Interpreting Decision Trees

Module 3 | Chapter 1 | Notebook 1

In addition to a model's prediction quality, it also plays a major role how easy it is to interpret the model. In this chapter you will learn methods that will allow you to interpret a model quantitatively. Let's start by looking at decision trees again, because these are a prime example of interpretability. We'll focus on this in the next two lessons.

By this end of this exercise you will have done the following:

- Visualized a decision tree.
- Interpreted the decisions of a decision tree.
- Learned the difference between local and global interpretation.

Visualizing a decision tree

Since this is so important, let's repeat it again. In the first two modules, you got to know some machine learning models and got an idea of how they work. We then used them to generate predictions. What was most important to us was the quality of the forecasts. In practice, the prediction quality is not the only important point. It's often even more important to understand how the predictions are made. We talk about a machine learning model's **interpretability**. If a model is easy to interpret, this has many advantages.

- Errors that are difficult to detect are then also easier to find (e.g. if the data is biased).
- Findings from the data are easier to generate. These in turn make it easier to recommend actions to be taken.
- It's possible to justify specific decisions of the model.
- A model can be more widely accepted.

Linear regression is another good example of an interpretable model. Every feature is assigned a weight, which allows us to judge exactly how important the feature is and how it contributes to individual predictions.

For classification, decision trees are an excellent example for interpretability. You learned about them in *Module 2, Chapter 1*. They use decision rules to divide data points into groups, which they then assign classes to. Decision trees can be displayed visually very well, thanks to their internal structures. This makes it even easier to understand their decisions. Now let's look at how you can display the decision tree.

Scenario: You work for an international global logistics company. Due to the tense situation on the labor market, the company is finding it increasingly difficult to attract new talent. For this

reason, management has decided to try and limit the number of employees who leave. You already developed a model that predicts which employees are likely to want to leave the company so that measures can be taken to encourage them to stay. Now the HR department not only wants to know who is leaving the company, but also why they are leaving. This enables the company to make individual arrangements in order to better bind the employees to the company.

You covered databases in *Employee Turnover, Module 2, Chapter 1*. They are located in the file *attrition_train.csv*. Import the data as a `DataFrame` named `df_train` and print the first 5 rows.

In [1]:
```
import pandas as pd
df_train = pd.read_csv('attrition_train.csv')
df_train.head()
```

Out[1]:

	attrition	age	gender	businesstravel	distancefromhome	education	joblevel	maritalstatus	mont
0	0	30	0	1	5.0	3	2	1	
1	1	33	0	1	5.0	3	1	0	
2	0	45	1	1	24.0	4	1	0	
3	0	28	1	1	15.0	2	1	1	
4	0	30	1	1	1.0	3	1	2	

5 rows × 21 columns

The code for that looks like this:

Column number	Column name	Type	Description
0	`'attrition'`	categorical	Whether the employee has left the company (`1`) or not (`0`)
1	`age`	continuous (`int`)	The person's age in years
2	`'gender'`	categorical (nominal, `int`)	Gender: male (`1`) or female (`0`)
3	`'businesstravel'`	categorical (ordinal, `int`)	How often the employee is on a business trip: often (`2`), rarely (`1`) or never (`0`)
4	`'distancefromhome'`	continuous (`int`)	Distance from home address to work address in kilometers
5	`'education'`	categorical (ordinal, `int`)	Level of education: doctorate (`5`), master (`4`), bachelor (`3`), apprenticeship(`2`), Secondary school qualifications (`1`)
6	`'joblevel'`	categorical (ordinal, `int`)	Level of responsibility: Executive (`5`), Manager (`4`), Team leader (`3`), Senior

