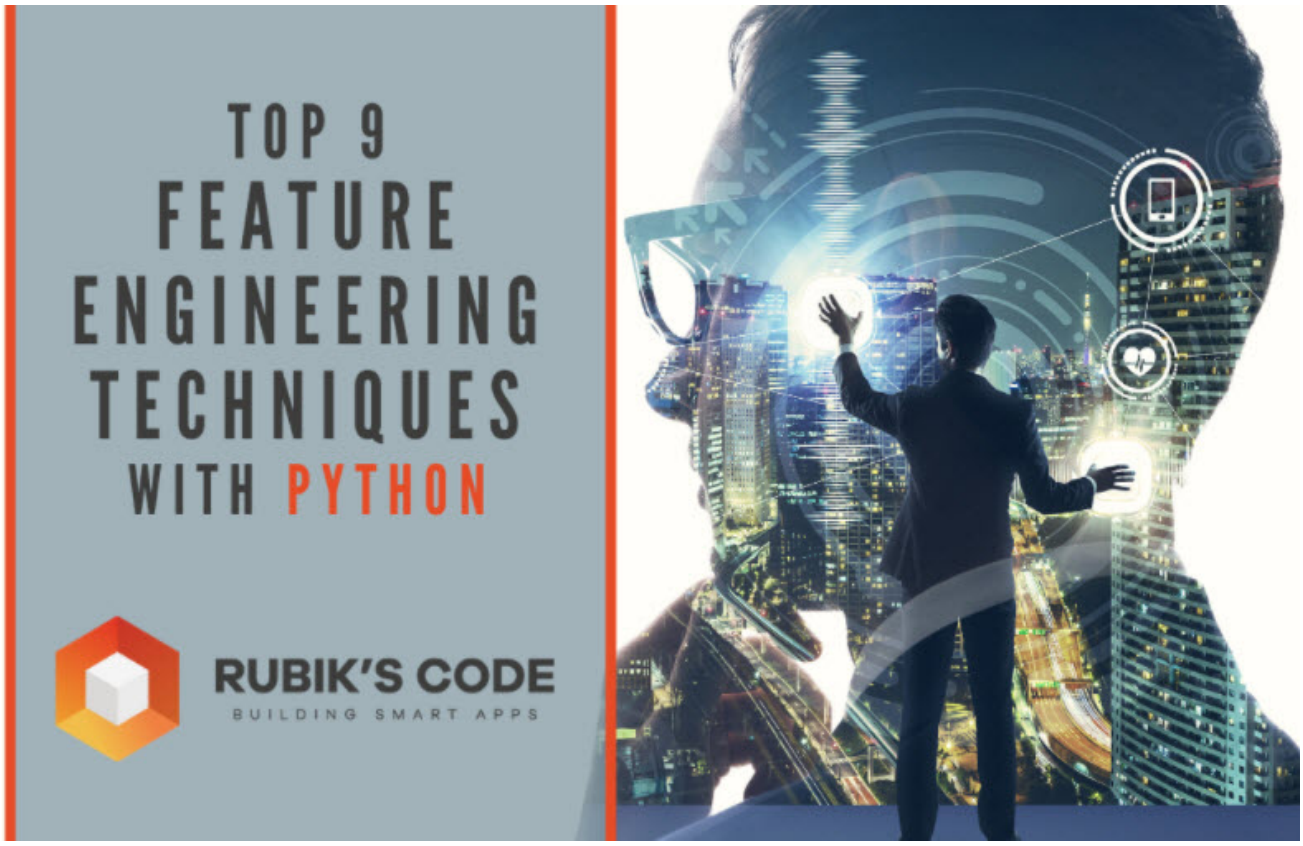
Top 9 Feature Engineering Techniques with Python



Don't get me wrong, 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. In this article we explore:

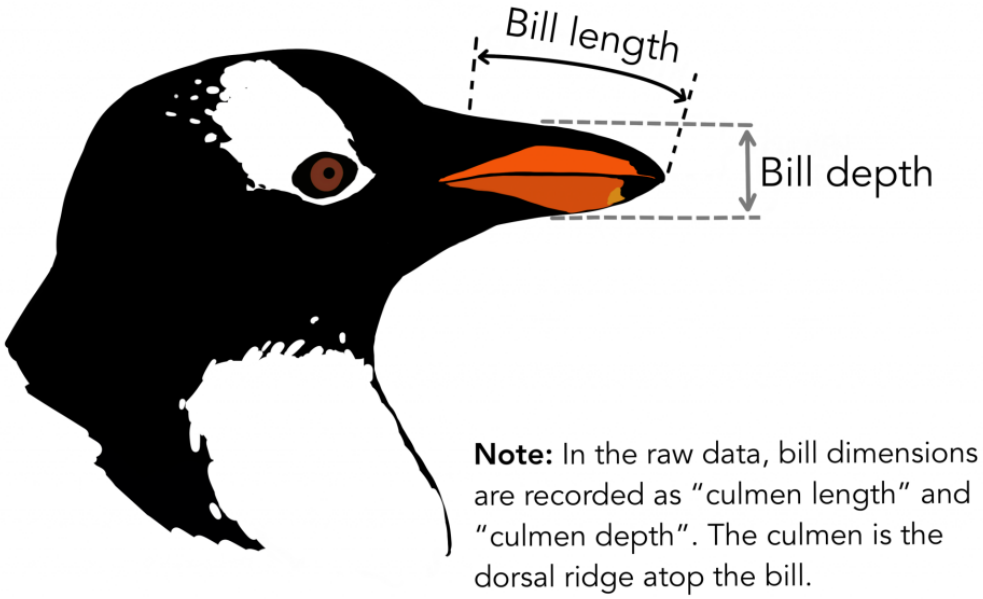
Dataset & Prerequisites

For the purpose of this tutorial, make sure that you have installed the following *Python* libraries:

- **NumPy** – Follow **this guide** if you need help with installation.
- **SciKit Learn** – Follow **this guide** if you need help with installation.
- **Pandas** – Follow **this guide** if you need help with installation.
- **Matplotlib** – Follow **this guide** if you need help with installation.
- **SeaBorn** – Follow **this guide** if you need help with installation.

Once installed make sure that you have imported all the necessary modules that are used in this tutorial.

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

from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler,
QuantileTransformer
from sklearn.feature_selection import SelectKBest, f_classif
```



Data that we use in this article 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 data of 344 penguins. Just like in 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*. Loading this dataset is done using Pandas:

```
data = pd.read_csv('./data/penguins_size.csv')
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
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

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 some value, or use the **mean** value of the feature, or an **average** value of the feature, etc. Still, we need 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 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 how 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 turnes out we have one record that has value '.' for this feature, which is not correct. We can observe these instances as a missing data and drop them or replace them:

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

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 categorical variable (‘*red*’, ‘*blue*‘, ‘*green*‘).

