MACHINE LEARNING WITH DECISION TREES AND RANDOM FORESTS

> by Dr Juan H Klopper

- Research Fellow
- School for Data Science and Computational Thinking
- Stellenbosch University

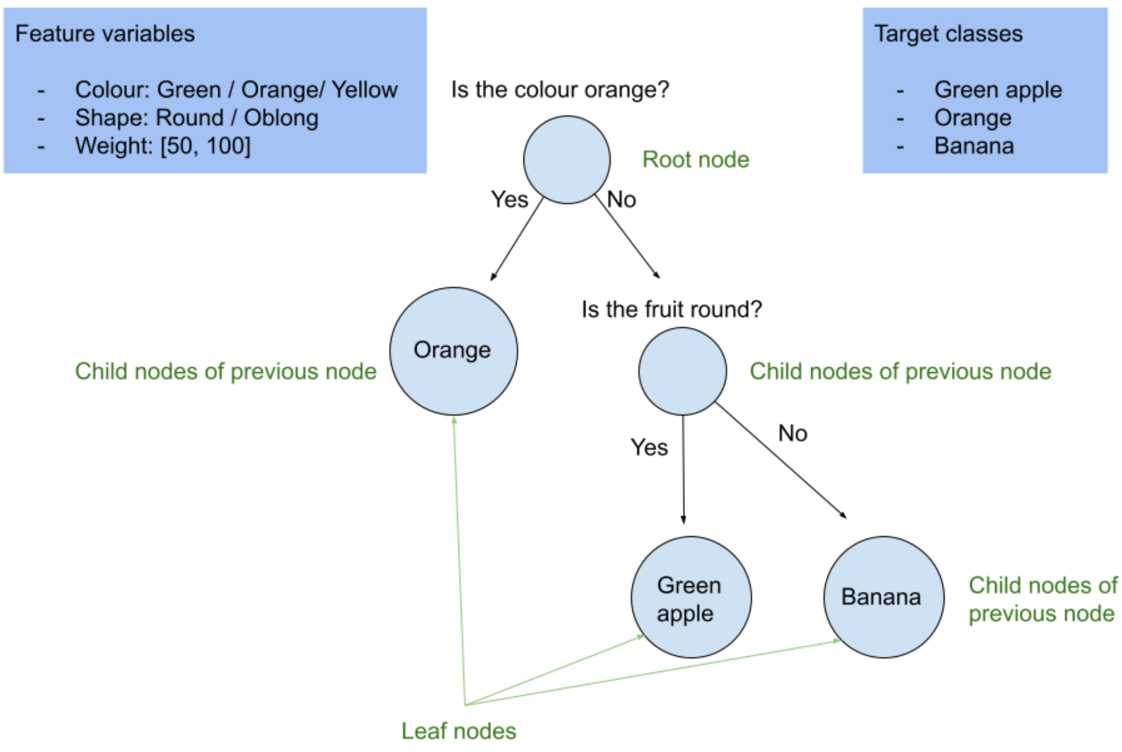
INTRODUCTION

Random Forests and **gradient boosted trees** are commonly used machine learning (ML) techniques used in classification and regression problems. They have the advantage over some other ML techniques in that the models are interpretable.

The basic building block of a random forest is a **decision tree**. The term decision tree is almost self-explanatory. The algorithm builds a tree structure by making repeated decisions on the data. As such, it is very similar to a flowchart.

In this notebook we explore a simple decision tree and take a closer look at random forests. We start with the concept of information gain, vital to random forests.

Imagine that we have a basket of green apples, oranges, and bananas. Without examining the basket, we have very little information. To gain more information we might consider if a fruit is orange in colour or not. This will immediately split the oranges from the green apples and the bananas. We have gained information. We see a simplified decision tree analgoue in the image below.



As the image shows, a decision tree asks questions at each **node**. The first question is the **root node**. All nodes that follow from a previous node are **child nodes** and the node from which it originated is a **parent node**. The last nodes are also termed **leaf nodes** or **terminal nodes**. A **branch** is any tree structure that *flows from* a parent node. The **depth** of a tree is longest path from the root node to a leaf. In the image above the depth is 2 (there are two layers below the root node on the right).

In our image above, the leaf nodes are **pure**. They only contain a single class. We have gained information by *asking our questions* and dividing the data set. We will se later that there are ways of calculating information gain.

Different questions could be asked of the data leading to different trees. In the image above, one of the feature variables (weight) was not even included.

Many trees can be generated together in an ensemble of trees. This leads to random forests and gradient boosted trees. Such algorithms can greatly improve on a simple decision tree.

PACKAGES USED IN THIS NOTEBOOK

```
# The usual suspects
```

```
import numpy as np
import pandas as pd
```

```
# The industry-leading scikit-learn package for machine learning in Python
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, export_g
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, cross_val_score, GridSearc
from sklearn import metrics
from sklearn.preprocessing import LabelEncoder, LabelBinarizer
import pydotplus
from IPython.display import Image
```

```
# Data visualisation
import plotly.graph_objects as go
import plotly.express as px
import plotly.figure_factory as ff
import plotly.io as pio
pio.templates.default = 'plotly_white'
```

```
# Two more data visualisation packages
import matplotlib.pyplot as plt
import seaborn as sns
```

```
%config InlineBackend.figure_format = "retina" # For Retina type displays
```

```
# Format tables printed to the screen (don't put this on the same line as the co
%load_ext google.colab.data_table
```

DECISION TREES

The knowledge we require to use random forests starts by understanding a decision tree. Below, we see a dataset with three categorical feature variables, each with three elements in its sample space. The target variable is dichotomous.

```
cat_1 = ['I', 'I', 'I', 'I', 'I', 'I', 'II', 'II', 'II', 'III', 'III', 'I', 'I',
cat_2 = ['A', 'A', 'A', 'B', 'C', 'C', 'A', 'A', 'B', 'B', 'C', 'A', 'B', 'B',
cat_3 = ['2', '2', '1', '2', '1', '1', '2', '1', '2', '1', '1', '2', '2', '2',
target = ['No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes

df = pd.DataFrame({
    'CAT1':cat_1,
    'CAT2':cat_2,
    'CAT3':cat_3,
    'Target':target
})

df
```

1 to 21 of 21 entries Filter ?

index	CAT1	CAT2	CAT3	Target
0	I	A	2	No
1	I	A	2	No
2	I	A	1	No
3	I	B	2	No
4	I	C	1	No
5	I	C	1	No
6	II	A	2	No
7	II	A	1	No
8	II	B	2	No
9	III	B	1	No
10	III	C	1	No
11	I	A	2	Yes
12	I	B	2	Yes
13	I	B	2	Yes
14	I	B	3	Yes
15	III	B	3	Yes
16	III	B	2	Yes
17	III	B	3	Yes
18	III	B	3	Yes
19	III	B	2	Yes
20	III	C	3	Yes