Column number	Column name	Type	Description
			employee (`2`), Junior employee (`1`)
7	`'maritalstatus'`	categorical (nominal, `int`)	Marital status: married (`2`), divorced (`1`), single (`0`)
8	`'monthlyincome'`	continuous (`int`)	Gross monthly salary in EUR
9	`'numcompaniesworked'`	continuous (`int`)	The number of enterprises where the employee worked before their current position
10	`'over18'`	categorical (`int`)	Whether the employee is over 18 years of age (`1`) or not (`0`)
11	`'overtime'`	categorically (`int`)	Whether or not they have accumulated overtime in the past year (`1`) or not (`0`)
12	`'percentsalaryhike'`	continuous (`int`)	Salary increase in percent within the last twelve months
13	`'standardhours'`	continuous (`int`)	contractual working hours per two weeks
14	`'stock option levels'`	categorical (ordinal, `int`)	options on company shares: very many (`4`), many (`3`), few (`2`), very little (`1`), none (`0`)
15	`'trainingtimeslastyear'`	continuous (`int`)	Number of training courses taken in the last 12 months
16	`'totalworkingyears'`	continuous (`int`)	Number of years worked: Number of years on the job market and as an employee
17	`'years_atcompany'`	continuous (`int`)	Number of years at the current company Number of years in the current company
18	`'years_currentrole'`	continuous (`int`)	Number of years in the current position
19	`'years_lastpromotion'`	continuous (`int`)	Number of years since the last promotion
20	`'years_withmanager'`	continuous (`int`)	Number of years working with current manager

Each row in `df_train` represents an employee

In the last module we further processed the data by combining all the columns beginning with `'years_'` with a PCA and removing the columns `'over18'` and `'standardhours'`. The `'years_'` features are strongly correlated and the two removed columns only contain one constant value.

But this time we'll omit these processing steps. The constant columns have no influence on a decision tree because they cannot be used to separate the data. Combining features by reducing their dimensions makes it harder to interpret a decision tree. Because then we would

have to extrapolate back what the principal components stand for. Although the values are highly correlated, they may indicate different reasons for the individual.

Now create the target vector as `target_train`. Use the Column `'attrition'` of `df_train`. Then store the features (all columns apart from `'attrition'`) as `features_train`.

In [6]:
```
target_train = df_train.iloc[:,0]
features_train = df_train.iloc[:,1:]
```

To keep a decision tree easy to interpret, it's important that it isn't too deep. Instantiate the decision tree as a `model with a depth of 3.` `Specify` random_state=0 `and` class_weight='balanced'`. In the last module, we found that the target categories were unbalanced (typical for a employee turnover).

Then fit `model` to the training data.

In [7]:
```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(class_weight='balanced', max_depth=3, random_state=0)
model.fit(features_train, target_train)
```

Out[7]:
```
DecisionTreeClassifier(class_weight='balanced', max_depth=3, random_state=0)
```

In *Decision Trees, Module 2, Chapter 3*, you already visualized and interpreted the decision tree's decisions superficially, using `sklearn`. Now we'll expand our visualization techniques.

For this we need the `export_graphviz` function from `sklearn.tree`. The return value of this function is a *string* which represents the tree in Graphviz's `Dot` format. Graphviz is piece of an open source software for visualizing network graphs and flowcharts (link to documentation).

Now import the `export_graphviz` function. Then use it to create a *string* from `model`.

`export_graphviz` uses the following parameters, among others:

```
export_graphviz(decision_tree=object,           # a fitted decision tree
classifier
                out_file=object,                # handle or name of the
output file
                feature_names=[`list` of `str`],# names of the features
                class_names=[`list` of `str`],  # names of the target
variable
                filled=bool,                   # colors the nodes
                impurity=bool                   # displays the criterion
value
               )
```

You can find other parameters in the documentation.

If you don't assign anything to the `out_file` parameter, you will get the string directly as the output. Store it as `tree_string`. You can improve the readability of the image by assigning

the feature and class names. With `filled=True` and `impurity=True` the nodes of the tree are colored and the *gini impurity* is displayed.

In [15]:
```
from sklearn.tree import export_graphviz
tree_string = export_graphviz(decision_tree=model,           # a fitted decision tree
                feature_names=df_train.columns[1:],# names of the features
                class_names='attrition',  # names of the target variable
                filled=True,                    # colors the nodes
                impurity=True                    # displays the criterion value
               )
```

Out[15]:
```
Index(['age', 'gender', 'businesstravel', 'distancefromhome', 'education',
       'joblevel', 'maritalstatus', 'monthlyincome', 'numcompaniesworked',
       'over18', 'overtime', 'percentsalaryhike', 'standardhours',
       'stockoptionlevels', 'trainingtimeslastyear', 'totalworkingyears',
       'years_atcompany', 'years_currentrole', 'years_lastpromotion',
       'years_withmanager'],
      dtype='object')
```

Now we have the string called `tree_string`, which contains the decision tree in *Dot* format. In order to create an image from it using Graphviz, we need the module `pydotplus`. It links Python with Graphviz. We only need the `graph_from_dot_data` function. Import the new function directly.