The challenge is including these variables into data analysis and use them with machine learning algorithms. Some machine learning algorithms support categorical variables without further manipulation, but some don’t. That is why we use a **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()
```

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

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

2.1 Label Encoding

Label encoding is **converting** each categorical value into some 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:

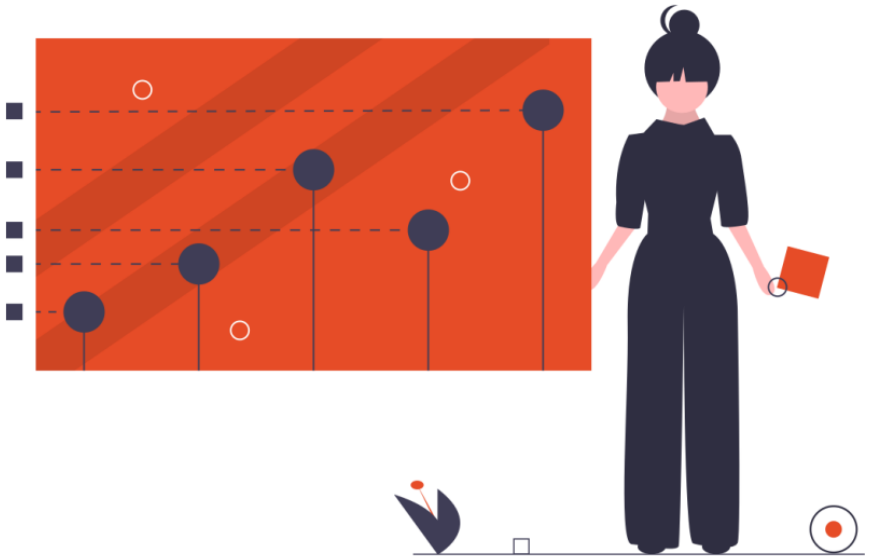
```
categorical_data["species_cat"] = categorical_data["species"].cat.codes
categorical_data["island_cat"] = categorical_data["island"].cat.codes
categorical_data["sex_cat"] = categorical_data["sex"].cat.codes
categorical_data.head()
```

	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 *species* category *Adelie* is encoded with value *0, island* category *Torgensesn* is encoded with value *2* and *sex* categories *FEMALE* and *MALE* are encoded with values 0 and 1 respectively.

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 '*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

As you we gave 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.

2.3 Count Encoding

Count encoding is converting each categorical value to its frequency, ie. 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:

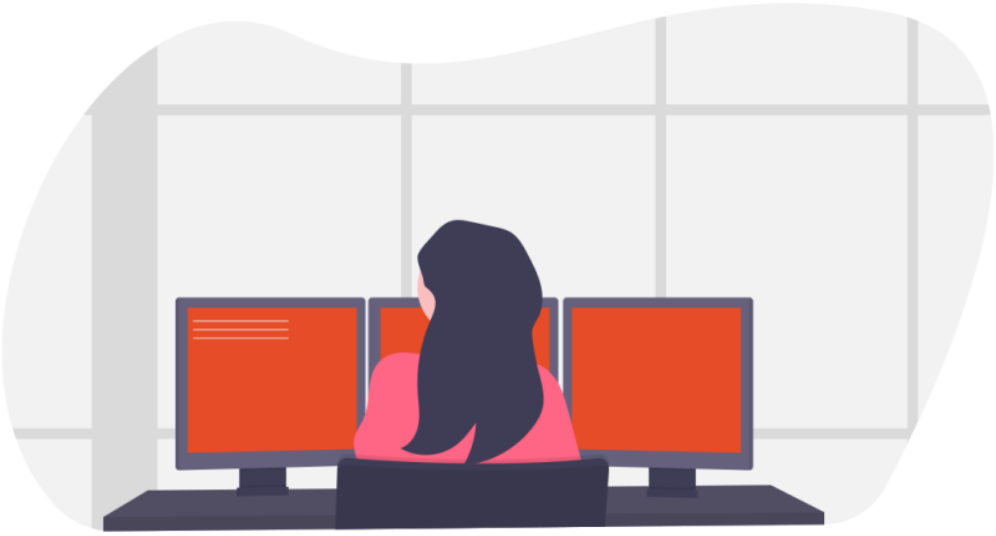
```
categorical_data = data.drop(['culmen_length_mm', 'culmen_depth_mm', \
                              'flipper_length_mm', 'body_mass_g'], axis=1)

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 occurances.

