Feature Engineering in python

In [66]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb

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

In [67]:
```
data = pd.read_csv('penguins_size.csv')
data.head(n=5)
```

Out[67]:

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_
0	Adelie	Torgersen	39.1	18.7	181.0	3750
1	Adelie	Torgersen	39.5	17.4	186.0	3800
2	Adelie	Torgersen	40.3	18.0	195.0	3250
3	Adelie	Torgersen	NaN	NaN	NaN	Na
4	Adelie	Torgersen	36.7	19.3	193.0	3450

The main parts of feature engineering are :

- Imputation
- Categorical Encoding
- Scaling or normalizing
- Feature Selection
- Log transform
- Feature Grouping

We will be exploring all this technique in python using our penguins dataset.

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.

However, let's first detect missing data. For that we can use Pandas:

In [68]:
```python
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
dtype: int64
```

We can see that there are a few missing data in the dataset.

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:

In [69]:
```python
data = data.dropna()
data.head()
```

Out[69]:

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_
0	Adelie	Torgersen	39.1	18.7	181.0	3750
1	Adelie	Torgersen	39.5	17.4	186.0	3800
2	Adelie	Torgersen	40.3	18.0	195.0	3250
4	Adelie	Torgersen	36.7	19.3	193.0	3450
5	Adelie	Torgersen	39.3	20.6	190.0	3650

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.

In [70]:
```python
data = data.fillna(0) ##filling with 0 for NA values
```

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:

In [71]:
```
data = pd.read_csv('penguins_size.csv')

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

Out[71]:

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_
0	Adelie	Torgersen	39.10000	18.70000	181.000000	3750.0000
1	Adelie	Torgersen	39.50000	17.40000	186.000000	3800.0000
2	Adelie	Torgersen	40.30000	18.00000	195.000000	3250.0000
3	Adelie	Torgersen	43.92193	17.15117	200.915205	4201.7543
4	Adelie	Torgersen	36.70000	19.30000	193.000000	3450.0000

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:

In [72]:
```
data.loc[(data['sex'] != 'FEMALE') & (data['sex'] != 'MALE')]
```

Out[72]:

	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	body_mass_
336	Gentoo	Biscoe	44.5	15.7	217.0	4875.

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:

In [73]:
```
data = data.drop([336])
data.reset_index()
```

Out[73]:

	index	species	island	culmen_length_mm	culmen_depth_mm	flipper_length_mm	bc
0	0	Adelie	Torgersen	39.10000	18.70000	181.000000	3
1	1	Adelie	Torgersen	39.50000	17.40000	186.000000	3
2	2	Adelie	Torgersen	40.30000	18.00000	195.000000	3
3	3	Adelie	Torgersen	43.92193	17.15117	200.915205	4
4	4	Adelie	Torgersen	36.70000	19.30000	193.000000	3
...	...	...	...	...	...	...	
338	339	Gentoo	Biscoe	43.92193	17.15117	200.915205	4
339	340	Gentoo	Biscoe	46.80000	14.30000	215.000000	4
340	341	Gentoo	Biscoe	50.40000	15.70000	222.000000	5
341	342	Gentoo	Biscoe	45.20000	14.80000	212.000000	5
342	343	Gentoo	Biscoe	49.90000	16.10000	213.000000	5

343 rows × 8 columns

Categorical Encoding

Let's first see what categorical variables we have in our dataset

In [74]:
```
data.dtypes
```

Out[74]:
```
species                object
island                 object
culmen_length_mm      float64
culmen_depth_mm       float64
flipper_length_mm     float64
body_mass_g           float64
sex                    object
dtype: object
```

We can see that here species , island and sex are as object we have to make their type as category

In [75]:
```
data["species"] = data["species"].astype('category')
data["island"] = data["island"].astype('category')
data["sex"] = data["sex"].astype('category')
data.dtypes
```

Out[75]:
```
species                category
island                 category
culmen_length_mm        float64
culmen_depth_mm         float64
flipper_length_mm       float64
body_mass_g             float64
sex                    category
dtype: object
```

In [76]:
```
categorical_data = data.drop(['culmen_length_mm', 'culmen_depth_mm', 'flipper_
categorical_data.head()
```

Out[76]:

	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.

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:

In [77]:
```
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()
```

Out[77]:

	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

One-Hot Encoding

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:

In [78]:
```
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)
categorical_data.head()
```

Out[78]:

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

Scaling or Normalizing

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‘.

In [79]:
```
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
```

We can see how to do standard scaling. There are other scaling techniques too.

In [80]:
```
standard_scaler = StandardScaler()
scaled_data['body_mass_scaled'] = standard_scaler.fit_transform(scaled_data[[