Show 25 per page

We can view the frequency of each variable's sample space elements.

```
df.CAT1.value_counts()
```

```
I      10
III     8
II      3
Name: CAT1, dtype: int64
```

```
df.CAT2.value_counts()
```

```
B    11
A     6
C     4
Name: CAT2, dtype: int64
```

```
df.CAT3.value_counts()
```

```
2    10
1     6
3     5
Name: CAT3, dtype: int64
```

```
df.Target.value_counts()
```

```
No      11
Yes     10
Name: Target, dtype: int64
```

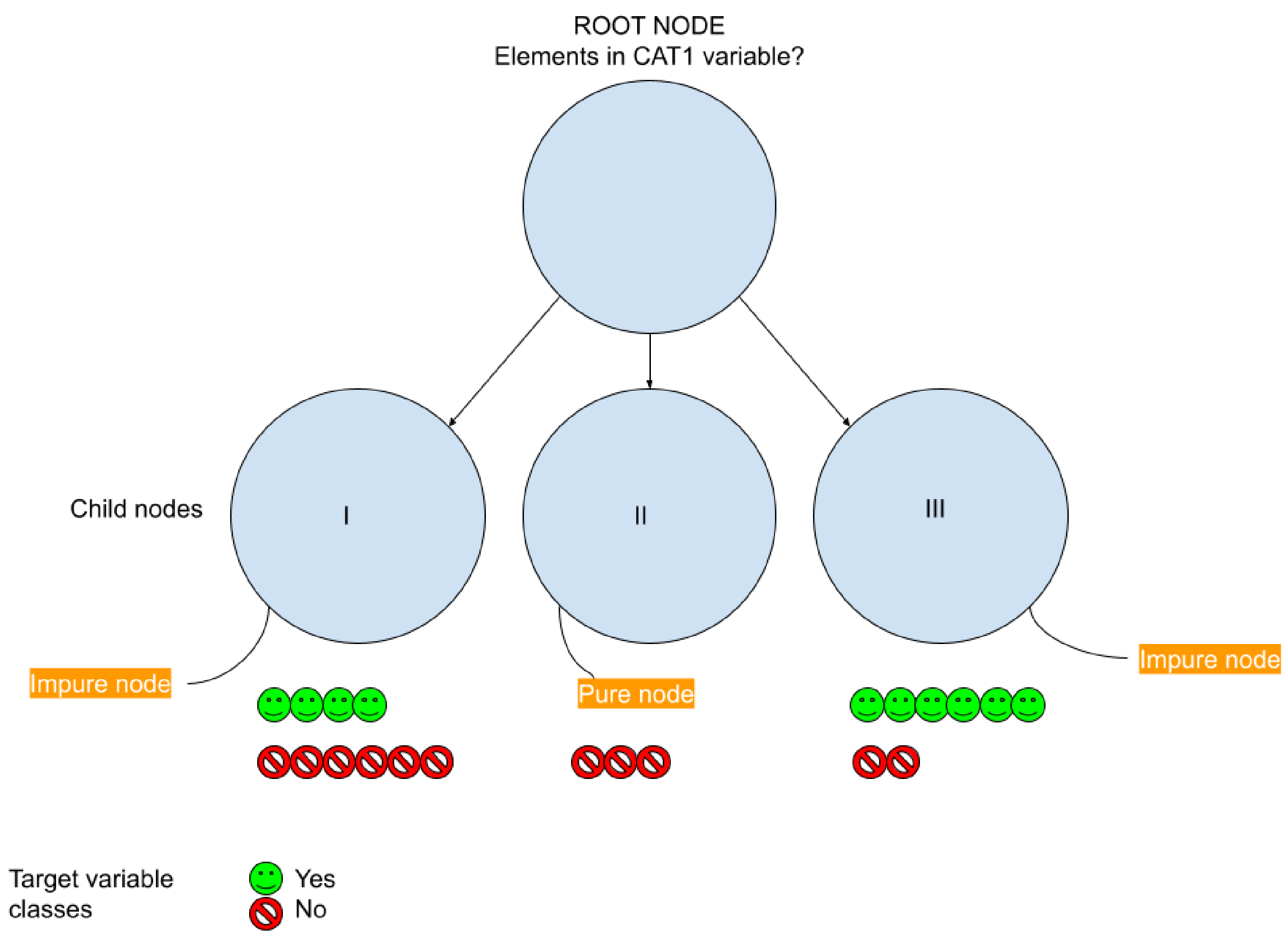
A decision tree is similar to a flowchart. Our aim is to build a decision tree to predict the target class.

We can make our root node any of the three feature variables. We shall start with the first categorical variable. Using the `groupby` method, we can see the proportion of target classes in each child node. There are three child nodes that follow from this root node as there are three sample space elements in the `CAT1` variable.

```
df.groupby('CAT1')['Target'].value_counts()
```

```
CAT1  Target
I     No        6
      Yes       4
II    No        3
III   Yes       6
      No        2
Name: Target, dtype: int64
```

The image below gives a visual representation of the results following from using `CAT1` as the root node.



We have been introduced to the term *pure node*. This means that there is also an *impure node*. **Purity** refers to class frequency. If a child node only contains a single class it is pure (as with the second child node), else it is impure.

With our aim that our decision three is knowing the target class, we already know that if we choose `CAT1` as our root node that a value of `II` will always predict a `No` for the target class. It is now a leaf or terminal node. Not so for the other two nodes (`CAT1` values of `I` and `III`). We need for them to *branch* further.

If we select `CAT2` for the firts child node on the left (`CAT1` being `I`) we get the following results (using `groupby` again, after selecting `I`).

```
df.loc[df.CAT1 == 'I'].groupby('CAT2')['Target'].value_counts()
```

```
CAT2  Target
A     No        3
      Yes       1
B     Yes       3
      No        1
C     No        2
Name: Target, dtype: int64
```

Now we see that for values of `C` we get a pure node. So, if `CAT1` is `I` and then `CAT2` is `C` then our decision tree predicts a target class of `No`. What about the other child node (`III`)? Below, we also choose `CAT2` for it.

```
df.loc[df.CAT1 == 'III'].groupby('CAT2')['Target'].value_counts()
```

```
CAT2  Target
B     Yes       5
      No        1
C     No        1
      Yes       1
Name: Target, dtype: int64
```

`CAT2` brings no pure nodes. So what if we choose `CAT3` instead?

```
df.loc[df.CAT1 == 'III'].groupby('CAT3')['Target'].value_counts()
```

```
CAT3  Target
1     No        2
2     Yes       2
3     Yes       4
Name: Target, dtype: int64
```

All three child nodes are pure.

We could carry on this process until all nodes are pure. This random selection might be very inefficient and the depth might be quite large. So, how do we improve on our selection of variables for our nodes? The answer is information gain.

INFORMATION GAIN

We have seen that any variable can be chosen at a node. Given the data, a decision tree must decide on these variables.

