Decision Trees

Module 2 | Chapter 3 | Notebook 4

In this notebook we'll turn to a new classification algorithm. Decision trees. These use binary decision rules similar to yes/no questions to predict categories. By the end of this lesson you will be able to:

- How to instantiate a decision tree model
- How a decision tree ends up with its category predictions
- How to simplify a decision tree

Your first decision tree

Scenario: You work for an international global logistics company, which wants to limit the number of existing employees who leave the company. You should predict which employees are likely to want to leave the company so that measures can be taken to encourage them to stay.

Import the pipeline and column names we created in the previous programming exercise as objects called `pipeline` (pipeline object) and `col_names` (list with column names). They are located in the files *pipeline.p* and *col_names.p*.

Tip: If you don't have the files, then run the last three cells of the last notebook.

In [1]:
```
import pickle
#load pipeline
pipeline = pickle.load(open("pipeline.p",'rb'))
col_names = pickle.load(open("col_names.p",'rb'))
```

Now you can use the pipeline to transform the training and test data. First, import the data from *attrition_train.csv* (training data) and *attrition_test.csv* (test data) with `pandas`, split `features` and `target` from each other and transform both the features data sets with `pipeline.transform()`. The outputs of these methods can be converted into a `DataFrame` with `pd.DataFrame()`. When creating these DataFrames, pass the parameter `columns = col_names` to set the column names directly. Store the training data as `features_train` and `target_train` and the test data accordingly as `features_test` and `target_test`.

Then print the first 5 rows of `features_train`.

In [2]:
```
import pandas as pd

#gather data
df_train = pd.read_csv('attrition_train.csv')
df_test = pd.read_csv('attrition_test.csv')
```

```
#extract features and target
features_train = df_train.drop('attrition', axis=1)
features_test = df_test.drop('attrition', axis=1)

target_train = df_train.loc[:,'attrition']
target_test = df_test.loc[:,'attrition']

#transform data
features_train = pd.DataFrame(pipeline.transform(features_train), columns=col_names) #
features_test = pd.DataFrame(pipeline.transform(features_test), columns=col_names)

# look at raw data
features_train.head()
```

Out[2]:

	pca_years_0	pca_years_1	age	gender	businesstravel	distancefromhome	education	joblevel	ma
0	0.385171	-0.156575	30.0	0.0	1.0	5.0	3.0	2.0	
1	-2.348248	-0.406330	33.0	0.0	1.0	5.0	3.0	1.0	
2	-0.781200	-0.233330	45.0	1.0	1.0	24.0	4.0	1.0	
3	-1.181156	-0.535303	28.0	1.0	1.0	15.0	2.0	1.0	
4	-1.447056	0.019780	30.0	1.0	1.0	1.0	3.0	1.0	

The code for that looks like this:

Column number	Column name	Type	Description
0	`'pca_years_0'`	continuous (`int`)	first principal component of the original columns `'totalworkingyears'` , `'years_atcompany'` , `'years_currentrole'` , `'years_lastpromotion'` and `'years_withmanager'`
1	`'pca_years_1'`	continuous (`int`)	second principal component of the original columns `'totalworkingyears'` , `'years_atcompany'` , `'years_currentrole'` , `'years_lastpromotion'` and `'years_withmanager'`
2	`'attrition'`	categorical	Whether the employee left the company (`1`) or not (`0`)
3	`'age'`	continuous (`int`)	The person's age in years
4	`'gender'`	categorical (nominal, `int`)	Gender: male (`1`) or female (`0`)
5	`'businesstravel'`	categorical (ordinal, `int`)	How often the employee is on a business trip: often (`2`), rarely (`1`) or never (`0`)

Column number	Column name	Type	Description
6	`'distancefromhome'`	continuous (`int`)	Distance from home address to work address in kilometers
7	`'education'`	categorical (ordinal, `int`)	Level of education: doctorate (`5`), master (`4`), bachelor (`3`), apprenticeship(`2`), Secondary school qualifications (`1`)
8	`'joblevel'`	categorical (ordinal, `int`)	Level of responsibility: Executive (`5`), Manager (`4`), Team leader (`3`), Senior employee (`2`), Junior employee (`1`)
9	`'maritalstatus'`	categorical (nominal, `int`)	Marital status: married (`2`), divorced (`1`), single (`0`)
10	`'monthlyincome'`	continuous (`int`)	Gross monthly salary in EUR
11	`'numcompaniesworked'`	continuous (`int`)	The number of enterprises where the employee worked before their current position
12	`'overtime'`	categorically (`int`)	Whether or not they have accumulated overtime in the past year (`1`) or not (`0`)
13	`'percentsalaryhike'`	continuous (`int`)	Salary increase in percent within the last twelve months
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

Each row in `df_train` represents an employee

Let's start by creating a very simple decision tree. To do this, import `DecisionTreeClassifier` directly from `sklearn.tree`.

In [3]:
```
from sklearn.tree import DecisionTreeClassifier
```

According to the official documentation we have some options for the hyperparameter settings. Here is a brief overview of the most important parameters:

```
DecisionTreeClassifier(criterion=str, # impurity measurement to minimize at each split
                       max_depth=int,         # restricts how consecutive splits are made in the tree
                       min_samples_split=int, # minimum observations that need to be left in a node to consider splitting
                       min_samples_leaf=int,  # minimum observations that need to be in the terminal nodes (leafs)
                       max_features=int,      # maximum number of features to consider at each split
```

```
                              random_state=int,       # set for reproducibility
                              max_leaf_nodes=int)     # max number of terminal
nodes (leaves)
```

In this case we'll limit ourselves to just `max_depth`. This parameter controls how many decision rules are used to classify data.

Instantiate `DecisionTreeClassifier` with `max_depth=1` so that the tree only uses one decision rule. Also use `random_state=0` so that our results are reproducible. Store the model in the variable `model_1`.

In [4]:
```
model_1 = DecisionTreeClassifier(max_depth=1, random_state=0)
```

By limiting the model to one decision rule, the model searches for the variable that best separates the categories (entries in `target_train`). A threshold value is then set for this variable, which predicts which category a data point is likely to belong to.

Next we can split the data into feature matrix and target vector. First of all, for the sake of easier understanding, we'll restrict ourselves to the continuous columns. Their names are summarized in `num_cols`:

In [5]:
```
num_cols = ['age',
            'distancefromhome',
            'monthlyincome',
            'numcompaniesworked',
            'percentsalaryhike',
            'trainingtimeslastyear',
            'pca_years_0',
            'pca_years_1']
```

Now overwrite the `features_train` variable by only selecting the numeric columns of `features_train`.

In [6]:
```
features_train = features_train.loc[:, num_cols]
```

Now we can fit the model to the data. Then the algorithm searches for a single decision rule which best separates the two `'attrition'` categories.

Tip: The data does not have to be standardized beforehand for a decision tree.

In [7]:
```
model_1.fit(features_train, target_train)
```

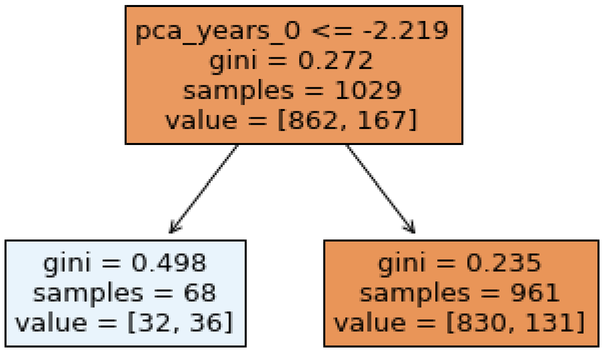
Out[7]:
```
DecisionTreeClassifier(max_depth=1, random_state=0)
```

The `my_model.tree_` attribute now contains the information about the decision rules used in the decision tree. To visualize a decision tree, use the `plot_tree()` function from the module `sklearn.tree`. You can pass this a decision tree model as well as a list of features and, if necessary, names for the target categories and get a simple visualization of the decision tree:

```
plot_tree(decision_tree=var, #decision tree model to be plotted
          feature_names=list, #names of each of the features
          class_names=list, #names of the target classes [class0 , class1,
...]
          filled=bool) #coloration of majority class for classification
```

Now use `plot_tree` and the column names of `features_train` to create a visualization.

In [8]:
```python
from sklearn.tree import plot_tree
plot_tree(decision_tree=model_1 ,
        feature_names=features_train.columns,
        filled=True);
```



You'll learn how to interpret the visualizations of decision trees for presentations in the context of data storytelling in *Module 3, Chapter 1*.

As you can be see here, it's usual to visualize the decision tree upside down: At the top you can see the root. Then the leaves come below the root. In the root you can see:

1. The first decision rule uses the feature `'pca_years_0'`. The threshold is `2.219`.
2. The *Gini impurity* in the root is `0.272`. This is the probability of misclassifying a data point based on this feature if all the `'attrition'` labels were distributed randomly. A *Gini impurity* of 0 would indicate that all the data points belong to the same class.
3. `samples` here indicates that `1029` data points are located in the root. This corresponds to the total number of data points in `features_train`.
4. `value`: There are `862` data points in the `0` category (didn't leave the company) and `167` data points in the `1` category (left the company).
5. The most likely category for the data points in the root as a whole: not leaving the company.

All data points that actually have a `'pca_years_0'` value of less than or equal to `2,219` take the left branch (`True`) and land immediately in the left hand leaf and are classified as "left the company". The rest ends up in the right-hand leaf (`False`) and is classified as "didn't leave the company". The leaves at the bottom of the decision tree represent the predicted categories when all decision rules have been gone through.

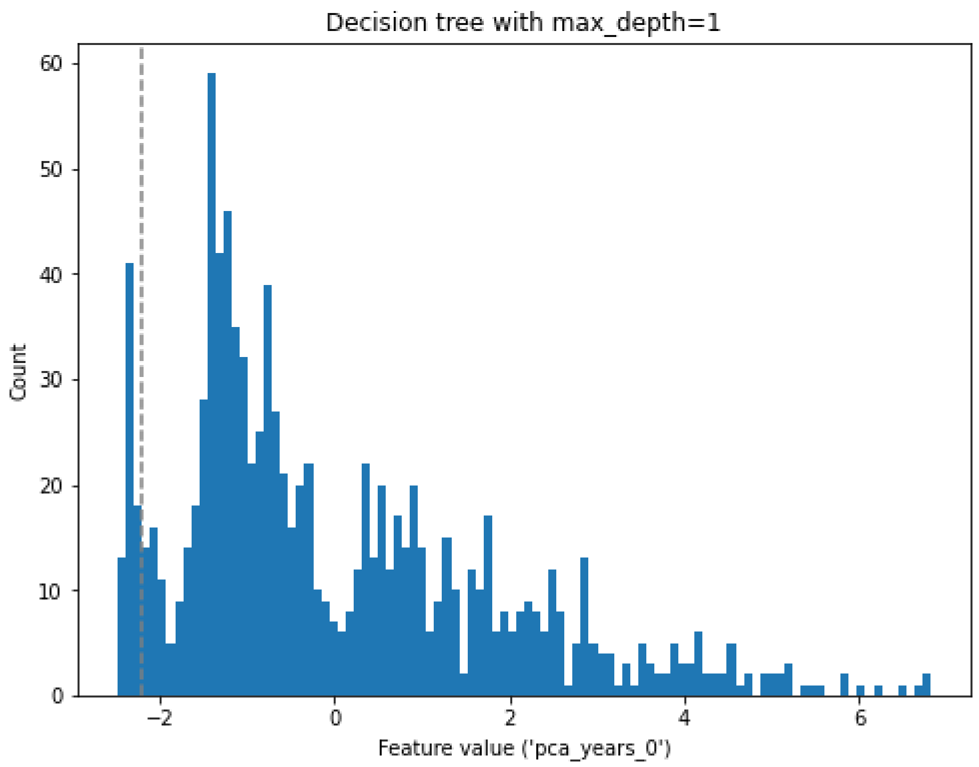
In the leaves, we now see the following: The left leaf contains only `68` data points (*samples* row), of which `32` correspond to the `0` category (didn't leave the company) and `36` to the

`1` category (left the company), see *value* row. Overall, the `1` category is predicted for all the data points in this leaf (left the company), see *class* row.

The right hand leaf contains the remaining `961` data points, which are all predicted as `0` category (didn't leave the company). For `830` data points this is actually correct, but for the remaining `131` data points in the leaf this categorization is wrong.

A decision rule can also be imagined as a decision line parallel to the axis. This would then look something like this:

In [9]:
```python
import matplotlib.pyplot as plt

# histogramm of first principal component
fig, ax = plt.subplots(figsize=(8,6))
features_train.loc[:, 'pca_years_0'].plot(kind='hist',
                                          bins=100,
                                          ax=ax)

# grey vertical line
ax.axvline(-2.219, # tree uses -2.219 as decision line
           ls='--',
           color = "grey")

# set title and label axes
ax.set(xlabel='Feature value (\'pca_years_0\')',
       ylabel='Count',
       title='Decision tree with max_depth=1');
```



All data points to the left of the grey decision line are predicted to leave the company. All the other data points are predicted to stay at the company.

One should keep in mind that a decision tree can only use axis-parallel decision rules to predict categories. A decision tree has no concept of an abstract neighborhood around data points like in the k-nearest-neighbors algorithm. The decision tree also doesn't learn any model coefficients for each feature, like logistic regression.

So if you want to predict a category for a new data point, you go through the decision rules and then take the predicted category of the leaf you land on. In our case, this is equivalent to looking to see if the `'pca_years_0'` value is less than or equal to -2.219 or not. If so, the model predicts that the employee will leave the company.

With the extremely simple decision tree in `model_1`, the `'pca_years_0'` value is the only one that counts. All the other features are basically the same.

With each decision rule, the decision tree tries to minimize the *Gini impurity*. In the end, the aim is to achieve the purest possible leaves, i.e. the predicted categories should match the actual categories as closely as possible. The `criterion` parameter controls which criterion is used to achieve the purest possible leaves. You can choose between `'gini'` and `'entropy'`. These two criteria give the same results in 98% of cases, but `'entropy'` can use more computing resources, which is why the default `'criterion='gini'` is often used.

Deep dive: The following content is optional, feel free to skip it.

How do you calculate the *Gini impurity*?

As mentioned above, the *Gini impurity* is a measure of the impurity of a class distribution. If there are two possible classes, it can take values between 0 and 0.5. The following applies: The greater the value, the more impure the distribution. A value of zero means that all values are assigned to the same class. A value of 0.5 means that all values are assigned random classes The formula to calculate it is: $\displaystyle \operatorname{I}_{Gini}=1-\sum _{i=1}^{J}{P(i)}^{2}$ Where $J$ represents the number of categories and $P(i)$ represents the probability of obtaining an observation of category$i$ by taking an element at random. In principle, it doesn't matter how many categories we want to assign when calculating the *Gini impurity*, but for the sake of simplicity we will restrict ourselves to the binary problem in our simple decision tree: Then we have the following formula: $\operatorname{I}_{Gini}=1-P(True)^{2}-P(False)^{2}$ So all we have to do is find out the probability of finding a person in this group who will leave the company ($P(True)$). To do this, we divide the number of people who will leave the company (167 people) by the number of all people in the group (1029 people). We do the same for the people who will stay in the company. We use both values in the formula to calcualte the *Gini impurity*: $\operatorname{I}_{Gini}=1-P(True)^{2}-P(False)^{2}=1-(\frac{167}{1029})^2-(\frac{862}{1029})^2 = 0.272$ We get the exact same value for the *Gini impurity* as in the top node in the visualization. This is the initial value for the decision of whether to split or not. The rule is that with each split, the total *Gini impurity* has to decrease. If more than one node is generated, the mean value of the *Gini impurity* of the resulting nodes has to be calculated, weighted by the number of observations. $\displaystyle \operatorname{{I}_{Gini}(Split1)} = \frac{0.498 \cdot 68 + 0.235 \cdot 961}{1029}=0.252$

Congratulations: You've generated your first decision tree. Now let's look in detail at what the rather unclear threshold of the PCA column actually reveals.

Interpret threshold values

In order to interpret the threshold value in terms of content, we create a mask (`mask`) that uses the threshold value found (`pca_years_0 <= -2,219`) to divide the data into two groups, just as the decision tree does. If we now apply this mask to our original training data and select the columns we used to create the PCA (these are stored in the corresponding transformer of our pipeline: `pipeline._columns[0]`), we can compare the medians in both groups in a way that can be directly interpreted, for example.

Run the following cell to get an overview:

In [10]:
```
#get column names from PCA
pca_cols = pipeline._columns[0]

# mask in piped training set
mask = features_train['pca_years_0']<=-2.219

# import relevant columns from original untransformed training data
df_train_original = pd.read_csv('attrition_train.csv', usecols=pca_cols)

#group and aggregate
df_train_original.groupby(mask).median().T
```

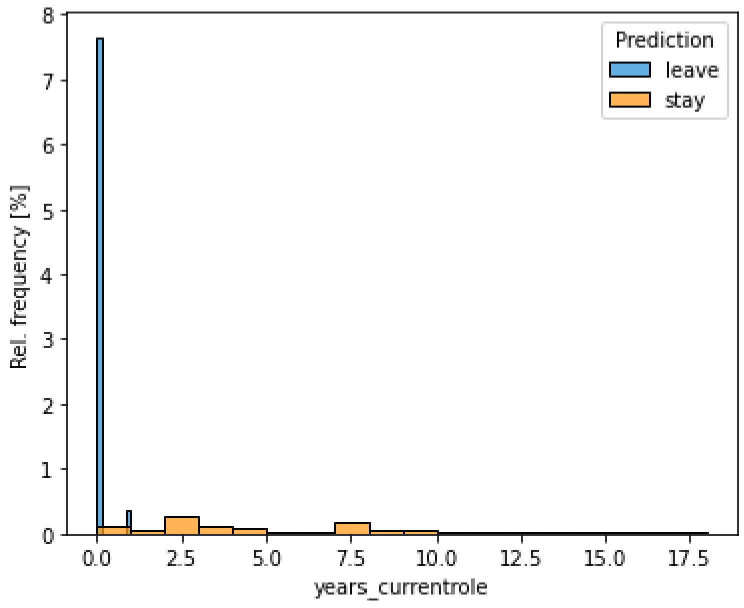
Out[10]:

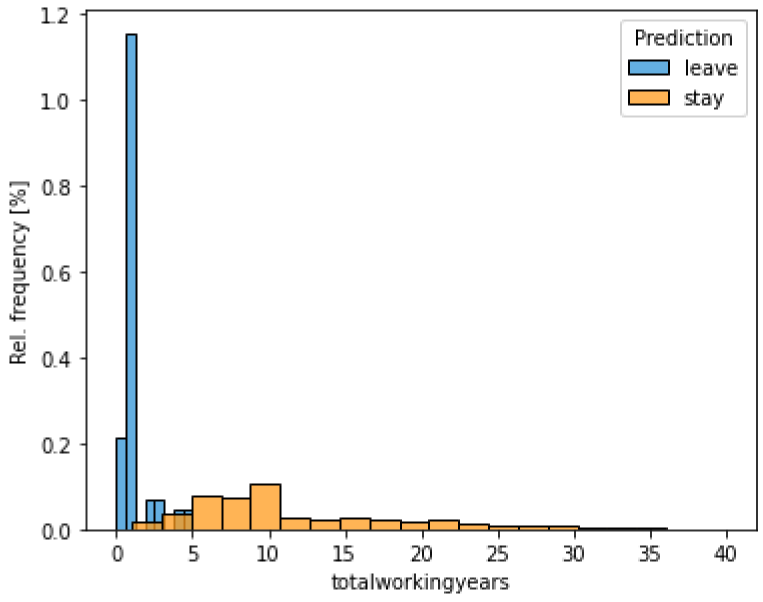
pca_years_0	False	True
totalworkingyears	10.0	1.0
years_atcompany	6.0	1.0
years_currentrole	3.0	0.0
years_lastpromotion	1.0	0.0
years_withmanager	3.0	0.0

It seems that most of the people who leave are young people at the start of their careers who have just joined the company. Let's look at this graphically:

In [11]:
```
import seaborn as sns

#select columns
cols = ['years_currentrole','totalworkingyears']

fig, axs = plt.subplots(len(cols),
                        figsize=((6,10)),
                        sharey=False)
#plot
for i, col in enumerate(cols):
    sns.histplot(df_train_original[mask].loc[:,col],
                 stat="density", # use relative frequncies
                 kde=False,      # mute kde-plot
                 label='leave',  # set label for legend
                 ax=axs[i],      # use subplot
                 color="#3399db")
    sns.histplot(df_train_original[~mask].loc[:,col],
                 stat="density",
                 kde=False,
                 label='stay',
                 ax=axs[i],
                 color="#ff9e1c")
    axs[i].set(ylabel='Rel. frequency [%]')
    axs[i].legend(title="Prediction") #title for legend

#prettify plot
fig.suptitle("Feature Distributions") #title for whole figure
fig.subplots_adjust(top=0.92) #set main title position
```

Feature Distributions





The visualizations confirm our first impression: The majority of the workers who left are people with a low value in `'totalworkingyears'` and `'years_currentrole'`.

Congratulations: You've generated and interpreted your first decision tree. It was an extremely simple tree, in order to make the principles of the classification model clear. The best thing to do now is to increase the complexity a little to improve the prediction quality.

Decision trees with several decision levels

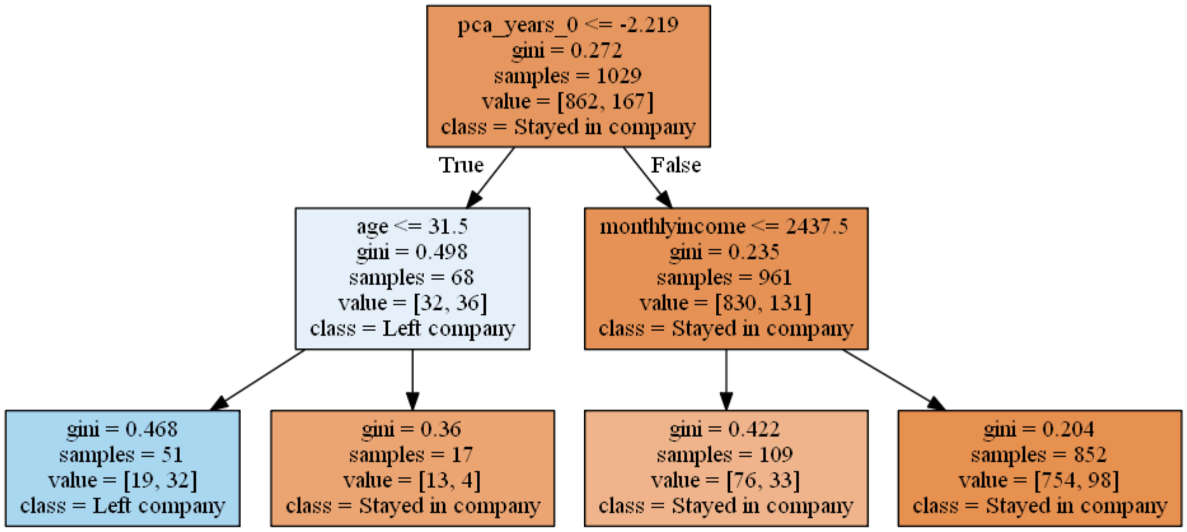
Instantiate a new decision tree model named `model_2`, which now uses two decision rules and `random_state=0` again.

In [12]:
```
model_2 = DecisionTreeClassifier(max_depth=2, random_state=0)
```

Now use `features_train` and `target_train` again to fit the model to the data.

In [13]:
```
model_2.fit(features_train, target_train)
```

Out[13]:
```
DecisionTreeClassifier(max_depth=2, random_state=0)
```



The decision rules the model has learned could now be displayed as follows in an upside down decision tree: decision tree depth 2

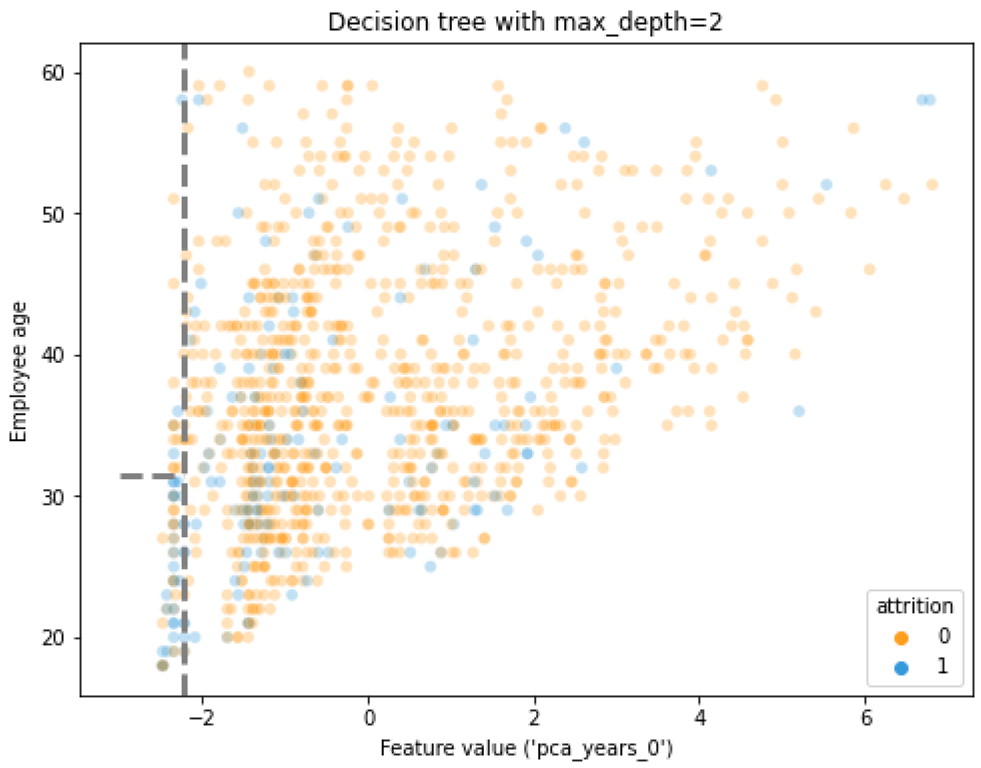
There are two decision rules before you get to the leaves. The upper two levels are identical to the decision tree with just one decision rule. The only difference is the bottom level with the leaves: The two leaves on the left differentiate between data points with different age values.

Right on the left is the group of people who are predicted to leave the company. So if the `'pca__years_0'` value is less than or equal to `-2.219` and the `'age'` value is also less than or equal to `31.5`, then the `'attrition'` category is predicted to be `1`. The opposite is predicted one leaf over to the right. So if the `'pca__years_0'` value is less than or equal to `-2.219` but the `'age'` value is greater than `31.5`, then the `'attrition'` category is predicted to be `0`.

The two leaves on the right predict the same category (`0` , i.e. not leaving the company). So there wasn't a feature that distinguishes the `'attrition'` categories particularly well after having already used `'pca_years_0'` . `'monthlyincome'` comes closest to in this regard. However, if you allowed additional decision rules by using a higher argument for `max_depth` , you could end up with completely different classifications further down through this branch.

If you visualize the decision rules as decision lines parallel to the axis, you can use a scatterplot.

In [14]:
```
fig, ax = plt.subplots(figsize=(8,6))
sns.scatterplot(data=features_train,
                x='pca_years_0',
                y='age',
                hue=target_train,
                alpha=0.3,
                ax=ax,
                palette=['#ff9e1c', '#3399db'])

# grey vertical line
ax.axvline(-2.219,
           ls='--',
           linewidth=3,
           color = "grey")

# grey horizontal line
ax.plot([-3,-2.219],
        [31.5, 31.5],
        ls='--',
        linewidth=3,
        color = "grey")

# set title and labels
ax.set(xlabel='Feature value (\'pca_years_0\')',
       ylabel='Employee age',
       title='Decision tree with max_depth=2');
```



In the scatter plot, the data points were assigned colors based on their actual `'attrition'` category. Grey decision lines represent threshold values. In our case everything to the right of the vertical decision line is predicted to be `0` (not leaving the company). For the data points to the left of the vertical line, it depends on whether the data points are below or on the horizontal decision line (`1` predicted) or above it (also `0` predicted).

By default, the decision tree generates as many decision rules as it needs to obtain leaves that are pure. Pure leaves only contain data points in one category. Let's also generate a maximum-depth tree so that we can compare as many different kinds of trees.

Instantiate a new decision tree model with the default settings of `DecisionTreeClassifier()` . Only pass the `random_state=0` parameter, so that the results are reproducible. Store the model in the variable `model_max` and then fit it to the data in `features_train` and `target_train.

In [15]:
```
model_max = DecisionTreeClassifier(random_state=0)
model_max.fit(features_train, target_train)
```

Out[15]:
```
DecisionTreeClassifier(random_state=0)
```

You could now display the decision rules the model has learned in an upside down decision tree like this:

decision tree depth 2

This decision tree now uses a lot of decision rules, which are hard to get an overview of. But remember that they are all nothing more than decision lines parallel to the axis. The first two levels of the decision tree are the same as in `model_2`. The branches below them are all that's new.

The colors indicate which category the model predicts. Just like in the scatter diagram, blue stands for `1` (predicted to leave the company) and orange for `0` (predicted not to leave the company). The color intensity indicates the purity of the collection of data points. Note that with a maximum-depth decision tree, the leaves are all absolutely pure. That's why the colors are so strong at the bottom of the tree.

Congratulations: You have learned how to generate a decision tree with as many decision rules as you like. The `max_depth` parameter is key here. However, to find out how well the model you generated copes with independent test data, you need to find out the model quality. We'll look at this next.

Evaluating decision trees

You evaluate decision trees just like any other classification model. In this case we'll concentrate on the following model quality metrics:

- Precision
- Recall

Define the variables `features_test` (all the numeric columns of `df_test`, these are in `num_cols`) and `target_test` (`'attrition'` column of `df_test`). Then calculate the predicted target values of the three models and store them in the variables `target_test_pred_1` (categories predicted with a decision tree with only one decision level), `target_test_pred_2` (categories predicted with a decision tree with two decision levels), `target_test_pred_max` (categories predicted with a decision tree with a maximum number of decision levels).

Then print the precision and recall metrics of each decision tree model, calculated based on the test data.

In [16]:
```python
features_test = features_test.loc[:, num_cols]

target_test_pred_1 = model_1.predict(features_test)
target_test_pred_2 = model_2.predict(features_test)
target_test_pred_max = model_max.predict(features_test)

from sklearn.metrics import precision_score, recall_score

print('Precision:')
print('1 decision level: ', precision_score(target_test, target_test_pred_1))
print('2 decision levels: ', precision_score(target_test, target_test_pred_2))
print('max decision levels: ', precision_score(target_test, target_test_pred_max))

print('\nRecall:')
print('1 decision level: ', recall_score(target_test, target_test_pred_1))
print('2 decision levels: ', recall_score(target_test, target_test_pred_2))
print('max decision levels: ', recall_score(target_test, target_test_pred_max))
```

```
Precision:
1 decision level:  0.5172413793103449
2 decision levels:  0.5416666666666666
max decision levels:  0.23684210526315788

Recall:
1 decision level:  0.21428571428571427
2 decision levels:  0.18571428571428572
max decision levels:  0.2571428571428571
```

You should get the following model quality values:

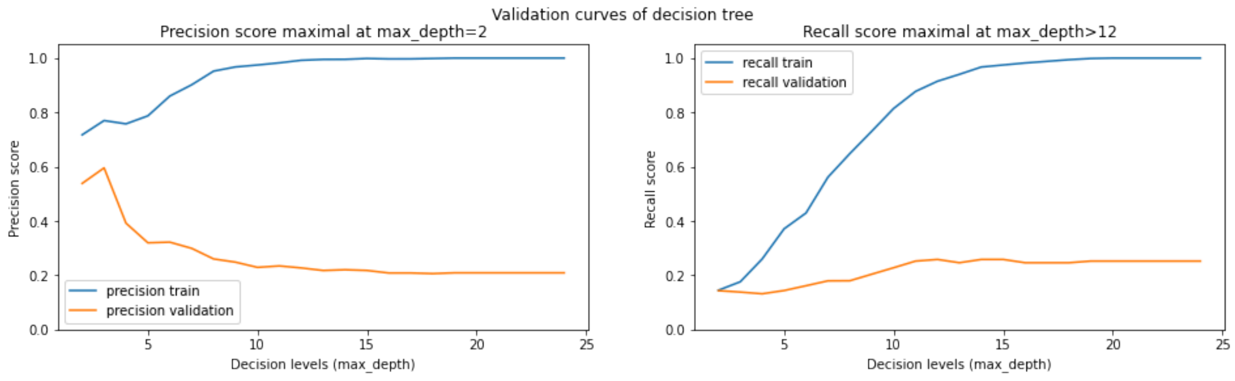
Model	precision	recall
`model_1`	51.7%	21.4%
`model_2`	54.2%	18.6%
`model_max`	23.7%	25.7%

It appears that as the decision tree size increases, the *precision score* measured from the test data decreases, while the *recall score* increases slightly. The difference in the precision indicates that the decision tree could suffer from overfitting as the decision levels increase. To verify this, it makes sense to look at the validation curves, see *Grid Search (Module 1, Chapter 2)*.

Validation curves show the model quality values at different hyperparameter settings. Typically, the quality of the model measured by the training data increases with increasing model complexity. But when measured based on the validation data, it decreases above a certain model complexity. This is an indicator of overfitting.

What do the validation curves look like with our data? For this purpose we'll use the `validation_curve()` function, which you got to know in *Grid Search (Module 1, Chapter 2)*.

In [18]:
```
# module import
import numpy as np
from sklearn.model_selection import validation_curve

# set-up for validation curve: which max_depth-settings to try
search_space = range(2, 25)

# initialize figures with two axes
fig, ax = plt.subplots(ncols=2, figsize=[16, 4])
fig.suptitle('Validation curves of decision tree')

################################################################################
# precision

# calculate precision values of different hyperparameter settings and cross validation
precision_train, precision_val = validation_curve(estimator=DecisionTreeClassifier(rar
                                                  X=features_train,
                                                  y=target_train,
                                                  param_name='max_depth',
                                                  param_range=search_space,
                                                  cv=5,
                                                  scoring='precision')

# calculate average precision values for each hyperparameter setting
precision_train_mean = np.mean(precision_train, axis=1)
precision_val_mean = np.mean(precision_val, axis=1)

# plot validation curve
ax[0].plot(search_space,
           precision_train_mean, label='precision train')
ax[0].plot(search_space,
           precision_val_mean, label='precision validation')
ax[0].set(ylim=[0, 1.05],
          xlabel='Decision levels (max_depth)',
          ylabel='Precision score',
          title='Precision score maximal at max_depth=2')
ax[0].legend()

#################################################################################
# recall

# calculate recall values of different hyperparameter settings and cross validation fo
recall_train, recall_val = validation_curve(estimator=DecisionTreeClassifier(random_st
                                            X=features_train,
                                            y=target_train,
                                            param_name='max_depth',
                                            param_range=search_space,
                                            cv=5,
                                            scoring='recall')

# calculate average recall values for each hyperparameter setting
recall_train_mean = np.mean(recall_train, axis=1)
recall_val_mean = np.mean(recall_val, axis=1)

# plot validation curve
ax[1].plot(search_space,
           recall_train_mean, label='recall train')
ax[1].plot(search_space,
           recall_val_mean, label='recall validation')
ax[1].set(ylim=[0, 1.05],
          xlabel='Decision levels (max_depth)',
          ylabel='Recall score',
          title='Recall score maximal at max_depth>12')
ax[1].legend();
```



It is noticeable that the precision measured for the validation data decreases as the model complexity increases. So as the depth of the tree increases, more people are falsely predicted to leave the company by the model. From about 12 decision levels upwards, the precision stops decreasing randomly and only fluctuates randomly.

At the same time, the recall increases up to a model complexity of 12 decision levels. After that it just fluctuates randomly. With 12 decision levels, we get a recall of about 25% on the validation data, i.e., not even half of the people who leave are correctly identified as such.

How can we explain this? We should remember that in the training data, only 16% of people leave the company. Due to the much larger proportion of people who didn't leave the company, the decision tree needs at least 12 decision steps to better identify the small number of people who did leave the company. You could therefore assume that the proportion of leaves predicting that someone will leave the company only increases only as the number of decision-making levels increases.

As a result, more and more data points are predicted as `1` (left the company) as the model becomes increasingly complex. This increases the chance of the predictions being off-target (precision decreases), but also increases the chance of identifying attrition cases (left the company) correctly (recall increases).

So it's not necessarily desirable to fit decision tree with the maximum amount of levels to the data. You typically use the various hyperparameters to simplify the decision tree model. This process is also called pruning. For example, you can limit the maximum number of decision-making levels (the `max_depth` parameter).

Alternatively, you can specify a minimum size of the leaves (`min_samples_leaf` parameter) or limit the number of leaves (`max_leaf_nodes` parameter). You can find more about this and other *pruning* options in the official documentation. To find the optimal values, we recommend using a grid search.

Congratulations: You got to know a new classification model: decision trees. It uses decision rules, i.e. decision lines parallel to the axis, to classify data. This approach produces particularly promising predictions if you combine several decision trees together to to form what's called a decision forest. We'll look at how to do this after we learn how to deal with unbalanced classes.

Remember:

- Import decision tree model: `from sklearn.tree import DecisionTreeClassifier`
- Instantiate model with `x` decision levels: `model = DecisionTreeClassifier(max_depth=x)`
- A maximum-depth tree can suffer from overfitting.

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