In [16]:
```
from pydotplus import graph_from_dot_data
```

Now use `graph_from_dot_data()` on `tree_string`. Store the result in the variable `graph`.

In [17]:
```
graph = graph_from_dot_data(tree_string)
```

`graph` now has the data type `pydotplus.graphviz.Dot`. This data type gives us the possibility of saving the decision tree as an image. Use the `graph.write_png()` method. Pass it the file name you want to give the image (e.g.: `'decision_tree.png'`).

In [19]:
```
graph.write_png('decision_tree.png')
```

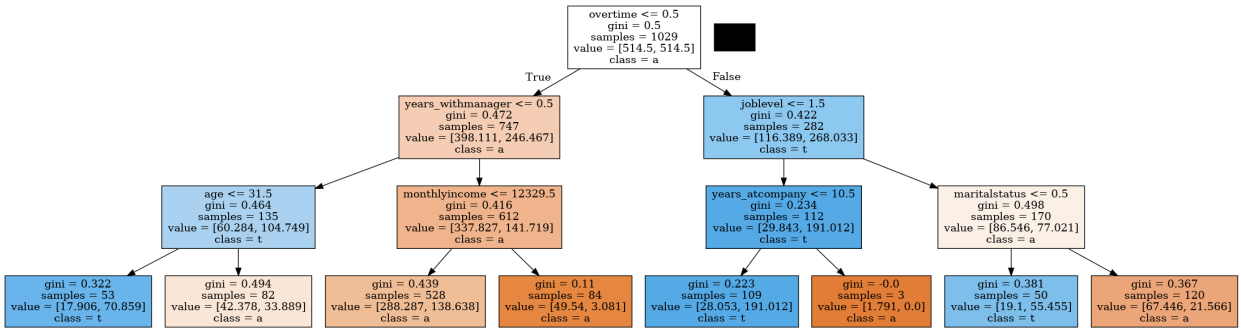
Out[19]:
```
True
```

You can also use other common image formats, as well as PNG. You can find the `pydotplus` documentation here.

Run the following code cell to display the decision tree in the DataLab. In the cell, we import the `Image` function from `IPython.display`. This function helps us to display images directly within a Jupyter environment. We pass it the picture's file name. Replace the name with the name you used.

In [20]:
```
from IPython.display import Image
Image('decision_tree.png')
```

Out[20]:



The decision tree should look like this:

"Decision Tree"

Now we have a good overview of the decision rules in the tree. In *Decision Trees, Module 2, Chapter 3*, We already learned what is represented in the decision tree. However, it won't hurt to do a brief recap of this now.

Let's start with the names of the different nodes in the tree. The first node is called the root or root node. The root doesn't have any incoming branches, it only has them coming out of it (displayed as arrows here). Nodes that have both incoming and outgoing branches are called internal nodes. The last nodes at the tip of the tree are called leaf nodes.

With each decision rule, the decision tree tries to minimize the *Gini impurity* of the resulting nodes. The *Gini impurity* corresponds to the probability of misclassifying a data point within the node if this is done randomly based only on the frequency and weighting of the classes in the node. The lower the *Gini impurity*, the lower the percentage of incorrectly classified data points.

The root and internal nodes contain 5 specifications. These are in the order of appearance:

1. The **decision rule** to then split the node into the two nodes below it. The decision rule is made up of a comparison of a feature and a value. This comparison is performed for each data point in the node. In our decision tree the first rule is `overtime<=0.5`. `overtime` is a binary feature. The tree first asks whether the employee has accumulated overtime (`overtime=1`) or not (`overtime=0`).
2. The **contamination criterion**, indicating the purity of the node. In our case that is the *Gini impurity*. The *Gini impurity* in the root is `0.272`. There is therefore a 50% probability of misclassifying a data point. Maybe you would expect a lower value because the target categories are unbalanced. However, with `class_weight='balanced'`, we adjusted the weighting of the classes according to their frequencies. After the first separation from the root, the *Gini impurity* improves to `gini=0.472` in the left internal node and to `gini=0.422` in the right internal node. As you can see, the nodes are colored because we used `filled=True` when drawing. The intensity of the color depends on the *Gini Impurity*. The node color is more intense, the purer the data points of the node are.