This decision is made using **information gain**. We require maximum information gain at each node. Information gain is the difference in information before and after a node. An equation for information gain in showed in (1).

$$\text{IG}\left(\text{D}_p, \text{f}\right) = \text{I}\left(\text{D}_p\right) - \sum_{i=1}^{m} \frac{N_i}{N} \text{I}\left(\text{D}_i\right) \tag{1}$$

Here IG is information gain given the data set of a parent node, D_p, and the feature, f. I is an impurity criterion (see below). N is the total number of samples and D_i is the data set of the i^{th} child node. The equation simply states that we subtract the averarge information from child nodes from that of their parent node

Two commonly used impurity criteria are the entropy and Gini index, shown in (2) and (3).

$$\text{I}_{\text{Entropy}} = - \sum_{i=1}^{c} p_i \log_2 (p_i) \tag{2}$$

$$\text{I}_{\text{Gini}} = 1 - \sum_{i=1}^{c} p_i^2 \tag{3}$$

Gini impurity can only be used for classification problems (categorical target variable). Here, p_i is the proportions of observations that belongs to class c for a particular node.

We will discuss entropy is more detail. It requires us to understand the logarithm function and summation notation.

As a quick reminder of the logarithm we have (4).

$$y = \log_2 (x) \text{ means } 2^y = x \tag{4}$$

The y (the solution we seek) is what we have to raise the base (2 is this case) to, to get x.

Σ is the summation symbol. It has a subscript and a superscript. The former tells us where to start counting and the latter is where we stop. The increment is 1. In (5) we get a look at how summation notation works for adding three numbers denoted as x_1, x_2, and x_3.

$$\sum_{i=1}^{3} (x_i) = x_1 + x_2 + x_3 \tag{5}$$

We simply increment the value of i at each step.

Back to our equation for entropy, (2). **Shannon entropy** is a measure of information. When we only have the data set and have not constructed a decision tree, our entropy (a measure of missing information) is high and our information is low. We need to gain information and decrease entropy (decrease the amount of missing knowledge about our target in this case). To understand this equation, we view our example from above.

We have two target classes, so i starts at 1 and goes to $c = 2$. From this we have p_1, the probability of say , `Yes` (we are free to choose), as the number of `Yes` classes at the first child node (`I`) divided by the total number of observations (`Yes + No`) of 10. So p_1 is 4 divided by 10. For $i = 2$, that is to say `No`, p_2 would be 6 divided by 10. The $\log_2$ is the logarithm base 2. For clarity, we have (6) that shows the entropy for the first child node in various ways. To remind us of the child nodes of the root node, we repeat the grouping again below.

```
df.groupby('CAT1')['Target'].value_counts()
```

```
CAT1  Target
I     No        6
      Yes       4
II    No        3
III   Yes       6
      No        2
Name: Target, dtype: int64
```

$$\begin{aligned} \text{I}_\text{I} &= -p_1 \log_2 p_1 - p_2 \log_2 p_2 \\ \text{I}_\text{I} &= -p_\text{Yes} \log_2 p_\text{Yes} - p_\text{No} \log_2 p_\text{No} \\ \text{I}_\text{I} &= -\frac{4}{10} \log_2 \frac{4}{10} - \frac{6}{10} \log_2 \frac{6}{10} \end{aligned} \tag{6}$$

The numpy `log2` function calculates the logarithm base 2.

```
cat1_I = -((4/10) * (np.log2(4/10))) - ((6/10) * (np.log2(6/10)))
cat1_I
```

```
0.9709505944546686
```

Below, we do this for the other two child nodes in the image above. Remember that in the second node (`CAT1 = II`) we have a pure node of three `No` classes. In the third node (`CAT1 = III`) we have six `Yes` classes and two `No` classes.

Since the logarithm of 0 is not defined, we do not include it in the equation.

```
# Second node
cat1_II = - ((3/3) * (np.log2(3/3)))
cat1_II
```

```
-0.0
```

The result is 0 (bar the rounding error).

```
# Third node
cat1 III = -((6/8) * (np.log2(6/8))) - ((2/8) * (np.log2(2/8)))
```

```
cat1_III
```

```
0.8112781244591328
```

Entropy ranges from 0 where we have complete information (a pure node) to 1 where we have no information.

We also look at the entropy of the parent node. At the root we simply have the frequency of the target classes.

```
df.Target.value_counts()
```

```
No     11
Yes    10
Name: Target, dtype: int64
```

```
# Root node
start = -((10/21) * (np.log2(10/21))) - ((11/21) * (np.log2(11/21)))
start
```

```
0.998363672593813
```

If we average over the entropy of each of the child nodes and subtract this average from the entropy of the root (parent) node, we know the information gain for choosing `CAT1` as our root node. This is the equation (1) above.

```
# Information gain given CAT1 as choice for root node
start - np.mean([cat1_I, cat1_II, cat1_III])
```

```
0.4042874329558792
```

Would it have been better to choose one of the other variables? We start by taking a look at the information gain from `CAT2` as root node.

```
df.groupby('CAT2').Target.value_counts()
```

```
CAT2  Target
A     No        5
      Yes       1
B     Yes       8
      No        3
C     No        3
      Yes       1
Name: Target, dtype: int64
```

```
# Calculating the three entropies
cat2_A = -((1/6) * np.log(1/6)) - ((5/6) * np.log(5/6))
cat2_B = -((8/11) * np.log(8/11)) - ((3/11) * np.log(3/11))
```

```
cat2_C = -((1/4) * np.log(1/4)) - ((3/4) * np.log(3/4))

start - np.mean([cat2_A, cat2_B, cat2_C])
```

```
0.4654140153309251
```

The information gain is higher. What about `CAT3`?

```
df.groupby('CAT3').Target.value_counts()
```

```
CAT3  Target
1     No        6
2     No        5
      Yes       5
3     Yes       5
Name: Target, dtype: int64
```

```
# Calculating the three entropies
cat3_1 = - ((6/6) * np.log(6/6))
cat3_2 = -((5/10) * np.log(5/10)) - ((5/10) * np.log(5/10))
cat3_3 = -((5/5) * np.log(5/5))

start - np.mean([cat3_1, cat3_2, cat3_3])
```

```
0.7673146124071646
```

The information gain is even higher. This would be the best choice for our first node.

This is one algorithm used by a decsion tree. It repeats this process at every branch until it reaches purity in all child nodes or until a hyperparameter setting requires it to stop branching (see later).

When can and should a decision tree stop? One obvious stopping criterium is when all the child nodes are leaves or terminal nodes, i.e. they are pure. This is a problematic approach as the depth can be large and the model will probably overfit the training data and not generalise well to unseen data.

In another method, we set a minimum information gain. Once successive branching fails to improve beyond this minimim, the decision tree terminates. We can also call a halt when a number of the child nodes contain less than a set proportion of the classes.

While a single decision tree is relatively easy to create and understand, it does have drawbacks. We have mentioned overfitting. This is worsened by smaller data sets. **Pruning** is a technique where the depth is made more shallow. This can be set or occur after a tree is fully constructed. Such pruned trees might do better on unseen data.

Another major drawback occurs when some feature variables contain many classes. A tree might preferentially split on this variable. **Information gain ratio** reduces the bias a tree has for these variables by looking at the size and number of branches of each variable.

In the next section, we use the `DecisionTreeClassifier` from the scikit-learn package to investigate our data set.

A DECISION TREE CLASSIFIER

The scikit-learn package provides a decision tree classifier for classification problems. We can use it on our simple data set. The process is as with the k nearest neighbour classifier from the previous notebook. First, we instantiate the classifier and then fit the data to it.

First, though, we have to transform our data. The `DecisionTreeClassifier` class only works with numerical data. We use the `LabelEncoder` and `LabelBinarizer` to transcode our variable values.

```
# Instantiate the label encoder
label_encoder = LabelEncoder()
```

```
# Instantiate the label binazier
label_binarizer = LabelBinarizer()
```

The `fit_transform` method for each of the encoders will fit and transform the data.

```
encoded_cat1 = label_encoder.fit_transform(cat_1)
encoded_cat2 = label_encoder.fit_transform(cat_2)
encoded_cat3 = label_encoder.fit_transform(cat_3)
y = label_binarizer.fit_transform(target).flatten()
```

Now we create a numpy array and append the three feature variables.

```
X = []

for i in range(len(y)):
  X.append([encoded_cat1[i], encoded_cat2[i], encoded_cat3[i]])
```

All that remains is to instantiate our classifier and fit the data.

```
# Instantiate the classifier
d_tree = DecisionTreeClassifier(criterion='entropy')
```

```
# Fit the data (in numpy array format)
d_tree.fit(
    X,
    y
)
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='entropy',
                       max_depth=None, max_features=None, max_leaf_nodes=None
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

If you are running this notebook on a local system then the following code will export a PNG image of the decision tree.

```
feature_names = ['Cat 1', 'Cat 2', 'Cat 3']
target_names = ['No', 'Yes']

dot_data = export_graphviz(
    d_tree,
    out_file=None,
    class_names=target_names
)

graph = pydotplus.graph_from_dot_data(dot_data)

Image(graph.create_png)
graph.write_png('tree.png')
```

We can use the `predict` method to pass an unseen observation to the model.

```
d_tree.predict([[1, 2, 1]])
```

```
array([0])
```

We can compute the accuracy of our model by passing the feature variable array to the `predict` method.

```
y_pred = d_tree.predict(X)
```

We use logic to return `True` and `False` values while comparing the predicted and the true target variable values. Since `True` is represented internally as a 1, we can sum over all the

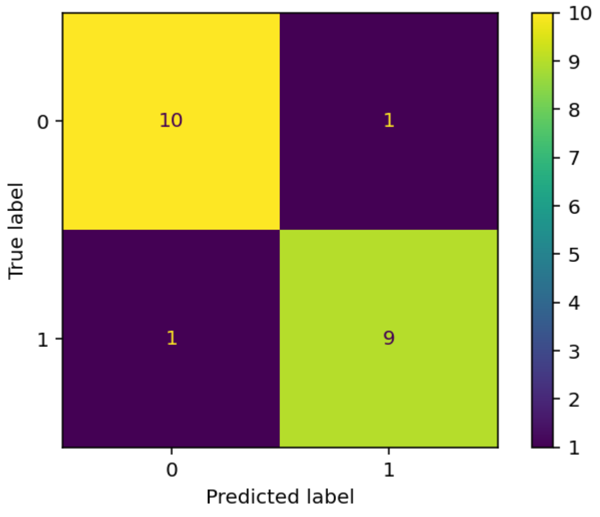
Boolean values and divide by the number of observations to return the accuracy of the model.

```
np.sum(y == y_pred) / len(y_pred)
```

```
0.9047619047619048
```

As with the k nearest neighbour classifier, we can use a confusion matrix plot to evaluate the model's prediction using the test data predictions and actual values.

```
metrics.plot_confusion_matrix(d_tree,
                              X,
                              y);
```



From this we can use all the other metrics described in the previous notebook.

A DECISION TREE REGRESSOR

The scikit-learn decision tree regressor class is very simular to the classifier class. In regression problems, the target variable is continuous numerical.

To work through an example of a decision tree regression problem, we generate data using the `make_regression` function from the models module of the scikit-learn package.

```
X, y = make_regression(
    n_samples=1000, # Sample size
    n_features=4, # Total number of feature variables
    n_informative=2, # Number of feature variable that are correlated to target
    noise=0.9, # Add noise to the data
    random_state=12 # For reproducible results
)
```

To visualise the correlation between every pair of variables we create a scatter plot matrix after importing the data into a pandas DataFrame object.

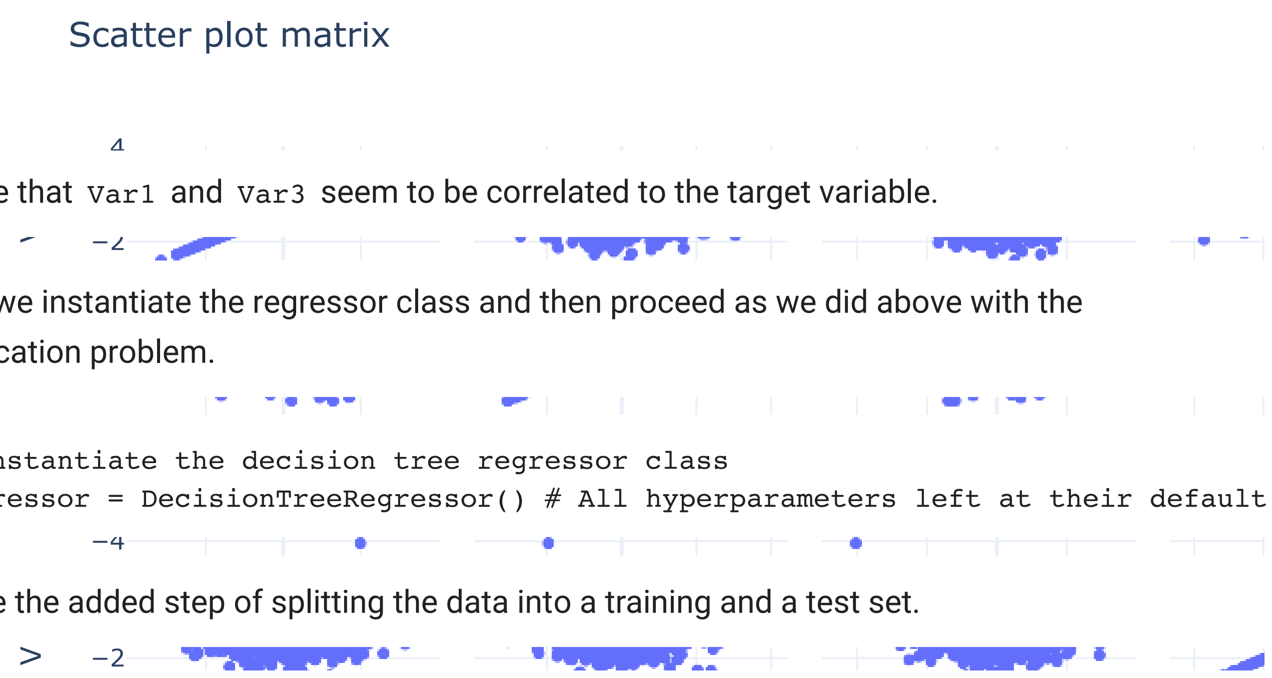
```
columns = ['Var1', 'Var2', 'Var3', 'Var4'] # Feature variable names

