From Decision Trees to Random Forests with Ensembling

Module 2 **|** Chapter 3 **|** Notebook 6

In this notebook you will learn how to combine classification algorithms to achieve better results. This approach is also used to combine decision trees to create random forests. By this end this lesson you will have learned about:

- Ensembling
- Bagging
- Random Forests

Ensembling and bagging

To ensure that machine learning models react quickly to changes in the training data (high *variance*, potential overfitting) without losing stability (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 three 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, which 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 this chapter we'll concentrate on ensembling. This is also the basis for developing decision trees into random forests. With regard to ensembling, it should be noted that the combination of classification models is only promising if the models differ. If all the models deliver the same predictions anyway, there is little to be gained from ensembling.

It would now be possible to use ensembling to combine the k-Nearest-Neighbors, logistic regression and decision tree classification models that you have got to know so far. We know that they have totally different approaches to generating predictions (neighbourhood voting, regression to log odds, decision rules). Instead, in this exercise we'll focus on how the very popular *random forest* algorithm is an ensemble of decision trees.

The decision trees we have got to know so far are barely suitable for ensembling because they lead to similar predictions. If you just combine the same predictions with each other again and again, it's no better than the predictions from the individual classification algorithms. We would need stronger random elements that lead to the decision tree results being different.

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

- 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, only a few randomly selected features are available to it How many features do you want? Generally, $\sqrt{number\ of \ features}$ (rounded down) is chosen for this. For the attrition data set, this would be $\sqrt{15}=3.87$, so we round down to 3 features. This means that when the decision tree generates each decision rule, it uses the best of three randomly selected features to distinguish the categories from each other.

Let's start by focusing on bootstrap aggregation - *bagging*.

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.

As usual, we'll start by preparing the data.

In [1]:
```python
import pandas as pd
import pickle

#load pipeline
pipeline = pickle.load(open("pipeline.p",'rb'))
col_names = pickle.load(open("col_names.p",'rb'))

#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[1]:

	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`)
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`)

Column number	Column name	Type	Description
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

In this lesson, it's easier if we put `features_train` and `target_train` together in one `DataFrame` . This will saves us a lot of effort when implementing our own decision tree. Run the following cell so we can get started quickly:

In [2]:
```
#combine features and target
df_train = pd.concat([features_train, target_train], axis=1)
df_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	

In the previous lesson we learned what the bootstrap procedure is: taking observations from the data set and putting them back so that they may be taken again. With the help of `my_df.sample()` you can use bootstrapping techniques to create new samples of any size (`n` parameter of `my_df.sample()`). Note that for bootstrapping, the `replace` parameter has to be set to `True` . Create a list (`list`) of 100 bootstrap samples from the training data (`df_train`) with the same length as this one. Name the list `list_df_train_bagging` . Use the iterator variable as an argument for the `random_state` parameter. This way you get different samples, but you can reproduce them at the same time.

Tip: Instead of the `n` parameter, you can use the `frac` parameter of `my_df.sample()`. You can use this to specify the number of data points as a ratio to the original data points. So with `frac=1` you end up with as many data points as the training data has.

In [3]:
```
list_df_train_bagging = []

for i in range(100):
    list_df_train_bagging.append(df_train.sample(frac=1, replace=True, random_state=i)
```

Now create similar lists of feature matrices and target vectors. Name them `list_features_train_bagging` and `list_target_train_bagging`. Use all the columns as features except `'attrition'`.

In [4]:
```
list_features_train_bagging = []
list_target_train_bagging = []

for df_tmp in list_df_train_bagging:
    list_features_train_bagging.append(df_tmp.drop('attrition', axis=1))
    list_target_train_bagging.append(df_tmp.loc[:, 'attrition'])
```

Congratulations: Now we have 100 bootstrap aggregated feature matrices and the corresponding target vectors. You've now completed the bagging phase. Next, we'll turn to the second random process of random forests.

A simple version of a random forest

The bootstrap aggregated feature matrices and target vectors form the training data for 100 decision trees, which we can combine after they have been trained. But before we train the 100 decision trees, let's look at the second random element of *random forests*:

- 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

Unfortunately there isn't a parameter in `DecisionTreeClassifier()` that lets you recreate this functionality 1:1. However, if we set the `splitter` parameter to `'random'`, only one feature will be randomly selected when generating a new decision rule. So we don't evaluate which feature is the best like we did before. We can add another random element this way.

Now train 100 decision tree models as if they were (approximately) part of a random forest. Proceed as follows:

- Import `DecisionTreeClassifier` from the `sklearn.tree` module
- First define an empty list called `tree_models`.
- Write a `for` loop with 100 iterations representing the 100 bootstrap-aggregated training data sets of the 100 decision trees. You can call the iterator variable `i`. At each iteration:

1. Instantiate a decision tree model (`model_tmp`) with 4 decision levels and `class_weight='balanced'` , `splitter='random'` and `random_state=i`
2. Fit this model to your own training data (`list_features_train_bagging[i]` and `list_target_train_bagging[i])`)
3. Add the trained model is added to the `tree_models` list

In [5]:
```
from sklearn.tree import DecisionTreeClassifier
tree_models = []
for x in range(len(list_df_train_bagging)):
    model_tmp = DecisionTreeClassifier(max_depth=4, class_weight='balanced',splitter='
    model_tmp.fit(list_features_train_bagging[x], list_target_train_bagging[x])
    tree_models.append(model_tmp)
```

Now we have the same decision tree model 100 times, which was trained on slightly different training data. Each model would now make a slightly different prediction based on the same test data. But the various forecasts together should then be better than each one individually.

The question now is how to combine the predictions. Originally, *random forests* only accessed the binary predicted categories and held a majority vote. However, `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 it out by creating a `DataFrame` called `df_target_test_pred_proba` which stores the predicted probabilities of the 100 decision trees in its 100 columns. Each row of `df_target_test_pred_proba` represents one test datapoint (rows in `features_test`). Later we can aggregate these probabilities.

Tip: We don't use bagging or anything similar on the test data. You shouldn't modify the test data for a model evaluation.

In [6]:
```
predictions= {}  # create dict to hold the predictions of each decision tree
for i, model_tmp in enumerate(tree_models):  # use enumerate to get numbers from 0 to
    # save the predictions of each tree for category attrition = 1 via [:, 1]
    predictions['model_{}'.format(i)] = model_tmp.predict_proba(features_test)[:, 1]

df_target_test_pred_proba = pd.DataFrame(predictions)  # convert the dict to a df

pd.set_option('precision', 2)  # show only 2 decimals

df_target_test_pred_proba.head()
```

Out[6]:

	model_0	model_1	model_2	model_3	model_4	model_5	model_6	model_7	model_8	model_9
0	0.42	0.09	0.37	0.14	0.00	0.12	0.00	0.28	0.21	0.23
1	0.61	0.50	0.37	0.14	0.37	0.65	0.54	0.59	0.21	0.63
2	0.60	0.05	0.00	0.77	0.28	0.82	0.74	0.30	0.83	0.27
3	0.40	0.09	0.14	0.14	0.61	0.09	0.00	0.28	0.38	0.23
4	0.90	0.99	0.92	0.89	0.92	0.96	0.95	0.88	0.96	0.94

5 rows × 100 columns

If you now scroll from left to right through the `DataFrame` of predicted probabilities, you'll see that most of the models are in agreement. The variability that still occurs due to *bagging* and feature selection is the core of why *random forests* are powerful.

Calculate the precision and the recall of the overall model combining the 100 decision trees using the data in `df_target_test_pred_proba`. Is the overall model better than the decision tree model model_balanced_by_class_weights from the last lesson (precision 30%, recall 59%)?

In [7]:
```
from sklearn.metrics import precision_score, recall_score

print('Precision: ', precision_score(target_test, df_target_test_pred_proba.mean(axis=
print('\nRecall: ', recall_score(target_test, df_target_test_pred_proba.mean(axis=1).r
```

```
Precision:  0.4567901234567901

Recall:  0.5285714285714286
```

Because bagging and feature selection involve random elements, your model quality scores may be slightly different. You should get the following values:

Model	precision	recall
`df_target_test_pred_proba`	approx. 46%	approx. 53%

Thus the replicated *random forest* has a much better recall than the best decision tree from the last lesson. In return, the precision is reduced in comparison to `model_balanced_by_class_weights` from the last lesson.

Congratulations: You've recreated a random forest. Admittedly, this wasn't easy. But now you have a good idea of what's going on behind the scenes of perhaps the most popular classification algorithm. So that you don't think that it's always this complicated to use a *random forest*, we'll finally look at `RandomForestClassifier`.

The random forest

Now instantiate a *random-forest* model directly by first importing `RandomForestClassifier` directly from `sklearn.ensemble`. Then instantiate the model and store it in the variable `model_rf`. Use 100 trees (`n_estimators=100`). You can also tell the *random forest* how the trees should proceed, like in `DecisionTreeClassifier`. So set the `max_depth` parameter to `4` and the `class_weight` parameter to `'balanced'`. Use `random_state=42` to get reproducible results.

In [20]:
```
from sklearn.ensemble import RandomForestClassifier
model_rf = RandomForestClassifier(n_estimators=100, max_depth=4, class_weight='balance
```

Train the random forest model. You can use the data `features_train` and target values `target_train` just as they were created at the beginning of this lesson. The algorithm does the *bagging* by itself.

In [21]:
```
model_rf.fit(features_train,target_train)
```

Out[21]:
```
RandomForestClassifier(class_weight='balanced', max_depth=4, random_state=42)
```

Now check how good this model is. Calculate the precision and recall for the random forest.

In [22]:
```
predictRandomForest = model_rf.predict(features_test)

print('Precision: ', precision_score(target_test, predictRandomForest))
print('Recall: ', recall_score(target_test, predictRandomForest))
```

```
Precision:  0.44871794871794873
Recall:  0.5
```

Since the random forest works with random elements, your results may vary slightly. Here are our values:

Model	precision	recall
`df_target_test_pred_proba`	approx. 46%	approx. 53%
`model_rf`	approx. 45%	approx. 50%

The random forest achieves similar values as the decision forest we created ourselves above.

The big advantage of *random forests* is that you can use them quite easily "out of the box". For example, while with logistic regression you have to worry about features using the same scales and not correlating strongly, with random forests you don't have to worry about that.

Because the decision trees of *random forests* only work with decision lines parallel to the axis, you can also use categorical data. It's not important whether the data is on an interval scale (measured value) or an ordinal scale (ranking). As long as they are numerically coded, the *random forest* can handle it well.

The price you pay is that you have to train a lot of decision trees. Random forest is an ensembling model. This requires more computing resources than a simple decision tree or logistic regression. How many trees should you use in the *random forest*? In general: The more the merrier. We tried a few values for `n_estimators` from `RandomForestClassifier` and got the following values:

`n_estimators`	mean value (standard deviation) of precision	mean value (standard deviation) of recall
2	approx. 46% (9%)	approx. 15% (4%)
5	approx. 48% (7%)	approx. 27% (5%)
10	approx. 65% (9%)	approx. 21% (4%)
50	approx. 74% (6%)	approx. 24% (3%)
100	approx. 77% (5%)	approx. 24% (2%)
500	approx. 76% (4%)	approx. 25% (1%)
1000	approx. 76% (3%)	approx. 24% (2%)

We trained each *random forest* model 100 times. We noticed that as the number of trees in the random forest increases, the variability of the results decreases. By averaging over more and more decision trees, the potential for random variations in a random forest's predictions decreases.

At the same time, the predictions get better and better as the number of trees increases. This is particularly evident in the precision. So the predictions become more and more reliable. But even the recall increases slightly with the number of trees in the random forest. So the model also gets better at recognizing employees who leave correctly. So, if you want to maximize both precision and recall, you should use as many decision trees in the random forest as possible.

Congratulations: You first recreated the random forest algorithm more or less from scratch and then saw how to use it directly in `sklearn`. Along the way, you learned an approach for combining machine learning models: ensembling This is also great for combining the classification models you have learned so far. You'll learn how to do that next.

Remember:

- `from sklearn.ensemble import RandomForestClassifier`
- The decision trees in a random forest differ from normal decision trees in two ways: training data *bagging* and the feature selection when generating a decision line
- The more decision trees in the random forest, the better (`n_estimators` parameter of `RandomForestClassifier`)

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.