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 (ie. 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 straight forward 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 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
```

Meaning 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

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 are calculating 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

3. Handling Outliers

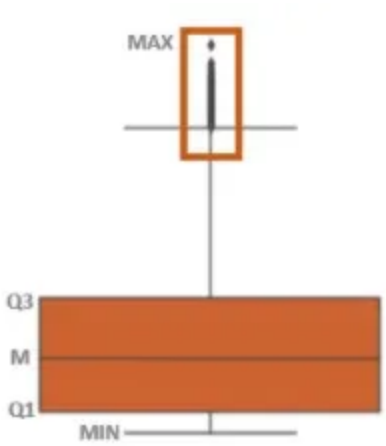
Outliers are values that are deviating from the whole **distribution** of the data. Sometimes these values are mistakes and wrong measurements and should be removed from datasets, but sometimes they are valuable **edge-case** information. This means that sometimes we want to leave these values in the 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 **Inter-quartile range** to detect these points. Inter-quartile range or *IQR* indicates where 50 percent of data is located. When we are looking for this value we first look for the median, since it splits data into half. Then we are locating 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*).

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 how one example of a boxplot looks like:

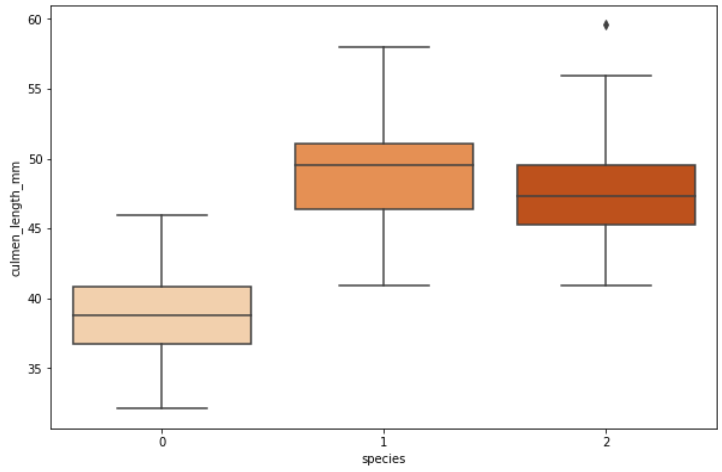
Let's apply it to *PalmerPenguins* dataset:

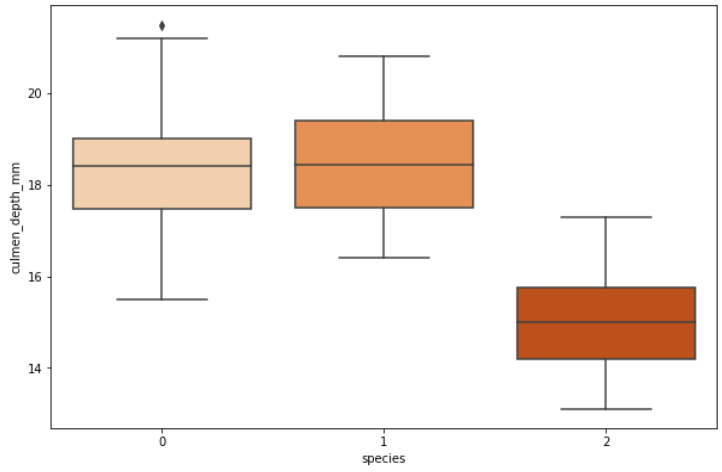
```
fig, axes = plt.subplots(nrows=4,ncols=1)
fig.set_size_inches(10, 30)
sb.boxplot(data=data,y="culmen_length_mm",x="species'
 palette="Oranges")
sb.boxplot(data=data,y="culmen_depth_mm",x="species",
 palette="Oranges")
sb.boxplot(data=data,y="flipper_length_mm",x="species
 palette="Oranges")
sb.boxplot(data=data,y="body_mass_g",x="species",orie
 palette="Oranges")
```

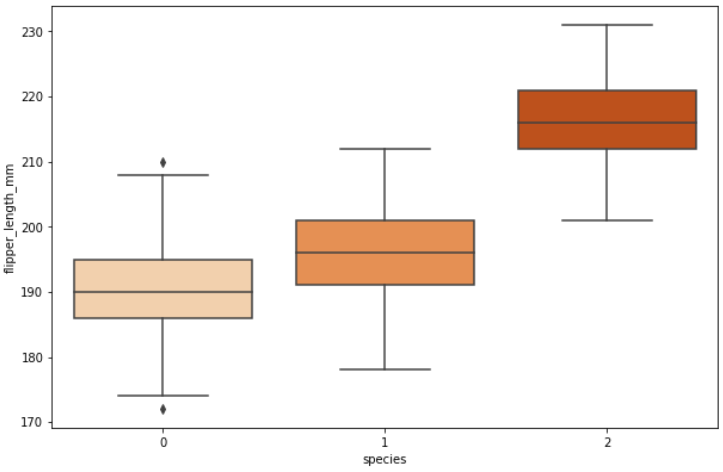


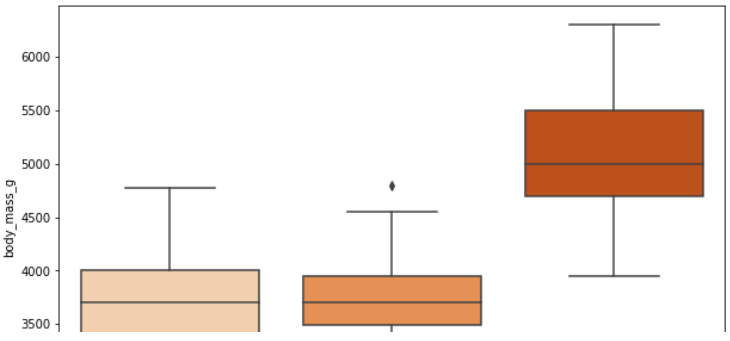
The other way for detecting and removing outliers would by using standard deviation.

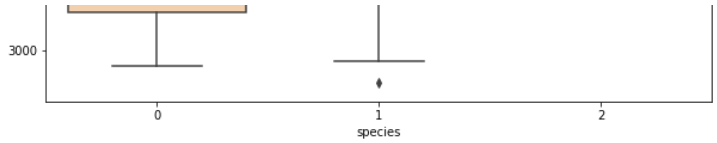
```
factor = 2
upper_lim = data['culmen_length_mm'].mean () +
data['culmen_length_mm'].std () * factor
lower_lim = data['culmen_length_mm'].mean () -
data['culmen_length_mm'].std () * factor

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


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



	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 now we 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 is to use **percentiles**. We can assume a certain percentage of the value from the top or the bottom as an outlier. Again a value for the percentiles we use as outliers border depends on the distribution of the data. Here is what we can do on *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

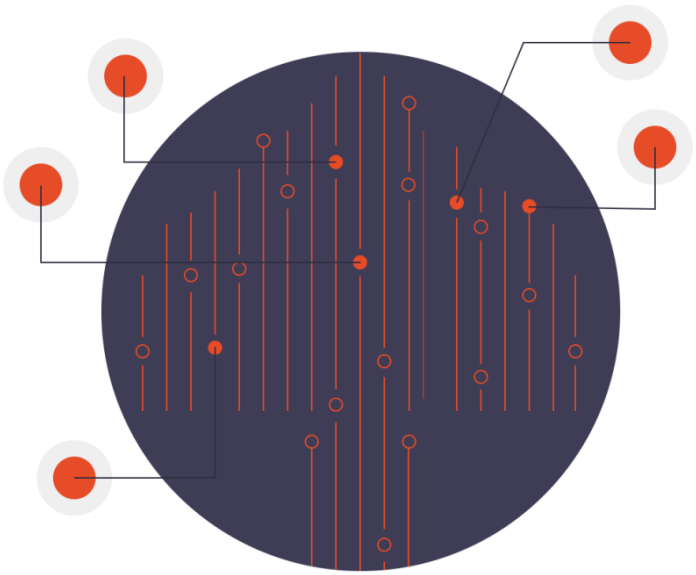
After this operation, we have 305 samples in our dataset. With this approach we need to be extremely careful since it reduces the dataset size and highly depends on the data distribution.

4. Binning

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

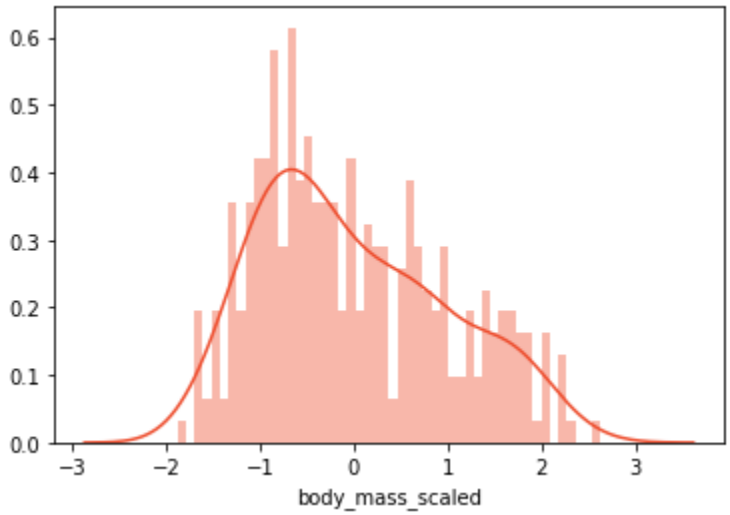
5. Scaling

In previous articles, we often had a chance to 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 lame 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 scaling techniques that preserve distribution.

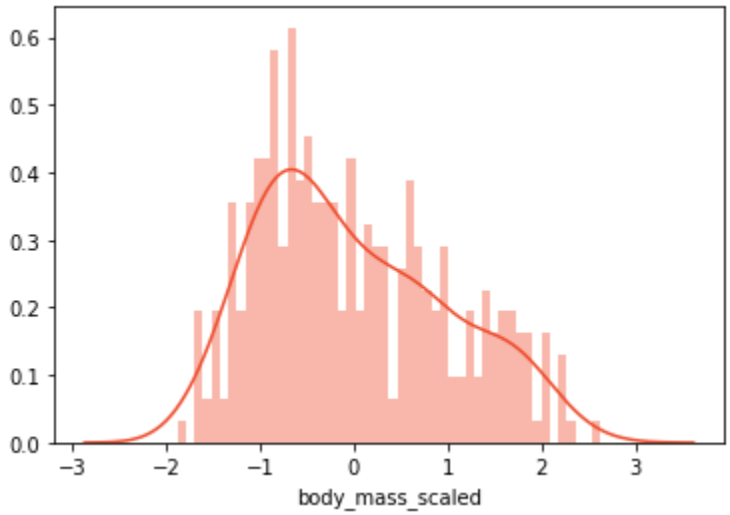
5.1 Standard Scaling

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.

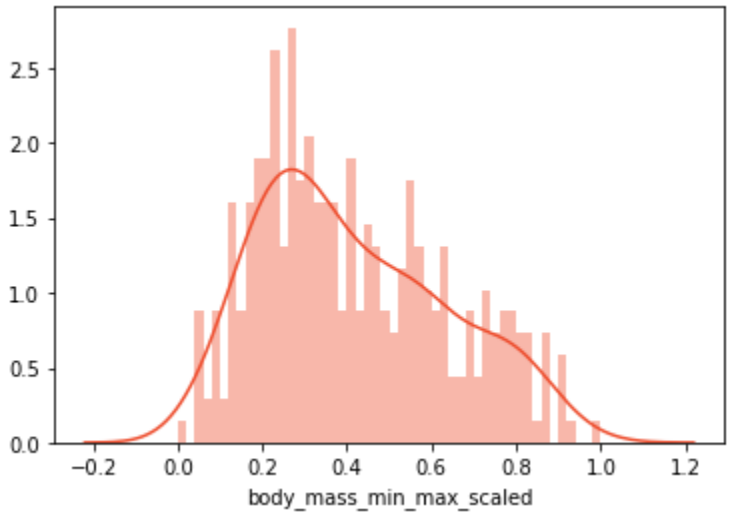
5.2 Min-Max Scaling (Normalization)

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
```

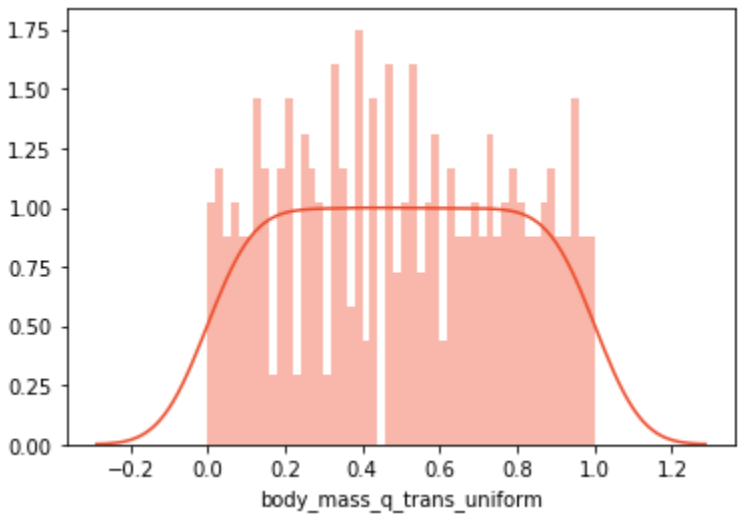


Distribution is perserved, but data is now in range from 0 to 1.

5.3 Quantile Transformation

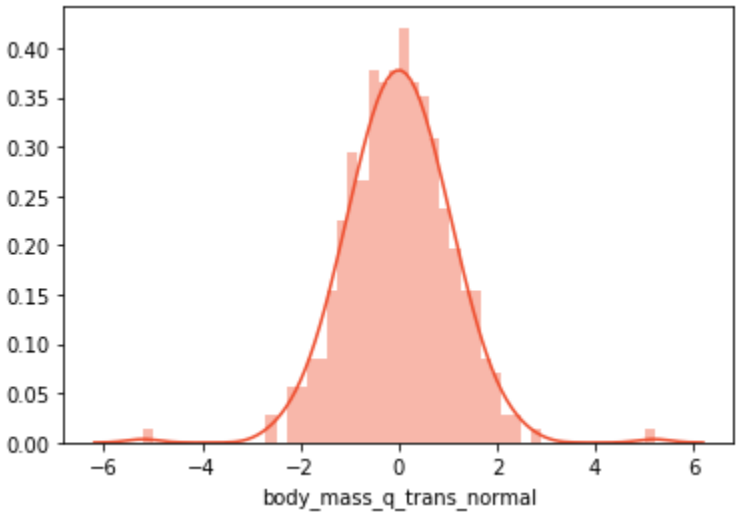
As we mentioned, sometimes machine learning algorithms **require** that the distribution of our data is **uniform** or **normal**. We can achieve that using *QuantileTransformer* class from *SciKit Learn*. First, here is how it looks like when we transform our data to uniform distribution:

```
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

6. Log Transform

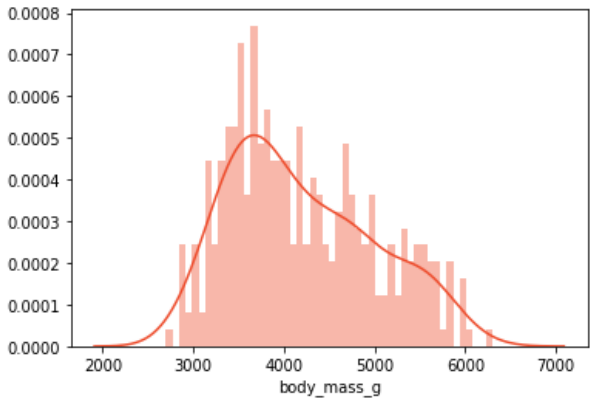
One of the most popular mathematical transformations of data is **logarithm transformation**. Essentially, we just apply the *log* function to the current values. It is important to note that data must be **positive**, so if you need a scale or normalize data

