, data['species'])
selected_data
```

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

Using hyperparameter k, we defined that we want to keep the 3 most influential features from the dataset. The output of this operation is a NumPy array which contains selected features. To make it into *pandas Dataframe*, we need to do the following:

```
selected_features = pd.DataFrame(selector.inverse_transform(selected_data),
                                 index=data.index,
                                 columns=feature_sel_data.columns)

selected_columns = selected_features.columns[selected_features.var() != 0]
selected_features[selected_columns].head()
```

	culmen_length_mm	culmen_depth_mm	flipper_length_mm
0	39.10000	18.70000	181.000000
1	39.50000	17.40000	186.000000
2	40.30000	18.00000	195.000000
3	43.92193	17.15117	200.915205
4	36.70000	19.30000	193.000000

4.7 Feature Grouping

The dataset that we observed so far is an almost perfect situation when it comes to terms of so-called "**tidiness**". This means that each feature has its own column, each observation is a row, and each type of observational unit is a table. However, sometimes we have observations that are **spread** over several rows. The goal of the *Feature Grouping* is to connect these rows into a single one and then use those aggregated rows. The main question when doing so is which type of aggregation function will be applied to features. This is especially complicated for categorical features.

As we mentioned, *PalmerPenguins* dataset is very tidy so the following example is just educational to show the code that can be used for this operation:

```
grouped_data = data.groupby('species')

sums_data = grouped_data['culmen_length_mm',
'culmen_depth_mm'].sum().add_suffix('_sum')
avgs_data = grouped_data['culmen_length_mm',
'culmen_depth_mm'].mean().add_suffix('_mean')

sumed_averaged = pd.concat([sums_data, avgs_data], axis=1)
sumed_averaged
```

Here we grouped data by *species* value and for each numerical value, we created two new features with sum and mean value.

	culmen_length_mm_sum	culmen_depth_mm_sum	culmen_length_mm_mean	culmen_depth_mm_mean
species				
0	5901.42193	2787.45117	38.825144	18.338495
1	3320.70000	1252.60000	48.833824	18.420588
2	5842.52193	1844.25117	47.500178	14.993912

4.8 Feature Split

Sometimes, data is not connected over rows, but over **columns**. For example, imagine you have a list of names in one of the features:

```
data.names
```

```
0 Andjela Zivkovic
1 Vanja Zivkovic
2 Petar Zivkovic
3 Veljko Zivkovic
4 Nikola Zivkovic
```

So, if we want to extract only the first name from this feature, we can do the following:

```
data.names
```

```
0    Andjela
1    Vanja
2    Petar
3    Veljko
4    Nikola
```

This technique is called feature splitting and it is often used with string data.