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
```

```
/var/folders/c5/1k4yqmhd029_nwyg7zh8lm9r0000gn/T/ipykernel_4217/2894967456.py:
2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
able/user_guide/indexing.html#returning-a-view-versus-a-copy
  scaled_data['body_mass_scaled'] = standard_scaler.fit_transform(scaled_data[
['body_mass_g']])
```

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. If we use MinMaxScaler from SciKit learn library:

In [81]:
```
minmax_scaler = MinMaxScaler()
scaled_data['body_mass_min_max_scaled'] = minmax_scaler.fit_transform(scaled_

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
```

```
/var/folders/c5/1k4yqmhd029_nwyg7zh8lm9r0000gn/T/ipykernel_4217/194873350.py:2
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
able/user_guide/indexing.html#returning-a-view-versus-a-copy
  scaled_data['body_mass_min_max_scaled'] = minmax_scaler.fit_transform(scaled
_data[['body_mass_g']])
```

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:

In [82]:

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

```
/var/folders/c5/1k4yqmhd029_nwyg7zh8lm9r0000gn/T/ipykernel_4217/4211894989.py:
2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
able/user_guide/indexing.html#returning-a-view-versus-a-copy
  log_data['body_mass_log'] = (data['body_mass_g'] + 1).transform(np.log)
```

Out[82]:

	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

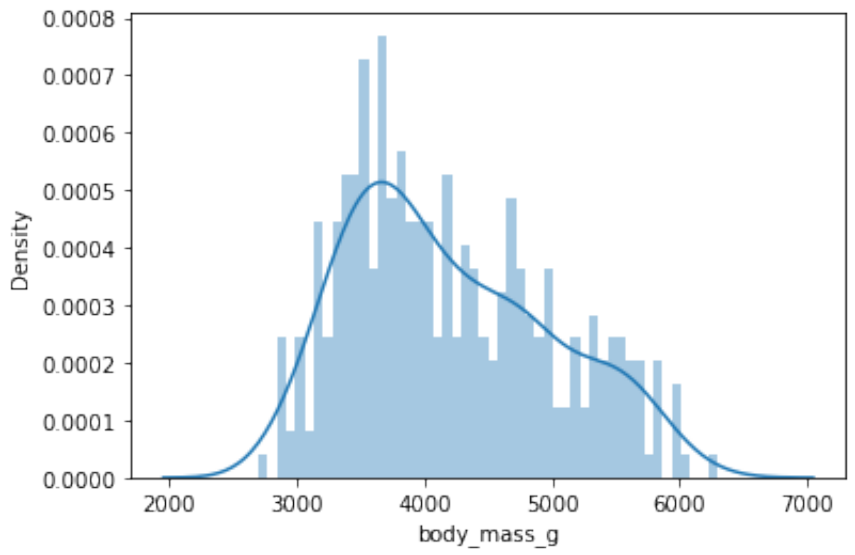
343 rows × 2 columns

In [83]:
```
# library & dataset
import seaborn as sns