regression_data = pd.DataFrame(
    X,
    columns=columns
)

regression_data['Target'] = y # Add target variable as another column

regression_data[:5] # First five observations
```

1 to 5 of 5 entries Filter ?

index	Var1	Var2	Var3	Var4
0	-1.2157716266611218	-0.6655547376105355	-0.1295499652356899	0.7401926697773301
1	-1.4151748655098688	-0.6838234561651751	-0.4676109737502123	-0.11645777099964782
2	0.43115757594982945	-0.19440331160671367	-0.29922017229834025	0.35274382941066434
3	-0.29433533660223227	0.02199109438922098	0.19433672361768067	-2.4370673335566844
4	-1.4874862533107103	-0.4413808394553389	-0.26141173856816013	0.5377553102969255

Show 25 per page

```
px.scatter_matrix(
    regression_data,
    title='Scatter plot matrix'
)
```



We note that `Var1` and `Var3` seem to be correlated to the target variable.

Below, we instantiate the regressor class and then proceed as we did above with the classification problem.

```
# Instantiate the decision tree regressor class
regressor = DecisionTreeRegressor() # All hyperparameters left at their default
```

We take the added step of splitting the data into a training and a test set.

```
x_train, x_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=12
)
```

```
x_train.shape, y_train.shape # Verifying the splitting result
```

```
((800, 4), (800,))
```

Now we can fit the model and evaluate its performance.

```
dt_reg_model = regressor.fit(
    x_train,
    y_train
)
```

We can use the coefficent of determination, R^2, to evaluate the model given the test set.

```
dt_reg_model.score(
    x_test,
    y_test
)
```

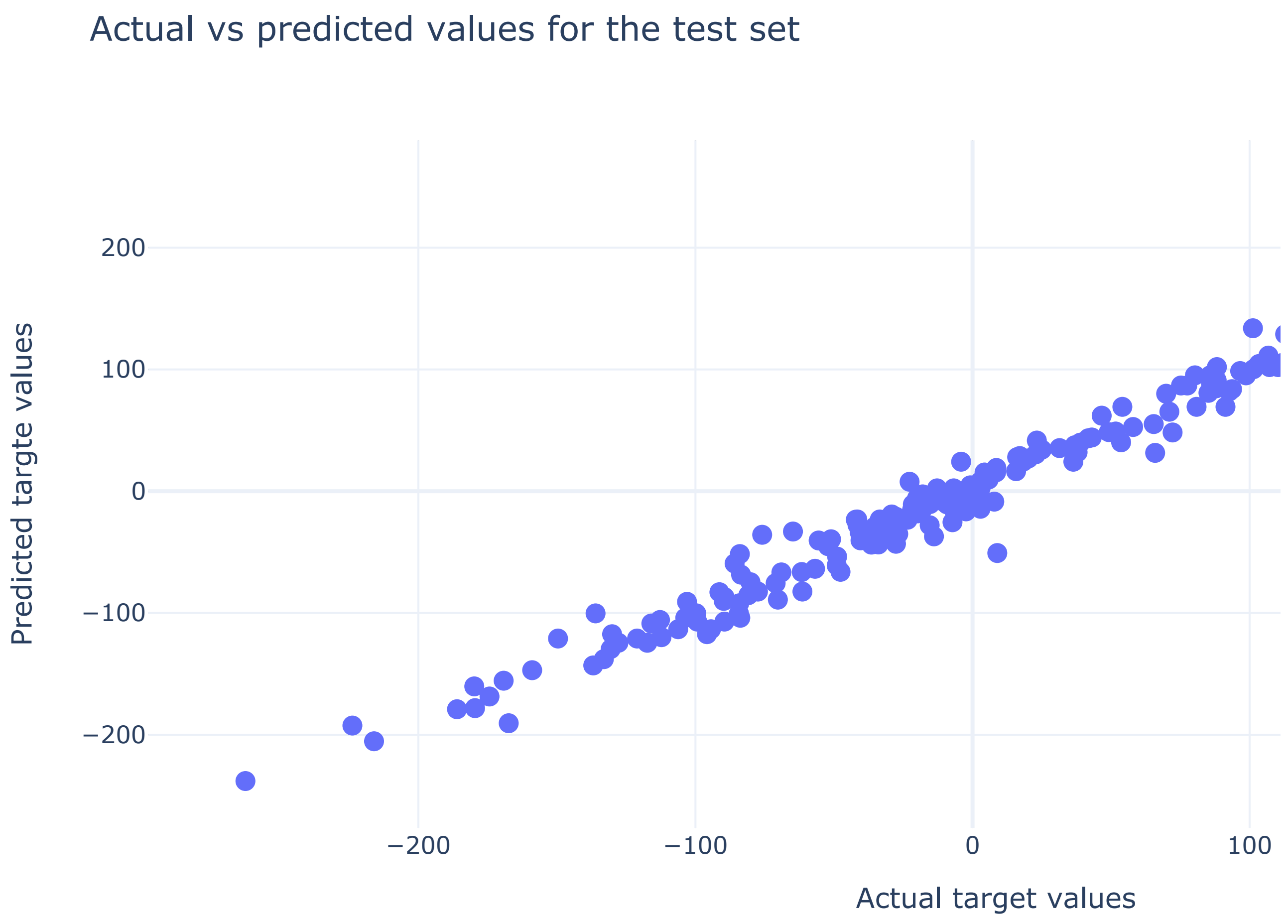
```
0.9761785339258129
```

We can calculate the predicted target values using the `predict` method.

```
y_reg_pred = dt_reg_model.predict(
    x_test
)
```

A scatter plot shows very good correlation between the actual and predicted target values.

```
go.Figure(
    go.Scatter(
        x=y_test,
        y=y_reg_pred,
        mode='markers',
        marker={
            'size':10
        }
    )
).update_layout(
    title='Actual vs predicted values for the test set',
    xaxis={'title':'Actual target values'},
    yaxis={'title':'Predicted targte values'}
)
```



RANDOM FORESTS

A single decision tree is easy to understand and to generate. It does not fare very well in the real world. To improve on the performance, we use ensemble techniques such a random forests. As the name suggests, it is a collection of trees.

The decision trees in a random forest are all individual models and the final result is majority vote or an average of all the predictions.

The trees themselves select a random set of the feature variables, a random sample of the observations (resampling with replacement), and are trained to various depths. These are all hyperpaarmeters that can be set. This combination can improve the performance on real-world data.

As an example, we will use the same data as with the decision tree regression problem above. The steps we take to generate, train, and evaluate the model should now be very familiar.

```
rf_regressor = RandomForestRegressor() # Hyperparameters are left at their defa
```

```
# Training the model
rf_reg_model = rf_regressor.fit(
    x_train,
    y_train
)
```

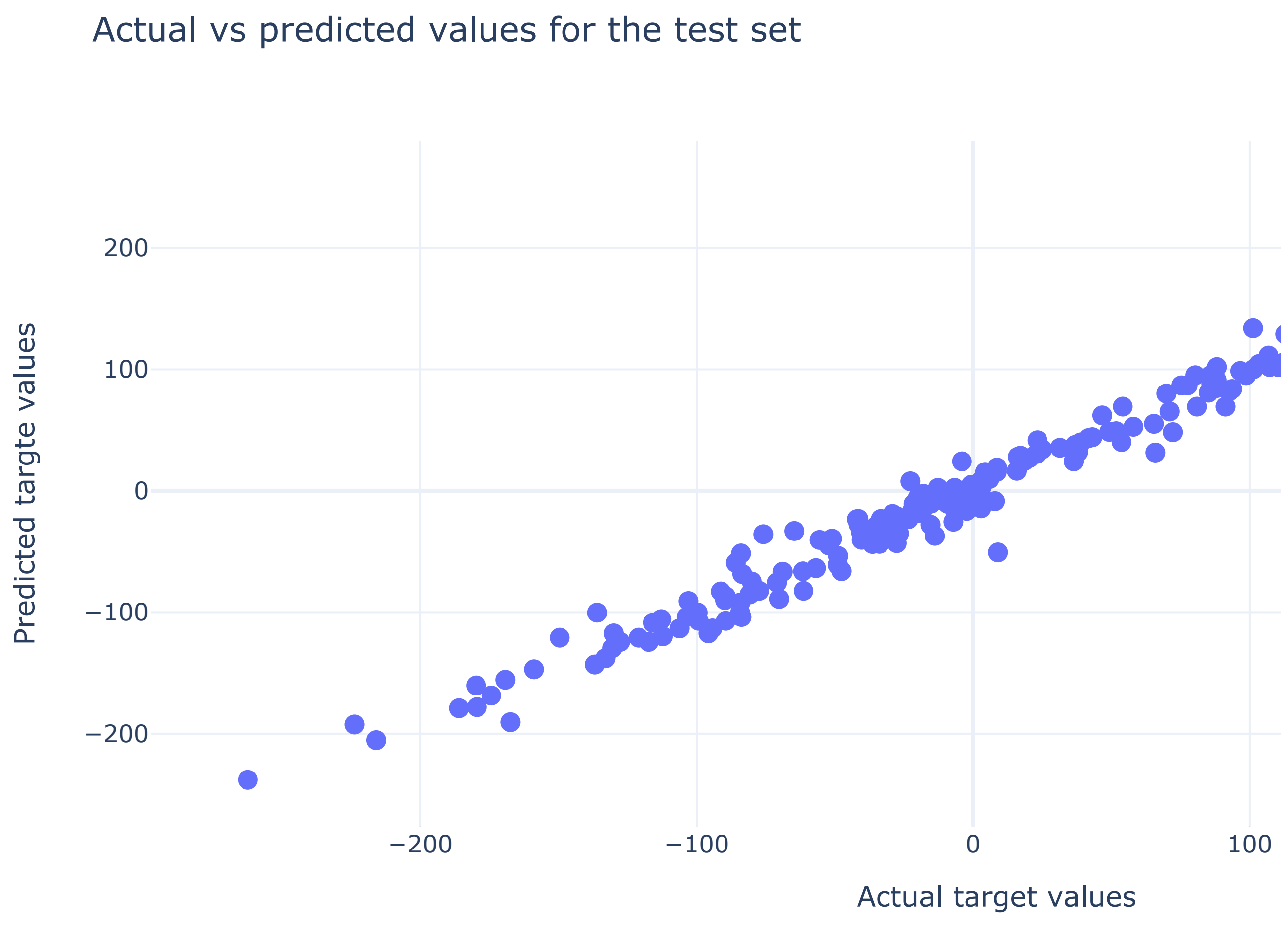
```
# Coefficent of correlation
rf_reg_model.score(
    x_test,
    y_test
)
```

```
0.9912287292944212
```

This is almost 1.0. Below, we look at a scatter plot of the actual versus the predicted values.

```
y_reg_pred = dt_reg_model.predict(
    x_test
)
```

```
go.Figure(
    go.Scatter(
        x=y_test,
        y=y_reg_pred,
        mode='markers',
        marker={
            'size':10
        }
    )
).update_layout(
    title='Actual vs predicted values for the test set',
    xaxis={'title':'Actual target values'},
    yaxis={'title':'Predicted targte values'}
)
```



TENSORFLOW DECISION FORESTS

Google, which provides the popular TensorFlow deep neural network architecture, now also provides a wrapper for the Yggdrasil Decision Forest C++ libraries named `tensorflow_decision_forests`. The name is a bit different from the traditional decion tree and random forest in that it combines both terms.

The tensorflow_decision_forests package can use numerical and categorical variables. We do not need to transform the data, i.e. standardize the numerical variables or convert categorical variables into numerical variables, except the target variable as it is used by the keras module of this package for metrics (see later). It can also manage missing data.

INSTALLING AND IMPORTING THE PACKAGE

At the time of the creation of this notebook, it is not yet part of Colab. We have to install it first.

```
# Install this package
!pip install tensorflow_decision_forests
```

```
Collecting tensorflow_decision_forests
  Downloading https://files.pythonhosted.org/packages/3a/34/10f20fe95d9882b82
     |████████████████████████████████| 6.2MB 2.9MB/s
Requirement already satisfied: tensorflow~=2.5 in /usr/local/lib/python3.7/di
Requirement already satisfied: wheel in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: absl-py in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: gast==0.4.0 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: h5py~=3.1.0 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/di
Requirement already satisfied: google-pasta~=0.2 in /usr/local/lib/python3.7/
Requirement already satisfied: astunparse~=1.6.3 in /usr/local/lib/python3.7/
Requirement already satisfied: keras-nightly~=2.5.0.dev in /usr/local/lib/pyt
Requirement already satisfied: wrapt~=1.12.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: grpcio~=1.34.0 in /usr/local/lib/python3.7/dis
Requirement already satisfied: tensorflow-estimator<2.6.0,>=2.5.0rc0 in /usr/
Requirement already satisfied: termcolor~=1.1.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: typing-extensions~=3.7.4 in /usr/local/lib/pyt
Requirement already satisfied: opt-einsum~=3.3.0 in /usr/local/lib/python3.7/
Requirement already satisfied: tensorboard~=2.5 in /usr/local/lib/python3.7/d
Requirement already satisfied: keras-preprocessing~=1.1.2 in /usr/local/lib/p
Requirement already satisfied: flatbuffers~=1.12.0 in /usr/local/lib/python3.
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/pytho
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: cached-property; python_version < "3.8" in /us
Requirement already satisfied: google-auth<2,>=1.6.3 in /usr/local/lib/python
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/li
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/di
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.7
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/pytho
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python
Requirement already satisfied: rsa<5,>=3.1.4; python_version >= "3.6" in /usr
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/pyt
Requirement already satisfied: importlib-metadata; python_version < "3.8" in
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /us
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-pac
Installing collected packages: tensorflow-decision-forests
Successfully installed tensorflow-decision-forests-0.1.7
```

We can now import the package, as well as some other packages we will need.

```
import tensorflow_decision_forests as tfdf
import tensorflow as tf
```

The tensorflow_decision_forests package is part of the TensorFlow family. Always check what the current version is for updates and changes.

```
# Current version
tfdf.__version__
```

```
'0.1.7'
```

LOADING A DATA SET

Along with the official tutorials by Google, we import the very famous Palmer penguins ML data set directly from the internet.

```
# Download the data set
!wget -q https://storage.googleapis.com/download.tensorflow.org/data/palmer_peng
```

Colab saves this as a temporary file that we can import using pandas.

```
penguins = pd.read_csv('/tmp/penguins.csv')
```

Below, we inspect the data set.

```
penguins.shape # Number of observations and variables
```

```
(344, 8)
```

This is a small data set with only 344 observations. There are eight variables.

```
penguins.info() # Variable data type and missing data information
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 344 entries, 0 to 343
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   species            344 non-null    object 
 1   island             344 non-null    object 
 2   bill_length_mm     342 non-null    float64
 3   bill_depth_mm      342 non-null    float64
 4   flipper_length_mm  342 non-null    float64
 5   body_mass_g        342 non-null    float64
 6   sex                333 non-null    object
 7   year               344 non-null    int64
dtypes: float64(4), int64(1), object(3)
memory usage: 21.6+ KB
```

We note a few missing values, especially for the `sex` variable.

```
penguins[:5] # First five observations
```

1 to 5 of 5 entries Filter ?

index	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	se
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	mal
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	fem
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	fem
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	Na
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	fem

Show 25 per page

There are three penguin species, with underrepresentation of the Chinstrap species.

```
penguins.species.value_counts()
```

```
Adelie       152
Gentoo       124
Chinstrap     68
Name: species, dtype: int64
```

In order to use the metrics, we need to transform the target variable to an integer data type.

```
classes = penguins.species.unique().tolist() # A list of the three species
classes
```

```
['Adelie', 'Gentoo', 'Chinstrap']
```

The `map` method and the `classes` list index is used to convert Adelie to 0, Gentoo to 1, and Chinstrap to 2.

```
penguins.species = penguins.species.map(classes.index)
```

DATA SPLITTING

Instead of using the train test split method from the scikit-learn package, we generate a function to split the data. We call the function `split`. It takes two arguments, `ds` from the DataFrame object, and `r` for the fraction of test set values. We set the default at 0.3 or 30% of the

Internal to the function we create a computer variable, `test_ind` to hold index values. To it we assign `True` and `False`. The effect is seen in the two code cells below.

The indices of the `True` values are used to generate the training set and for `false`, the test set.

```
def split(ds, r=0.3):
  test_ind = np.random.rand(len(ds)) < r

  return ds[~test_ind], ds[test_ind]
```

```
# Splitting the data
np.random.seed(12)
penguins_train, penguins_test = split(penguins)
```

We investiagte the number of observations in each set using the `shape` attribute of each DataFrame object to ensure that our split was executed correctly.

```
penguins_train.shape
```

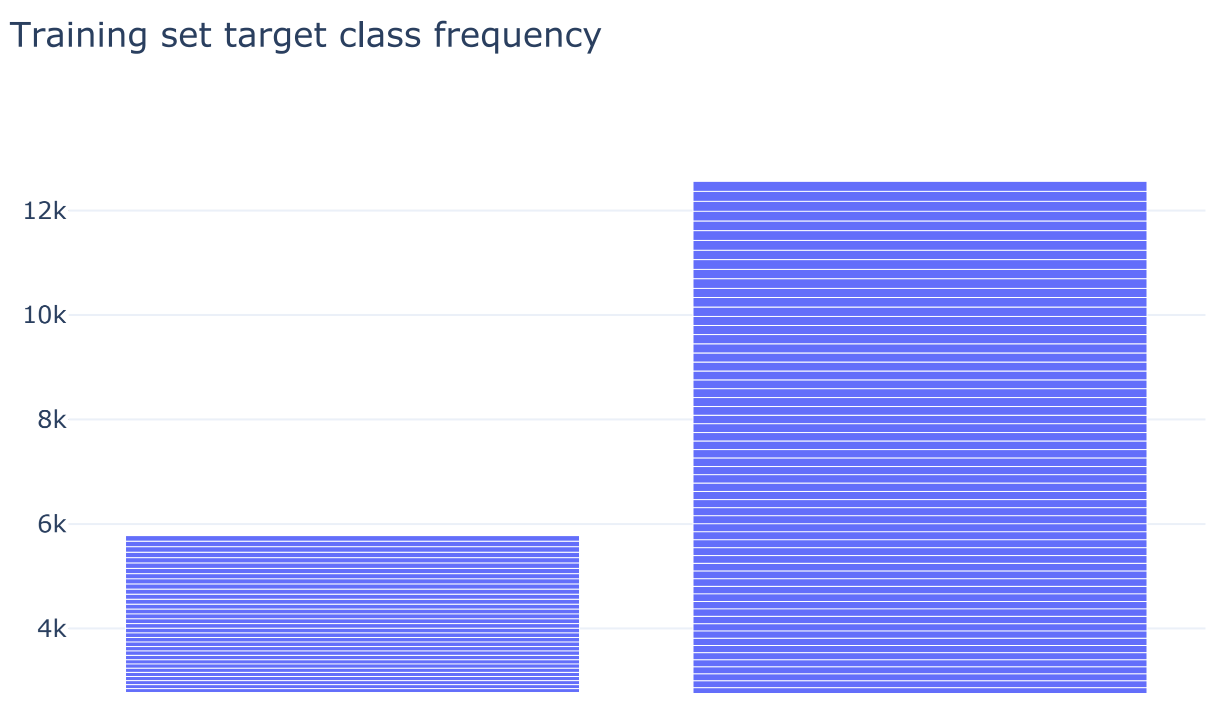
```
(237, 8)
```

```
penguins_test.shape
```

```
(107, 8)
```

We also need to make sure that we have proper representation of the target classes in each set.

```
px.bar(
    penguins_train,
    x='species',
    title='Training set target class frequency',
    labels={
        'species':'Species'
    }
).update_xaxes(
    type='category'
)
```



```
px.bar(
    penguins_test,
    x='species',
    title='Test set target class frequency',
    labels={
        'species':'Species'
    }
).update_xaxes(
    type='category'
)
```

Test set target class frequency

Finally, we need to transform the pandas dataframe objects to TensorFlow dataset objects using the `pd_dataframe_to_tf_dataset` function.

```
penguins_train = tfdf.keras.pd_dataframe_to_tf_dataset(
    penguins_train,
    label='species'
)

penguins_test = tfdf.keras.pd_dataframe_to_tf_dataset(
    penguins_test,
    label='species'
)
```

CREATING A DECISION FOREST MODEL

Below, we instantiate a random forest model, with default hyperparameter values.

```
rf_model = tfdf.keras.RandomForestModel()
```

We set accuracy as metric and compile the model. This step is only required if we want to specify metrics.

```
rf_model.compile(
    metrics=['accuracy']
)
```

TRAINING THE MODEL

Now we fit the data to the model.

```
rf_model.fit(
    x=penguins_train
)
```

```
4/4 [==============================] - 5s 3ms/step
<tensorflow.python.keras.callbacks.History at 0x7f4b8cf13ed0>
```

The `summary` function provides information about the model.

```
1 print(rf_model.summary())
```

```
            240 : body_mass_g [NUMERICAL]
            52 : sex [CATEGORICAL]
            10 : year [NUMERICAL]

    Condition type in nodes:
            1835 : HigherCondition
            357 : ContainsBitmapCondition
    Condition type in nodes with depth <= 0:
            294 : HigherCondition
            6 : ContainsBitmapCondition
    Condition type in nodes with depth <= 1:
            678 : HigherCondition
            185 : ContainsBitmapCondition

    Condition type in nodes with depth <= 2:
            1257 : HigherCondition
            273 : ContainsBitmapCondition
    Condition type in nodes with depth <= 3:
            1665 : HigherCondition
            331 : ContainsBitmapCondition
    Condition type in nodes with depth <= 5:
            1834 : HigherCondition
            357 : ContainsBitmapCondition
    Node format: NOT_SET

    Training OOB:
            trees: 1, Out-of-bag evaluation: accuracy:0.914634 logloss:3.0769
            trees: 11, Out-of-bag evaluation: accuracy:0.957447 logloss:0.52163
            trees: 21, Out-of-bag evaluation: accuracy:0.957806 logloss:0.235875
            trees: 31, Out-of-bag evaluation: accuracy:0.966245 logloss:0.099537
            trees: 41, Out-of-bag evaluation: accuracy:0.957806 logloss:0.093554
            trees: 51, Out-of-bag evaluation: accuracy:0.966245 logloss:0.091649
            trees: 61, Out-of-bag evaluation: accuracy:0.962025 logloss:0.090114
            trees: 71, Out-of-bag evaluation: accuracy:0.962025 logloss:0.092699
            trees: 81, Out-of-bag evaluation: accuracy:0.962025 logloss:0.093299
            trees: 91, Out-of-bag evaluation: accuracy:0.966245 logloss:0.091943
            trees: 101, Out-of-bag evaluation: accuracy:0.970464 logloss:0.09059
            trees: 111, Out-of-bag evaluation: accuracy:0.970464 logloss:0.09080
            trees: 121, Out-of-bag evaluation: accuracy:0.974684 logloss:0.09158
            trees: 131, Out-of-bag evaluation: accuracy:0.974684 logloss:0.09258
            trees: 141, Out-of-bag evaluation: accuracy:0.970464 logloss:0.08979
            trees: 151, Out-of-bag evaluation: accuracy:0.974684 logloss:0.08717
            trees: 161, Out-of-bag evaluation: accuracy:0.974684 logloss:0.08659
            trees: 171, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08669
            trees: 181, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08715
            trees: 191, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08789
            trees: 201, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08901
            trees: 211, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08965
            trees: 221, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08899
            trees: 231, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08888
            trees: 241, Out-of-bag evaluation: accuracy:0.983122 logloss:0.0892
            trees: 251, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08865
            trees: 261, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08828
            trees: 271, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08839
            trees: 281, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08852
            trees: 291, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08858
            trees: 300, Out-of-bag evaluation: accuracy:0.983122 logloss:0.08884

    None
```

EVALUATING THE MODEL

The `evaluate` method provides a loss and an accuracy value. Below, we evaluate the model using the test set.

```
evaluation = rf_model.evaluate(
    penguins_test,
    return_dict=True # Returning a dictionary of metrics
)
```

```
2/2 [==============================] - 0s 8ms/step - loss: 0.0000e+00 - accur
```

We can already see the loss and the accuracy. Below, we use the `keys` and the `values` methods to return the parts of the metrics dictionary.

```
evaluation.keys() # Metric dictionary keys
```

```
dict_keys(['loss', 'accuracy'])
```

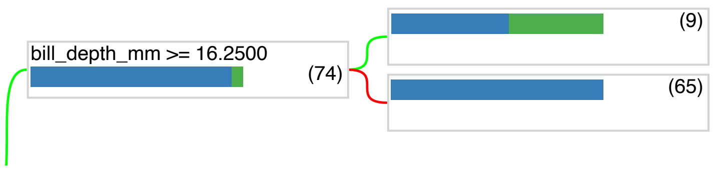
```
evaluation.values() # Metric values
```

```
dict_values([0.0, 1.0])
```

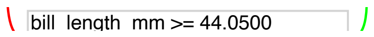
VISUALISING THE MODEL

As mentioned, decision trees and random forests are interpretable. Plotting the model shows us how information was gained.

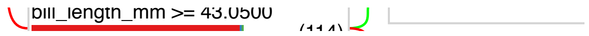
```
tfdf.model_plotter.plot_model_in_colab(
    rf_model,
    tree_idx=0,
    max_depth=4
)
```



The model chose `flipper_length_mm` as the first node and split on whether the length was equal to or more than 207. The first four decision node layers are shown.



We can inspect the feature variable importance using the `variable_importance` method.



```
rf_model.make_inspector().variable_importances()

    {'NUM_AS_ROOT': [("flipper_length_mm" (1; #3), 142.0),
      ("bill_length_mm" (1; #1), 88.0),
      ("bill_depth_mm" (1; #0), 62.0),
      ("island" (4; #4), 6.0),
      ("body_mass_g" (1; #2), 2.0)]}
```

The model can also provide a self-assessment. The `evaluation` method returns the number of samples and the accuraccy.

```
rf_model.make_inspector().evaluation()

    Evaluation(num_examples=237, accuracy=0.9831223628691983, loss=0.088848469708
```

CONCLUSION

Random forests and other ensemble techniques using decision trees have very recently gained a lot of attention. They are easier to interpret and perform better than state of the art deep neural networks in many cases.

✓ 0s completed at 14:11