3. The **number of data points** that ended up in this node following the decision. The root contains all the data points in the data set (`samples=1029`). Data points for which the decision rule is true go along the left branches. So 747 employees have not accumulated any overtime (number in the left internal node). For 282 employees, the first decision rule was evaluated as incorrect. They are assigned to the right internal node because they accumulated overtime.
4. The **number of data points for each class**. In *Decision Trees, Module 2, Chapter 3* you saw that the sum of the values in `value` gives the value in `samples` . Here we notice that this is not the case. In particular, we have floating point numbers in `value` , so it can't just be the number. This is because when we change the class weights in the decision tree, the values in `value` are also adjusted accordingly.
5. The **predicted class** for data points within the node. The predicted class is the class with the largest weighted number of data points in the node. If we were only to consider people who accumulated overtime, those without overtime would be classified as remaining (orange, `attrition=0`). People with accumulated overtime would then be classified as leaving the company (blue, `attrition=1`).

If we get new data points, we can now follow the decision rules of the tree and see which decision rules lead to the prediction. So we can understand the predictions of the tree very well and that makes it so easy to interpret.

What conclusions could the HR department draw from this decision tree? Have another look at the tree.

The first decision rule checks whether the person has accumulated overtime. It seems that people who work a lot of overtime are more likely to leave the company. This could be because employees feel too stressed with a lot of overtime and therefore prefer to look for another job.

If we follow the right-hand branch, we see that especially people with a lot of overtime (`overtime>0.5`) and a junior position (`job level<1.5`) change employers. They might either feel that they are not valued enough because they have no prospect of promotion, despite working overtime. It could also be that the bond with the employer is too weak. If the employees have a higher position, they are less likely to leave the company despite doing overtime. They appear to be more rooted in the company, particularly when they are married (`marital status>0.5`). This may be because they don't want to have to move somewhere else with a partner and potential family.

If we follow the left branch (`overtime<0.5`), we can see that apparently employees who haven't been working with their supervisor for very long (`years_withmanager<0.5`) are more willing to leave the company. This could indicate that the superior's behaviour may not always be ideal. Then younger people in particular (`age<31.5`) change employers more frequently. Maybe they hired a manger who behaves like a bull in a china shop. But perhaps it is also the case that the relationship with their superior is particularly important for younger people.

However, we should also always consider the `gini` value for interpretations. For the "no overtime, no new supervisor and high income" branch , we can assume with relative certainty that these people will stay at the company (`gini=0.11`). However, with the "no overtime, new supervisor and older than 31" branch, this decision is not much more certain (`gini=0.494`) than if we don't follow any rules at all (`gini=0.5`).

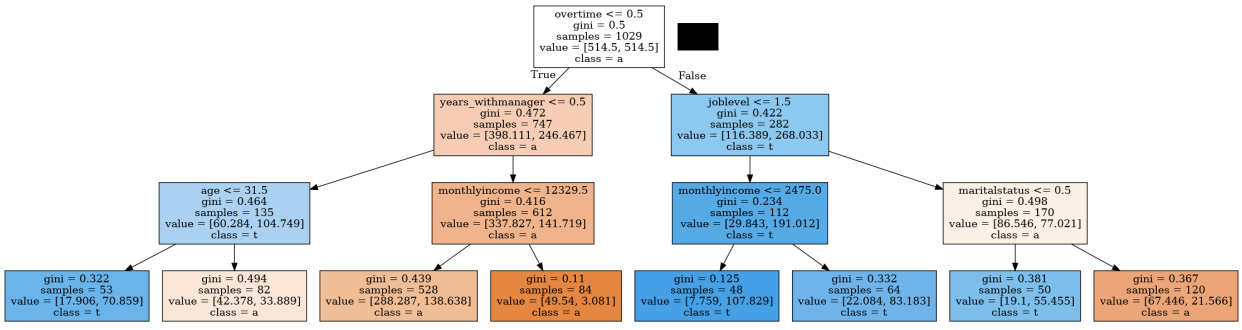
In the last level, the leaf with `gini=0` is particularly striking. Even if it looks like a very certain decision, there are only three people in this node. We can prevent leaves with a very small number of people by increasing the `min_samples_leaf` parameter in the decision tree.