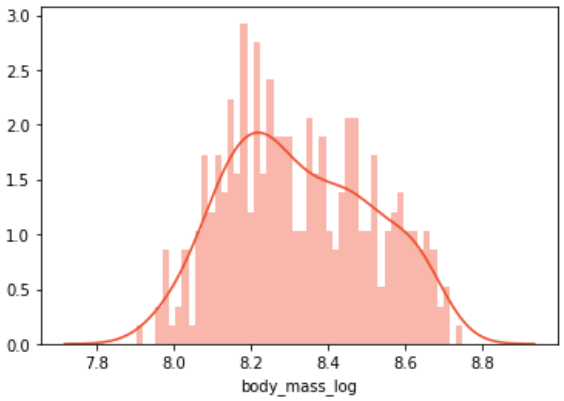
beforehand. This transformation brings many **benefits**. One of them is that the distribution of the data becomes more **normal**. In turn, this helps us to handle **skewed** data and decreases the impact of the **outliers**. Here is what that looks like in the code:

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

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

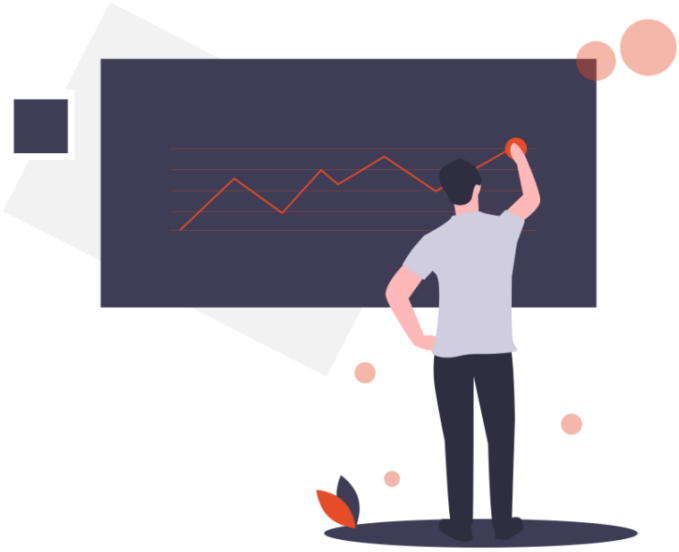
	body_mass_g	body_mass_log
0	3750.000000	8.229778
1	3800.000000	8.243019
2	3250.000000	8.086718
3	4201.754386	8.343495
4	3450.000000	8.146419
...	...	...
339	4201.754386	8.343495
340	4850.000000	8.486940
341	5750.000000	8.657129
342	5200.000000	8.556606
343	5400.000000	8.594339





7. Feature Selection

Datasets that are 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 *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

8. 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 it's 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 tydi 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 *spicies* value and for each numerical value we created two new features with sum and mean value.

species	culmen_length_mm_sum	culmen_depth_mm_sum	culmen_length_mm_mean	culmen_depth_mm_mean
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

9. Feature Split

Sometimes, data is not connected over rows, but over **colums**. For example, imagine you have 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 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 spliting and it is often used with string data.

Where to go from here?

In general, Machine Learning algorithms are a part of some bigger application. More often than not, we need to use features for other parts of the application. This is just one of the many reasons why taking Machine Learning applications to production is especially challenging. One way to mitigate this problem is by using so-called Feature Stores. This is a relatively new concept in data architecture, however it is already applied by companies such as Uber and Gojek.

The feature store is both a computational and storage service. In essence, they expose features, so they can be discovered and used as part of Machine Learning pipelines and online applications. Since they need to provide both, storing large volumes of data and providing low computational latency, Feature Stores are implemented as dual-database

systems. One one end there is a low latency key-value store or real-time database and on the other end there is a SQL database that can store a lot of data. This is an interesting concept that can be further explored.