# Plot the histogram
sns.distplot( a=data["body_mass_g"], hist=True, kde=True
             , rug=False, bins =50 )
```

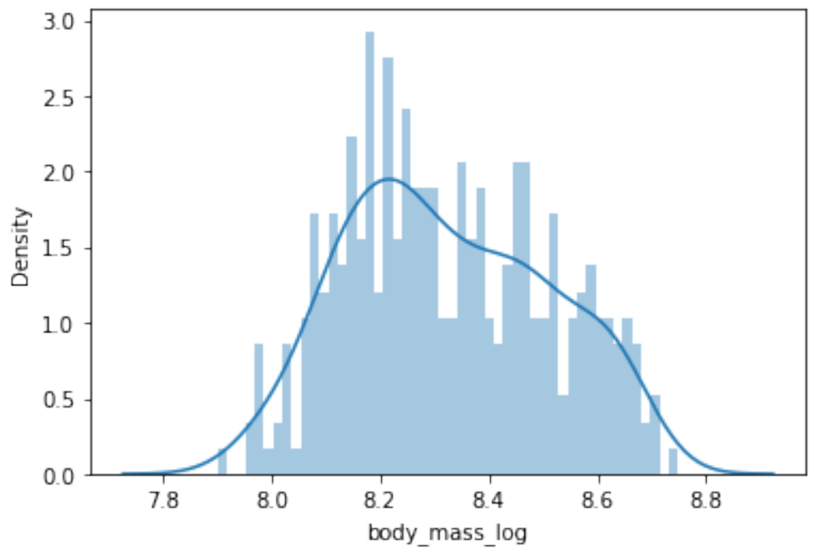
```
/Users/nafisahmedmunim/opt/anaconda3/lib/python3.9/site-packages/seaborn/distr
ibutions.py:2619: FutureWarning: `distplot` is a deprecated function and will
be removed in a future version. Please adapt your code to use either `displot`
(a figure-level function with similar flexibility) or `histplot` (an axes-leve
l function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[83]:
```
<AxesSubplot:xlabel='body_mass_g', ylabel='Density'>
```



In [84]:
```
sns.distplot( a=log_data["body_mass_log"], hist=True, kde=True, rug=False,bin
/Users/nafisahmedmunim/opt/anaconda3/lib/python3.9/site-packages/seaborn/distr
ibutions.py:2619: FutureWarning: `distplot` is a deprecated function and will
be removed in a future version. Please adapt your code to use either `displot`
(a figure-level function with similar flexibility) or `histplot` (an axes-leve
l function for histograms).
  warnings.warn(msg, FutureWarning)
```

Out[84]:
```
<AxesSubplot:xlabel='body_mass_log', ylabel='Density'>
```



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:

In [85]:
```
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
```

Out[85]:
```
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:

In [86]:
```
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()
```

Out[86]:

	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

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:

In [87]:

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

sums_data = grouped_data['culmen_length_mm', 'culmen_depth_mm'].sum().add_suf
avgs_data = grouped_data['culmen_length_mm', 'culmen_depth_mm'].mean().add_su

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

```
/var/folders/c5/1k4yqmhd029_nwyg7zh8lm9r0000gn/T/ipykernel_4217/3682604076.py:
3: FutureWarning: Indexing with multiple keys (implicitly converted to a tuple
of keys) will be deprecated, use a list instead.
  sums_data = grouped_data['culmen_length_mm', 'culmen_depth_mm'].sum().add_su
ffix('_sum')
/var/folders/c5/1k4yqmhd029_nwyg7zh8lm9r0000gn/T/ipykernel_4217/3682604076.py:
4: FutureWarning: Indexing with multiple keys (implicitly converted to a tuple
of keys) will be deprecated, use a list instead.
  avgs_data = grouped_data['culmen_length_mm', 'culmen_depth_mm'].mean().add_s
uffix('_mean')
```

Out[87]:

	culmen_length_mm_sum	culmen_depth_mm_sum	culmen_length_mm_mean	culme
species				
Adelie	5901.42193	2787.45117	38.825144	
Chinstrap	3320.70000	1252.60000	48.833824	
Gentoo	5842.52193	1844.25117	47.500178	

In [ ]: