

Decision Tree Recap

Module 2 | Chapter 4 | Notebook 1

In this notebook we will recap the most important things from the previous chapter. This is all repetition. If you discover a lot of new things here, we recommend that you take a look back at Chapter 3. The relevant lessons for each section are clearly marked.

Decision Trees

In *Decision Trees (Chapter 3)* you got to know this model. It uses decision rules to differentiate between categories in the data. You can imagine the decision rules as a cascade of `if` queries.

In this recap lesson, we'll use the following data to get a more accurate picture. We'll use customer data from a bank.

```
In [1]: import pandas as pd
df = pd.read_csv('churn_data.csv')
print('Number of rows and columns:', df.shape)
df.head()
```

Number of rows and columns: (10000, 12)

```
Out[1]:
```

	CreditScore	Gender	Age	Years	Balance	NumOfProducts	HasCrCard	EstimatedSalary	Geograph
0	619	0	42	2	0.00	1	1	101348.88	
1	608	0	41	1	83807.86	1	0	112542.58	
2	502	0	42	8	159660.80	3	1	113931.57	
3	699	0	39	1	0.00	2	0	93826.63	
4	850	0	43	2	125510.82	1	1	79084.10	

The aim is to find out which customers are particularly at risk of leaving. Each row represents one customer. The columns have the following meaning:

Column number	Column name	Type	Description
0	'CreditScore'	continuous (<code>int</code>)	Credit rating score
1	'Gender'	categorical (nominal, <code>int</code>)	Gender: female (<code>0</code>) or male (<code>1</code>)
2	'Age'	continuous (<code>int</code>)	The person's age in years
3	'Years'	continuous (<code>int</code>)	The length of the business relationship in

Column number	Column name	Type	Description
			years
4	'Balance'	continuous (float)	Sum of all account balances
5	'NumOfProducts'	continuous (int)	Number of products (accounts, credit cards, etc.)
6	'HasCrCard'	categorical (nominal, int)	Has a credit card: yes (1) or no (0)
7	'EstimatedSalary'	continuous (float)	Estimated gross yearly salary in EUR
8	'Geography_France'	categorical (nominal, int)	Has business relationship in France: yes (1) or no (0)
9	'Geography_Germany'	categorical (nominal, int)	Has business relationship in Germany: yes (1) or no (0)
10	'Geography_Spain'	categorical (nominal, int)	Has business relationship in Spain: yes (1) or no (0)
11	'Exited'	categorical (nominal, int)	Customer has left: yes (1) or no (0)

The target variable is the 'Exited' column. We'll use the other columns as features.

```
In [2]: features = df.iloc[:, :-1] # use all columns except the last one as features
target = df.iloc[:, -1] # use last column as target
```

A decision tree can be a very good choice for this kind of problem. The model can make it clear to us with the help of decision rules why some customers might leave and why others might not. The decision tree model is located in `sklearn.tree` and is called `DecisionTreeClassifier`.

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


According to the [official documentation](#) we have some options for the hyperparameter settings. The `max_depth` parameter is particularly important. It controls how many decision rules are used to classify the data.

If we only want to know which feature is the most meaningful for classification purposes, we can set the tree depth `max_depth=1` and fit the model to the data.

```
In [4]: model = DecisionTreeClassifier(max_depth=1, random_state=0) # setting a random_state
model.fit(features, target)
```

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

We can imagine the decision tree as follows:


 Decision tree with depth of 1

As you can see here, it's conventional to visualize the decision tree upside down. At the top you can see the root. The leaves are at the bottom. We will denote `'Exited' = 0` as `Stayed` and `'Exited' = 1` as `Exited`. In the root you can see:

1. The first decision rule uses the feature `'Age'`. The threshold is `42.5`.
2. The *Gini impurity* in the root is `0.374`. This is the probability of misclassifying a data point based on this feature if all the `'Exited'` label were distributed randomly. A *Gini impurity* of 0 would indicate that all the data points belong to the same class.
3. The root contains `10000` data points. This corresponds to the total number of data points in `df_train`.
4. There are `7963` data points in the `0` category (`Stayed`) and `2037` data points in the `1` category (`Exited`).
5. The most likely category for the data points as a whole: stayed.

All data points that actually have an `'Age'` value less than or equal to `42.5` take the branch on the left (`True`) and immediately end up in the left-hand leaf and are classified as "Stayed". The rest end up in the right-hand leaf and are also classified as "Stayed". The leaves at the bottom of the decision tree represent the predicted categories when all decision rules have been gone through. You can see here that one decision rule on its own is apparently not enough to identify customers who leave.

