

# Model Specific and Model Agnostic Methods

Module 3 | Chapter 1 | Notebook 3

---

In the last two exercises we took a closer look at the decision tree and found that visualizing its decision rules made it much easier to interpret. We were able to use a local interpretation. Unfortunately, other machine learning models can't be visualized and interpreted as easily. The reason for this is simple: The decisions are made by more complex and comprehensive algorithms. But there are other interpretation methods that can be applied to all models. We will look at some of them in this lesson and the following lesson.

By the end of this lesson:

- You will know the difference between model-specific and model-agnostic methods.
  - You will know what *permutation feature importance* means.
  - You will be able to calculate permutation feature importance.
- 

## Global interpretation methods

As you have learned, there is a difference between local and global interpretability. Local interpretation is only possible with understandable and clear models. You saw this in action, using a decision tree as an example. But now let's turn to more complicated models. We can only use global methods for these, which means identifying feature importances. We differentiate between model-specific and model-agnostic methods for this.

- **Model-specific methods:** Functions built into a model that calculate feature importance.
- **Model-agnostic methods:** General strategies for obtaining feature importance. Can be applied to any model.

We will now look at an example of both methods, using random forest - a black box model.

## Model-specific methods

Let's remember what we've learned so far.

In the field of machine learning, a **black box model** is a model that cannot be interpreted locally. Black box models often provide significantly better predictions, but it's difficult to say how the algorithm arrived at these results. This in turn makes effective data storytelling more difficult.

Random forests are an example of a black box model. You have already learned how to visualize decision trees and gain insights from them. This visualization made it possible to trace the

decision tree's decisions precisely, and see which features led to the prediction. Visualizing random forests in this way would make little sense because they are basically a collection of lots of decision trees (with a random subset of features and data points). So on one hand, visualizing individual trees would offer little explanatory potential and, on the other hand, visualizing all the decisions would be incomprehensible and overwhelming.

And this is where **global methods** come into play. They are a data scientist's first choice to interpret black box models. We have already got to know one of these methods. The feature importance of the decision tree. You could find these with the trained model's `.feature_importances_` attribute. This attribute is specific to tree algorithms, which means that random forests also have it. So it is a model-specific method for tree algorithms.

Now let's look at how it works in the random forest in `sklearn` to get a feel for black box models.

**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 management wants to develop a strategy to improve morale in the company. So they are interested in which factors would have the greatest effects.

They are already quite happy with your decision tree. However, they (correctly) suspect that the result can be improved upon with a random forest. So they would like an analysis based on this model.

Run the following code cell to import, split and print the training data.

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

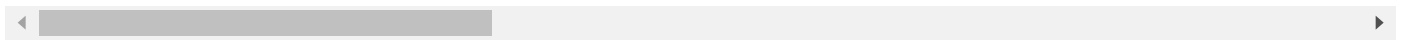
# split training data into features and target
features_train = df_train.drop('attrition',axis=1)
target_train = df_train.loc[:, 'attrition']

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 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

Column number	Column name	Type	Description
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

Now instantiate a random forest with `RandomForestClassifier` from `sklearn.ensemble` as the variable `model_rf`. Use balanced class weights, a maximum depth of 12 and `random_state=0`. Your random forest should consist of 100 individual trees. Then fit `model_rf` to the training data.

```
In [2]: from sklearn.ensemble import RandomForestClassifier
model_rf = RandomForestClassifier(class_weight='balanced', random_state=0, max_depth=12)
model_rf.fit(features_train, target_train)
```

```
Out[2]: RandomForestClassifier(class_weight='balanced', max_depth=12, random_state=0)
```

**Deep Dive:** Now you can briefly consider what would happen if you interpreted this kind of model locally. For the decision tree we simply followed the decision rules and arrived at a prediction after reaching the maximum depth of 3. Here we would have to do this for 100 decision trees with a depth of 12. It really would be too confusing and we don't recommend trying it. Only global methods have a chance here.

Which features are particularly important for the model? Just like `DecisionTreeClassifier`, `RandomForestClassifier` has the attribute `my_model.feature_importances_`. Create a bar chart from it. It's only explorative, so you don't need to make it particularly pretty.

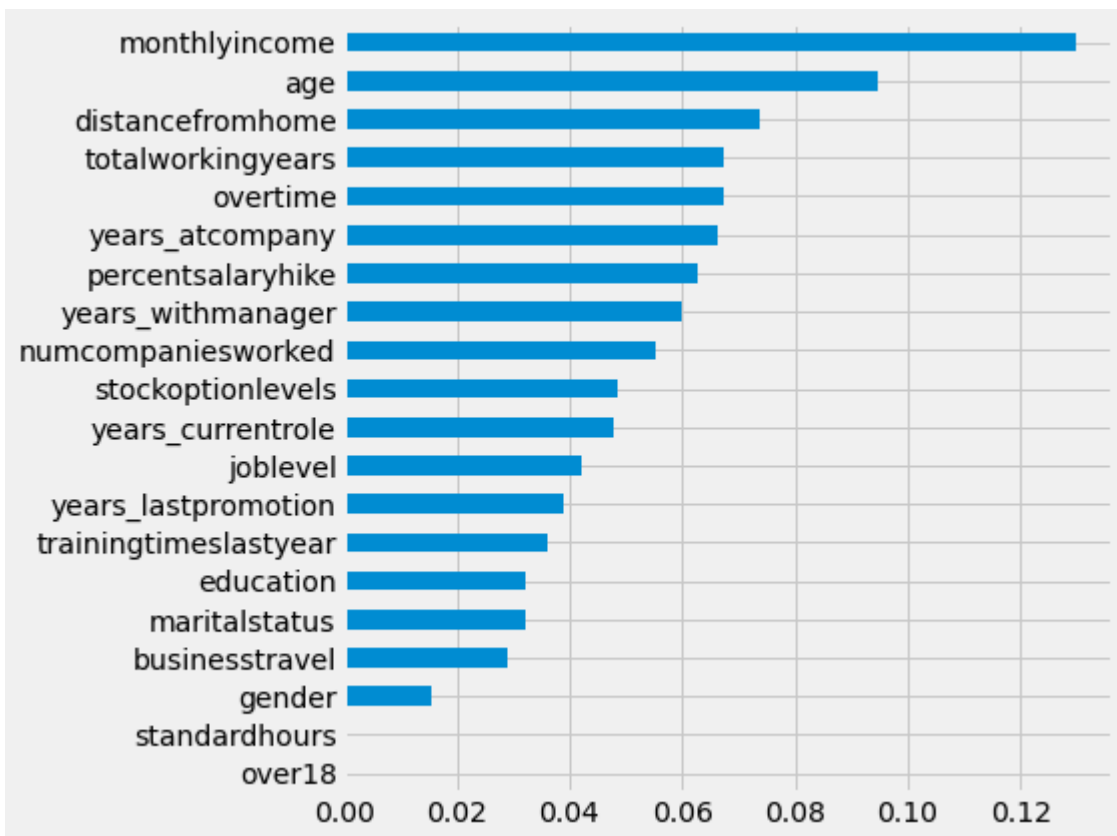
```
In [3]: import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')
```


```
fig, ax = plt.subplots(figsize=(8, 6))

# Convert feature importance array into a series and visualize
feature_importance = pd.Series(model_rf.feature_importances_,
                               index=features_train.columns,
                               ).sort_values()

feature_importance.plot(kind='barh', ax=ax)
fig.tight_layout()
```



We didn't make any special customizations on our bar chart except to use the ready-made style `'fivethirtyeight'`. Your visualization should look something like this:

 Feature Importance of the random forest

An interesting result! Let's look at it in more detail.

As expected, you can see that `standardhours` and `over18` are completely unimportant, since all the values in these columns are the same (see *Employee Turnover, Module 2, Chapter 3*).

Apart from that, the result is quite different from the decision tree. In the last lesson, the decision tree told us that `overtime` was the main reason for termination, at 33%. The random forest, on the other hand, rates `overtime` as the third most important feature (with only about 7% importance). `monthlyincome` is now also the most important feature (with about 13% importance), which was only the sixth most important feature in the decision tree (with 6% importance).

The different results are not necessarily surprising. As you saw in *Module 2, Chapter 3*, the problem of employee turnover is a difficult one. The *random forest* seems to capture this

complexity well, since almost all the features have an impact on the turnover rate this time, in complete contrast to the decision tree (where only 6 features were important).

It's important to understand here that each model can bring a new perspective so you should always look at several models. This makes it even more important to be able to interpret each model so that you can create a consistent and convincing data story.

**Remember:** A good data story is based on the results of many models!

**Deep Dive:** How did the random forest determine the feature importances? First, the feature importances for each individual decision tree are determined, and then the overall result is calculated as an average of all the individual results.

Feature importances only make sense in the context prediction quality they are associated with. You shouldn't create a data story based on a model with bad predictions. Let us look at the quality of our random forest.

Run the following cell to import the test data.

```
In [4]: # read data
import pandas as pd
df_test = pd.read_csv('attrition_test.csv')

# split training data into features and target
features_test = df_test.iloc[:, 1:]
target_test = df_test.loc[:, 'attrition']
```

What precision and recall values do we get with the test data?

```
In [5]: from sklearn.metrics import precision_score, recall_score
target_test_pred = model_rf.predict(features_test)

print('Precision: ', precision_score(target_test, target_test_pred))
print('Recall: ', recall_score(target_test, target_test_pred))

Precision:  0.8260869565217391
Recall:    0.2714285714285714
```

We get a precision of about 82% and a recall of about 27%. So of all employees in the test set who have already left the company, we found 27%. That's not very high, but 82% of our predictions that an employee will leave the company are correct. So our model is relatively cautious in predicting that someone will leave the company, but when it does predict this, it's pretty accurate.

Take a few minutes to evaluate whether this is a good model.

You could also consider a grid search to get better results. But that's not the focus of this chapter. We'll look at more global methods to interpret a model.

**Congratulations:** You got to know the *feature importance* for random forests as a first example of global interpretation of a black box model. Now let's take a look at the *permutation feature*

importance.

## Permutation feature importance

Unfortunately, the feature Importance is not as accessible for all models as it is for decision trees or random forests. For example, imagine a support vector machine with a polynomial `kernel` (see *Module 2, Chapter 4*). A critical part of this algorithm involves creating more features from existing features. Then it looks for a decision plane in an abstract, higher dimensional space. You can easily determine the *feature importances* of these new features, but you then have to work very hard to determine the *feature importances* of the original features (this is often not even possible). So model-specific methods don't seem to make sense here.

In these cases, you can fall back on the model agnostic methods. The first one we want to learn is what we call *permutation feature importance*.

The order is also important here: We want to find out how much a single feature contributes to our predictions by training two models. One is trained with all the features as usual and provides the "real" results. The other model is also trained with all the features, whereby the connection between a single feature and the target is destroyed. Then you compare the predictions. The greater the deviations, the more important the feature is.

Technically, the following steps are performed:

1. The model, including hyperparameters, is fitted to the data as usual.
2. The prediction quality  $V_{\text{orig}}$  is calculated with whichever metric you select.
3. The values of a single feature, i.e. a single column, are mixed randomly, so the link between this feature and the target is lost. No values are deleted or generated, they are only shuffled, so to speak.
4. The new prediction quality  $V_{\text{perm}}$  is calculated based on the new data, i.e. with the permuted (mixed) column.
5. The feature importance is calculated either as the ratio  $\frac{V_{\text{orig}}}{V_{\text{perm}}}$ , or as the difference  $V_{\text{orig}} - V_{\text{perm}}$ . 6 Steps 2-5 are repeated for each feature and the final result is presented as a data story.

Now let's go through the steps one by one. We've already completed step 1, the fitted model is in `model_rf`. For step 2, we take the simplest metric in classification: accuracy. Calculate the accuracy for the test data and store it as `acc_orig`.

```
In [6]: from sklearn.metrics import accuracy_score
acc_orig = accuracy_score(target_test, target_test_pred)
acc_orig
```

```
Out[6]: 0.8752834467120182
```

We get an accuracy of over 87%. This is the same as  $V_{\text{original}}$ . Now use the `my_df.copy()` method to create a copy of `features_test`. Name it

```
features_test_perm.
```

```
In [7]: features_test_perm = features_test.copy()
```

Now we want to shuffle the values in one of the columns. In *Unbalanced Target Categories (Module 2, Chapter 3)* you learned about the `my_df.sample()` method to obtain samples of a `DataFrame`. Now we'll use `my_Series.sample()` in the same way. We need the parameters `frac`, `replace` and `random_state`. `frac` returns the size of the sample as a proportion of all the values in the column. With `frac=1`, you get all values back. With `replace=False`, we make sure that no values are drawn multiple times. This way no values are created or deleted. Select the `'age'` column and shuffle it. Store the result as `age_series_perm` and print the first 5 values. Compare the result with the first 5 values from the same column of `features_test`. Use `random_state=0` when mixing.

```
In [18]: age_series_perm = features_test_perm.loc[:, 'age'].sample(frac=1, replace=False, random_state=0)
age_series_perm.head()
```

```
age_series_perm = features_test_perm.loc[:, 'age'].sample(frac=1, replace=False, random_state=0)
print(age_series_perm[:5], features_test.age[:6])
```

```
361    46
249    31
271    31
434    55
397    40
Name: age, dtype: int64 0    36
1     33
2     35
3     40
4     29
5     30
Name: age, dtype: int64
```

As you can see, not only the values but also the row names are shuffled. If we add this column back to `features_test_perm` like this, the column would look like it did before we shuffled it, because `pandas` recognizes where the row names belong. So we have to reset the index. We can achieve this with the `my_series.reset_index()` method. However, it adds the old index as a new column. To prevent this, we have to use the `drop=True` parameter.

Reset the index of `age_series_perm` and print the result.

```
In [19]: age_series_perm = age_series_perm.reset_index(drop=True)
```

The index now ranges from 0 to 440. Now step 4. Overwrite the `'age'` column of `features_test_perm` with `age_series_perm`. Then generate the predictions for `features_test_perm` and calculate the accuracy. Save the accuracy as `acc_age`.

```
In [20]: features_test_perm.loc[:, 'age'] = age_series_perm

target_test__pred_perm = model_rf.predict(features_test_perm)

acc_age = accuracy_score(target_test, target_test__pred_perm)
```



```
print('accuracy with age permuted', acc_age)
```

accuracy with age permuted 0.8639455782312925

Now for Step 5: The difference is easier to interpret, while the ratio ensures that the feature importance can be compared across different problems. Here we'll use the difference for the direct interpretation. How high is it?

```
In [22]: acc_orig - acc_age
```

```
Out[22]: 0.01133786848072571
```

Shuffling 'age' only makes the prediction slightly worse. So the feature importance of the 'age' feature is small.

What happens if we try this for the other columns. Go through all the columns in `features_test_perm` in a loop and permute them. Then calculate the differences from the original accuracy on the test data. Save these differences as a list called `perm_importances`.

**Important:** After each step you should reset the permuted column to its original state!

```
In [25]: perm_importances = []
features_test_perm = features_test.copy()

for x in features_test_perm.columns:
    #permute feature column
    newCol_Shuffle = features_test_perm.loc[:,x].sample(frac=1, replace=False, random_s
    features_test_perm.loc[:, x] = newCol_Shuffle.reset_index(drop=True)

    target_test__pred_perm = model_rf.predict(features_test_perm)

    acc_perm = accuracy_score(target_test, target_test__pred_perm)

    perm_importances.append(acc_orig-acc_perm)

    # reset permutation
    features_test_perm.loc[:, x] = features_test.loc[:, x]

perm_importances
```

```
Out[25]: [0.01133786848072571,
0.0022675736961451642,
0.0022675736961451642,
0.015873015873015928,
0.006802721088435382,
0.01133786848072571,
0.006802721088435382,
0.020408163265306145,
0.0022675736961451642,
0.0,
0.018140589569161092,
0.006802721088435382,
0.0,
0.01133786848072571,
0.0045351473922903285,
0.0022675736961451642,
0.0045351473922903285,
0.01133786848072571,
-0.006802721088435382,
0.0045351473922903285]
```

Visualize `perm_importances` in a bar chart. Assign the values to the corresponding column names.

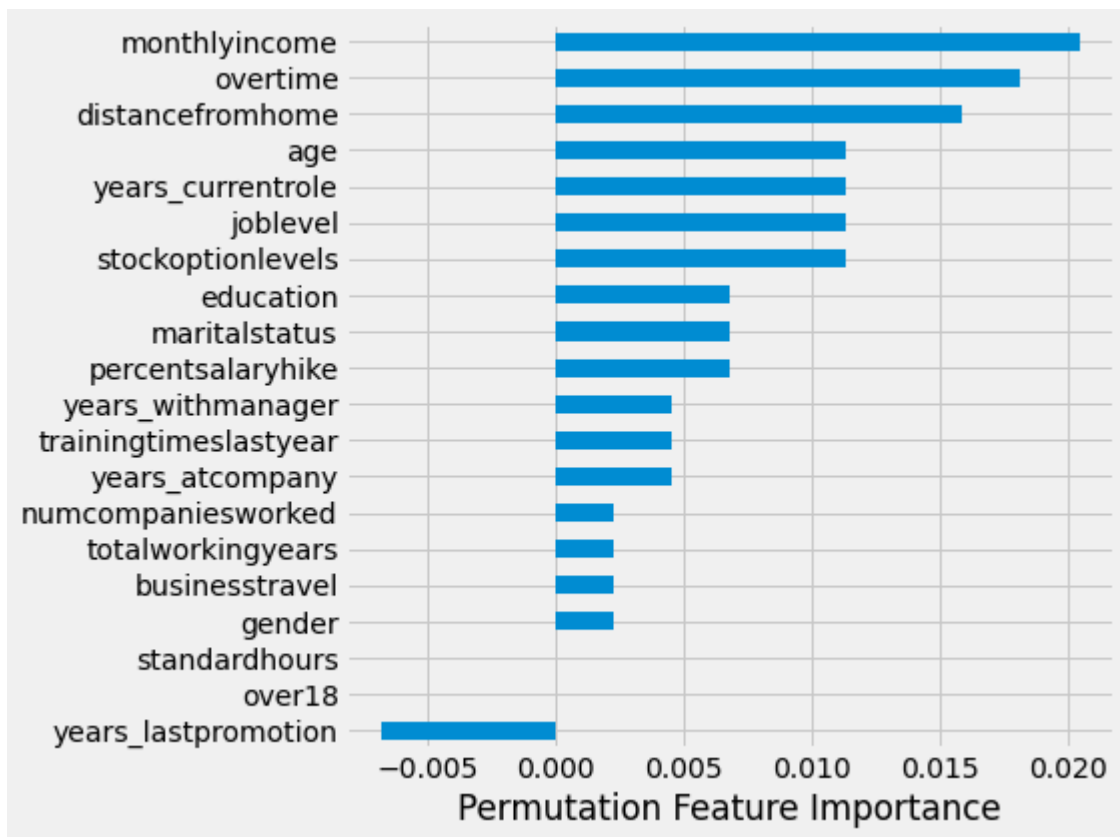
```
In [26]: import matplotlib.pyplot as plt

plt.style.use('fivethirtyeight')


fig, ax = plt.subplots(figsize=(8, 6))
ax.set_xlabel('Permutation Feature Importance')

# Convert feature importance array into a series and visualize
perm_importance = pd.Series(perm_importances, index=features_test.columns).sort_values
perm_importance.plot(kind='barh', ax=ax)

fig.tight_layout();
```



Your bar chart should look something like this:

 Permutation Feature Importance

As you can see, some of the values are negative. These columns misled our model, i.e. the model's prediction performance improves if these columns have no connection to the target. If we remove them, we get a slightly increased accuracy. Otherwise, the picture is relatively similar to the previous feature importance. 'monthlyincome' is also identified as the most important feature here. However, 'age' is identified as less important. We can also see that no feature makes a very large contribution to reducing the error. The accuracy if we shuffle the monthlyincome column is only about 1.8% lower. One advantage of this method is that we get the contributions of a column in absolute values, unlike the built-in feature importance, where all contributions are already normalized to 100%. The values may differ for other permutations. For increased accuracy, you can repeat this process several times and use calculate the average for the results.

You can use the permutation feature importance with any metric or model to gain insight into the model, because the method only works with the end results and does not take model-internal properties into account. Truly model agnostic!

**Note:** While the permutation feature importance may seem like a universal technique for global interpretation, it has important drawbacks.

1. Assume that two features are strongly correlated. If you now shuffle one of the two, you create data that could never exist. The permuted prediction would therefore never be

relevant and would be much lower than for uncorrelated features. So the feature importance would be valued too highly. you should always keep this in mind when making recommendations.

2. It's unclear whether the permutation feature importance should be determined from the training or test data. In our case, we determined it from the test data, but you could criticize the fact that the model was always trained on unpermuted data, and so it's unclear how realistic the feature importances are. So you should test several possibilities (also shuffle the training data) to get a better overview. The permutation feature importance is thus always only a statement about the tendency.

**Congratulations:** You have got to know permutation feature importance, your first model agnostic method, which you can use to interpret your models. Management thanks you for your hard work!

**Remember:**

- There are model-specific and model agnostic methods for global interpretability.
- You can use permutation feature importance to estimate the importance of the features
- Shuffling the values in a column destroys the direct link between the column and the target variable

---

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](mailto:support@stackfuel.com).

---