What does our decision tree look like when we set `min_samples_leaf=20` ? Go through the following steps:

- Instantiate a new `DecisionTreeClassifier` with the parameters `class_weight='balanced', max_depth=3, random_state=0, min_samples_leaf=20` . Call it `model` again. We no longer need the previous decision tree.
- Fit the new decision tree to the training data.
- Use the `export_graphviz` function to create a string from the decision tree.
- Pass the new string to `graph_from_dot_data()` .
- Save the decision tree as an image.
- Finally, display the image using the `Image` function.

In [23]:
```
model  = DecisionTreeClassifier(class_weight='balanced', max_depth=3, random_state=0,
model.fit(features_train, target_train)

tree_string = export_graphviz(decision_tree=model,  # a fitted decision tree classifie
                feature_names=df_train.columns[1:], # names of the features
                class_names='attrition',            # names of the target variable
                filled=True,                        # colors the nodes
                impurity=True                       # displays the criterion value
               )

graph = graph_from_dot_data(tree_string)

graph.write_png('decision_tree.png')

Image('decision_tree.png')
```

Out[23]:



The result should look like this:

The decision rule `years_atcompany <= 10.5`, which resulted in a pure sheet with only 3 data points, has now been replaced by the `monthlyincome <= 2475.0` rule. It results in a relatively clean leaf with `gini = 0.125` and 48 data points. It enables a further insight: When employees accumulate overtime, it is not just relevant whether they hold a junior position or not, but their income also plays a major role. If someone earns very little, perhaps a promotion isn't the best way to keep the person. A salary adjustment could have a very similar effect.

Congratulations: You've visualized a decision tree from `sklearn`! You are therefore well equipped to back up recommendations for action with a model. The HR department is very pleased with the visualization. It makes it easy for them to understand the model's decisions.

Local interpreting of the decision tree

As you've already learned, we speak of local interpretability when you can say for individual data points why they were assigned to a class. The decision tree is perfectly suited for this.

The HR department comes back to you. This time they have a small list of personal data with them and ask you for your assessment of whether these people want to leave the company. The data is stored in the file *employee_data.csv*. Import it and print the contents.

In [24]:
```
df_train = pd.read_csv('employee_data.csv')
df_train.head()
```

Out[24]:

	overtime	joblevel	monthlyincome	maritalstatus	years_withmanager	age
0	1	1	2439.0	0	0.0	29
1	0	1	2083.0	1	0.0	30
2	1	2	5228.0	2	9.0	32

How do you assess whether these employees want to leave the company? Remember that the *Gini-Impurity* of the leaves can be interpreted as the probability of a misclassification. You don't need to program anything, just follow the branches of your last decision tree.

Tip: Use the following code cell to summarize your conclusion as a comment.

In [25]:
```
Has accumulated overtime and a holds junior position with very low income. This means
```

```
  Input In [25]
    Has accumulated overtime and a holds junior position with very low income. This m
eans that this person would very probably (approx. 87%) leave the company. A salary i
ncrease can reduce this probability.
         ^
SyntaxError: invalid syntax
```

Congratulations: The HR department is very grateful! You have made a very difficult task easier thanks to your easily interpretable model. Next we'll turn to global interpretation of the decision tree. In particular we'll learn how to calculate the importance of features in the decision tree.

Remember:

- To visualize a decision tree with `sklearn`, you need the following functions:
  - `export_graphviz` from `sklearn.tree` to convert the decision tree to Graphviz dot format and return it as `str`.
  - `graph_from_dot_data` from `pydotplus` to create an image file.
- With global interpretation you want to understand how important individual features are for the model's prediction.
- With local interpretation, you want to understand why certain data points received the prediction they got.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.