If we add another decision rule, then it looks like this:

 Decision tree with depth of 2

The 2nd decision rule checks whether customers use two or fewer products. With this second rule, we get leaves on the tree that contain data points classified as `Exited`.

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.

The *Gini impurity* of the bottom leaf on the far right is very low. This means that it's very pure and, most importantly, contains correctly classified data points. Data point end up in this leaf if the answers to both questions asked by the tree are `False` (the left branch is `True` and the right branch is to `False`). So if customers are over 42 years old and have more than 2 products, it seems relatively certain that they will leave.

Congratulations: You have recapped what a decision tree is and how you can imagine it.

Dealing with imbalanced target categories

Data scientists often face the problem of generating predictions for unbalanced target categories. In the search for the best model, machine learning algorithms minimize the prediction error for all data points in the training set. Since the majority class has more data points, it usually has a stronger influence on the prediction error. For this reason, the model focuses particularly on the majority class, although we're usually particularly interested in the minority class.

In *Imbalanced Target Categories (chapter 3)* you learned 3 strategies to deal with this problem:

- Reduce the number of majority class data points (typically a random sample)
- Artificially increase the number of datapoints in the minority class (typically using what's called [bootstrapping](#))
- Give more weight to the incorrect classification of minority data points when training the model

More customers stayed rather than left. Is that the case here?

We can determine this quickly with the `pd.crosstab()` function, for example. You saw this function in *Employee Attrition (Chapter 3)*. It creates a contingency table for a column.

`pd.crosstab()` therefore counts how often each value of a column occurs and returns this as a new `DataFrame`. `pd.crosstab()` takes the parameters `index`, `columns` and `normalize`. You assign the feature you're interested in to `index`, e.g. `df_train.loc[:, 'attrition']`. `columns` is the name of the columns in the contingency table. You can assign this a `str` such as `'count'`. The `normalize` parameter specifies whether you want the contingency table to contain whole numbers or proportions. With `normalize='columns'` the proportions are calculated column by column.

Let's look at the proportions of customers who have left and customers who have stayed in the whole data set.

```
In [5]: pd.crosstab(index=df.loc[:, 'Exited'], columns='count', normalize='columns')
```

```
Out[5]:
```

col_0	count
Exited	
0	0.7963
1	0.2037

About 20% of the customers in the data set left. Finding this out with a model is particularly interesting for the bank.

Let's go through the 3 strategies again briefly before applying them.

- *Undersampling*: reducing the number of majority class data points (typically a random sample).

You can imagine this strategy by first counting how many minority class data points there are. Then you select the same number of data points of the majority class at random

The training data set shrinks considerably as a result, which can cause complications when training classification models. Therefore, we only recommend this strategy if you have an extremely large training data set and the smaller category is not too small.

So sometimes it's better to use the second strategy:

- *Oversampling*: artificially increasing the number of datapoints in the minority class (typically using what's called [bootstrapping](#))

You can imagine this strategy by first counting how many minority class data points there are. Now multiply the data points of the smaller category until there are just as many of them as there are in the bigger category. You use the bootstrap method, which is a selection with replacement. This means that the same data point can be selected multiple times.

You can implement both strategies with the `my_df.sample()` method. Its `n` parameter controls the size of the sample. The `replace` parameter controls whether a data point can be selected multiple times (`replace=True`) or not (`replace=False`).

In the following cell we'll create two models to test both strategies. To validate the models, we'll use `KFold` from `sklearn.model_selection`, which we learned about in *evaluating models with cross-validation (Module 1, Chapter 2)*. This allows us to divide the data into training and validation sets. We'll apply the *undersampling* and *oversampling* methods to the training sets. We'll then train the models and calculate the precision and recall using the validation data.

```
In [6]: # import everything we need
import numpy as np
from sklearn.model_selection import KFold
from sklearn.metrics import precision_score, recall_score

# make two models with maximum depth
model_undersampling = DecisionTreeClassifier(random_state=0)
model_oversampling = DecisionTreeClassifier(random_state=0)

# create empty lists to hold the validation scores
precision_undersampling = []
precision_oversampling = []
recall_undersampling = []
recall_oversampling = []

kf = KFold(n_splits=5) # create 5 cross-validation folds

# under- and oversample in the training data for every fold and validate model
for train_idx, val_idx in kf.split(features):

    # training
    features_train = features.iloc[train_idx, :] # get training features
    target_train = target.iloc[train_idx] # get training target

    mask_majority = (target_train == 0) # identify on which position the majority val
```

```

mask_minority = (target_train == 1) # do the same for the minority values

len_majority = mask_majority.sum() # sum of mask_majority is the number of majority
len_minority = mask_minority.sum() # do the same for the minority values

features_train_majority = features_train.loc[mask_majority, :] # select only majority
features_train_minority = features_train.loc[mask_minority, :] # select only minority

# undersample majority values to the number of minority values and append the minority
features_train_undersampling = features_train_majority.sample(n=len_minority, random_state=42)
features_train_undersampling = features_train_undersampling.append(features_train_minority)
# target is 0 for undersampled majority and 1 for minority
target_train_undersampling = [0 for idx in range(len_minority)] + [1 for idx in range(len_majority)]

# oversample minority values to the number of majority values and append the majority
features_train_oversampling = features_train_minority.sample(n=len_majority, random_state=42, replace=True)
features_train_oversampling = features_train_oversampling.append(features_train_majority)
# target is 1 for oversampled minority and 0 for majority
target_train_oversampling = [1 for idx in range(len_majority)] + [0 for idx in range(len_minority)]

# fit the models to the under- and oversampled data
model_undersampling.fit(features_train_undersampling, target_train_undersampling)
model_oversampling.fit(features_train_oversampling, target_train_oversampling)

# validation
features_val = features.iloc[val_idx, :] # get validation features
target_val = target.iloc[val_idx] # get validation target

target_pred_undersampling = model_undersampling.predict(features_val) # predict with undersampling
target_pred_oversampling = model_oversampling.predict(features_val) # predict with oversampling

precision_undersampling.append(precision_score(target_val, target_pred_undersampling))
recall_undersampling.append(recall_score(target_val, target_pred_undersampling))

precision_oversampling.append(precision_score(target_val, target_pred_oversampling))
recall_oversampling.append(recall_score(target_val, target_pred_oversampling)) #

# print mean of precision and recall for both models
print('Undersampling: precision:', np.mean(precision_undersampling), 'recall:', np.mean(recall_undersampling))
print('Oversampling: precision:', np.mean(precision_oversampling), 'recall:', np.mean(recall_oversampling))

```

Undersampling: precision: 0.3645847101898111 recall: 0.6829996133310482

Oversampling: precision: 0.4676147355542669 recall: 0.4492556561057543

If we process the data with *undersampling*, we get a precision of about 36% and a recall of about 68%. So we manage to identify 68% of the customers who left, but we also catch a lot of customers who didn't.

If we use *oversampling* in this example, the precision is slightly higher (approx. 47%) and the recall is significantly lower (approx. 45%). This means that we can track down significantly fewer customers who actually migrate. But we are often right about those we classify as migrating.

In this example, *undersampling* would be preferable in order to identify as many of the customers who will actually leave as possible so the bank can try to retain them. You should always evaluate which method is better individually for each case.

As well as these two approaches there is an even simpler strategy:

- Give more weight to the incorrect classification of minority data points when training the model.

This strategy doesn't start with the data, but with the classification algorithm itself. If you set the `class_weight` parameter to `'balanced'` when you instantiate the model, then in our case, the false classification of a data point in category `1` (leaves company) is weighted four times more heavily than the false classification of a data point in category `0` when training the model. Almost all the models in `sklearn` have this parameter.

If we apply this setting in this example, we get the following results:

```
In [12]: # make model with class weights
model_balanced = DecisionTreeClassifier(class_weight='balanced', random_state=0)

# create empty lists to hold the validation scores
precision_balanced = []
recall_balanced = []

kf = KFold(n_splits=5) # create 5 cross-validation folds

# train and validate for every fold
for train_idx, val_idx in kf.split(features):

    # training
    features_train = features.iloc[train_idx, :] # get training features
    target_train = target.iloc[train_idx] # get training target

    model_balanced.fit(features_train, target_train) # fit the model to the data

    # validation
    features_val = features.iloc[val_idx, :] # get validation features
    target_val = target.iloc[val_idx] # get validation target

    target_pred_balanced = model_balanced.predict(features_val) # predict with model

    precision_balanced.append(precision_score(target_val, target_pred_balanced)) # co
    recall_balanced.append(recall_score(target_val, target_pred_balanced)) # fo

# print mean of precision and recall
print('Balanced class weights: precision:', np.mean(precision_balanced), 'recall:', np
```

Balanced class weights: precision: 0.4699524980382811 recall: 0.4485974413249621

With the parameter `class_weights='balanced'` we get very similar values (precision 47%, recall 45%) here to the model that uses *oversampling*. But we were able to save a few lines of code.

Congratulations: You've recapped what *undersampling* and *oversampling* are. You also recapped the parameter `class_weights='balanced'` for a quick way of dealing with imbalanced target categories.

Decision trees combined to make a *random forest*

To ensure that machine learning models react quickly to changes in the training data (high *variance*, potential overfitting) without becoming less stable (low *bias*), people have developed methods that combine models. The hope is that the resulting meta-models will generate better predictions than each individual model that they are made up of.

Roughly speaking, there are 3 approaches to combine machine learning models:

1. *Ensembling*: Combining predictions so that the meta-model provides more stable predictions (implemented in `sklearn` with `from sklearn.ensemble.VotingClassifier`)
2. *Stacking*: Using predictions from some models as features for a higher-level model to generate better predictions (implemented in the `vecstack` module)
3. *Boosting*: A series of models is trained so that that the weight of data points that were falsely classified in the previous step of the series, is heavier. So the boosting model as a whole focuses on the difficult data points, in the hope that this will improve the overall predictions (implemented in `sklearn` with `sklearn.ensemble.AdaBoostClassifier` , for example)

In the last chapter we focused on ensembling. This is also the basis for developing decision trees into *random forests*, which we learned about in *From Decision Trees to Random Forests with Ensembling (Chapter 3)*. With regard to ensembling, it should be noted that the combination of classification models is only promising if the models differ.

A `random forest` uses two strategies to give chance more influence on the decision trees to combine various different predictions:

- *Bagging, which is short for bootstrap aggregation*: For each decision tree, the data points are selected randomly via the bootstrap procedure, where data points can be selected several times. The data set for each decision tree is the same size as the training data, but data points can occur more than once. If you set the `bootstrap` hyperparameter to `False` , it uses the entire training data set each time.
- Random feature selection: Each time you want a decision tree to generate a decision rule, it only has a few randomly selected features available to it. Generally, we use $\sqrt{\text{number of features}}$ (rounded down) for this.

So the random forest creates trees that make slightly different predictions. But the different predictions combined together should then be better than each individual. The question now is how to combine the predictions. `sklearn` uses the predicted category probabilities, calculates their average and bases the predicted category on this average.

Decision trees calculate their probabilities as follows: The data point goes through all the decision rules and ends up on a particular leaf. A certain number of data points also ended up in this leaf during training. The number of data points that have belonged to a category is in relation to all the data points in the leaf. This value is then used as the probability that the new data point belongs to this category.

Let's try out a random forest directly on the data. Import `RandomForestClassifier` directly from `sklearn.ensemble`.

```
In [8]: from sklearn.ensemble import RandomForestClassifier
```

Now we'll instantiate the the random forest with 100 trees (`n_estimators=100`). We'll also set the `class_weight` parameter to `'balanced'` . We'll also set `random_state=0` here to get reproducible results.

```
In [9]: model_rf = RandomForestClassifier(n_estimators=100,
                                         class_weight='balanced',
                                         random_state=0)
```

Now we'll calculate the precision and recall again.

```
In [10]: # create empty lists to hold the validation scores
precision_rf = []
recall_rf = []

kf = KFold(n_splits=5) # create 5 cross-validation folds

# train and validate for every fold
for train_idx, val_idx in kf.split(features):

    # training
    features_train = features.iloc[train_idx, :] # get training features
    target_train = target.iloc[train_idx] # get training target

    model_rf.fit(features_train, target_train) # fit the model to the data

    # validation
    features_val = features.iloc[val_idx, :] # get validation features
    target_val = target.iloc[val_idx] # get validation target

    target_pred_rf = model_rf.predict(features_val) # predict with random forest

    precision_rf.append(precision_score(target_val, target_pred_rf)) # calculate precision
    recall_rf.append(recall_score(target_val, target_pred_rf)) # for random forest

# print mean of precision and recall
print('Random Forest: precision:', np.mean(precision_rf), 'recall:', np.mean(recall_rf))
```

```
Random Forest: precision: 0.7362957360396972 recall: 0.41623558564070856
```

The random forest's precision score of 74% is significantly better, compared to 47% from the single decision tree. The recall (42%) is only slightly worse. With this model, we are able to identify a similar number of customers who actually leave, but with a significantly lower error rate.

Congratulations: You've recapped random forests. This is a very popular model as it very often produces good results.

Remember:

- Instantiate decision tree with x decision levels: `model = DecisionTreeClassifier(max_depth=x)`
- Undersampling, oversampling and `class_weight='balanced'` can help with imbalanced target categories
- A random forest combines decision trees with random elements and often produces good predictions as a result.

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

Found a mistake? Contact Support at support@stackfuel.com.