Feature Importance

Module 3 | Chapter 1 | Notebook 2

Finally, we looked at the local interpretation of the decision tree. In doing so, you traced which properties lead to a certain classification for individual data points. Now we want to take a global view and find out how important individual features are for our model.

In this lesson you will learn:

- How you can calculate the importance of features in decision trees.
- How you can represent the importance of features.
- How data stories work

Determining feature importance in the decision tree

Scenario: You work for an international global logistics company. Due to the tense situation on the labor market, the company is finding it increasingly difficult to attract new talent. For this reason, management has decided to try and limit the number of employees who leave. You already developed a model that predicts which employees are likely to want to leave the company so that measures can be taken to encourage them to stay. Now management wants to develop a strategy to improve morale in the company. So they are interested in which factors would have the greatest effects.

The question often arises - which features of the data are particularly important. This is called **Feature Importance**. There are several advantages of being able to quantify this. It can help us to better understand the data and extract recommendations for action from the model. Furthermore, removing unimportant features can lead to an improvement of the model, especially if they tend to overfit (too many features can lead to overfitting).

There are several options when it comes to measuring the feature importance. In this chapter we'll concentrate on decision trees. Feature importance here is the same as how much a feature contributes to reducing the *Gini impurity*. Before we take a closer look at this, let's import the data:

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

Out[1]:

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

5 rows × 21 columns

The code for that looks like this:

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

Column number	Column name	Type	Description
12	`'percentsalaryhike'`	continuous (`int`)	Salary increase in percent within the last twelve months
13	`'standardhours'`	continuous (`int`)	contractual working hours per two weeks
14	`'stock option levels'`	categorical (ordinal, `int`)	options on company shares: very many (`4`), many (`3`), few (`2`), very little (`1`), none (`0`)
15	`'trainingtimeslastyear'`	continuous (`int`)	Number of training courses taken in the last 12 months
16	`'totalworkingyears'`	continuous (`int`)	Number of years worked: Number of years on the job market and as an employee
17	`'years_atcompany'`	continuous (`int`)	Number of years at the current company Number of years in the current company
18	`'years_currentrole'`	continuous (`int`)	Number of years in the current position
19	`'years_lastpromotion'`	continuous (`int`)	Number of years since the last promotion
20	`'years_withmanager'`	continuous (`int`)	Number of years working with current manager

Each row in `df_train` represents an employee.

Run the following code cell to divide the data into feature matrix `features_train` and target vector `target_train` and train the decision tree from the last exercise

In [2]:
```
# split training data into features and target
features_train = df_train.drop('attrition',axis=1)
target_train = df_train.loc[:, 'attrition']

# instantiate and fit a decision tree
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(class_weight='balanced', max_depth=3, random_state=0, m
model.fit(features_train, target_train)
```

Out[2]:
```
DecisionTreeClassifier(class_weight='balanced', max_depth=3,
                       min_samples_leaf=20, random_state=0)
```

This gives us the following decision tree:

 "New Decision Tree"

Each time a node is split into two more, the *Gini impurity* for the data points of the nodes is

reduced. However, there is another related value is that is relevant for feature importance - the **Gini importance**.

The higher the *Gini importance*, the more important the feature is; *Gini impurity* and *Gini importance* are related, but have different meanings. The former can be interpreted as the probability of misclassification of a group of data points, while the latter measures the relevance of a feature, if this probability is reduced.

Deep Dive: How is the ***Gini importance*** calculated in `DecisionTreeClassifier`?

We need the following sizes first:

- **Weight**: Each **node** is assigned a **weight**, which we call $\mathbf \omega_\mathbf{node}$ (a small omega from the Greek alphabet). The **weight** is the ratio of ***samples* in the node** and the **total number of *samples* in the data set**. For example, the *joblevel*-**node** has 282 data points and would therefore carry a **weight** of $\omega_{joblevel} = \frac{282}{1029}$.
- The ***Gini impurity*** for each **node**. We use the variable $\Gamma_{\mathbf{Node}}$ (a big gamma from the Greek alphabet) and can read it directly from the decision tree.

Once these values have been determined for all **nodes**, the ***Gini importance***, which we call $\mathbf{G}_\mathbf{Node}$, can be calculated as follows:

$$\begin{equation*} \mathbf{G}_{\mathbf{node}} = \omega_{\mathbf{node}} \cdot \Gamma_{\mathbf{node}} - \omega_{\mathbf{left\ node}} \cdot \Gamma_{\mathbf{left\ node}} - \omega_{\mathbf{right\ node}} \cdot \Gamma_{\mathbf{right\ node}} \end{equation*}$$

In other words: The ***Gini importance*** is the difference between the **weighted *Gini impurity*** of a node and the **weighted *Gini impurities*** of the **left** and **right nodes**.

If a feature is involved in **several decision rules**, the ***Gini importances*** for this feature are **added up**. Once you've calculated all the ***Gini importances***, you can calculate the ***feature importances*** by determining the **percentage** of each ***Gini importance*** associated with each feature from the **sum** of all ***Gini importances***. A simple rule of three is then sufficient for this :).

Luckily, we don't need to calculate the *feature importances* ourselves, as `DecisionTreeClassifier` has already done this when it fitted the model to the data. You can find them in `my_model.feature_importances_` attribute. Print them.

In [3]:
```
model.feature_importances_
```

Out[3]:
```
array([0.07100213, 0.        , 0.        , 0.        , 0.        ,
       0.20155824, 0.14101587, 0.06135996, 0.        , 0.        ,
       0.32988041, 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.19518339])
```

Each value represents the *feature importance* of a feature. Most of them are 0, because our tree uses only 7 decision rules in total. The other features don't contribute to a reduction of the *Gini impurity*. `monthlyincome` is used in 2 decision rules, so there are only 6 values remaining that are greater than 0.

Now which feature is the most important? To do this, assign the column names to the values of the *feature importances* and store them in the `feature_importance` variable. Sort the values in descending order and print them.

Tip: Use `pd.Series()` in conjunction with the parameters `data` and `index`. You can sort these with the `my_Series.sort_values()` method.

In [4]:
```
feature_importance = pd.Series(model.feature_importances_, index=features_train.columr
feature_importance.sort_values(ascending=False)
```

Out[4]:
```
overtime                0.329880
joblevel                0.201558
years_withmanager       0.195183
maritalstatus           0.141016
age                     0.071002
monthlyincome           0.061360
stockoptionlevels       0.000000
years_lastpromotion     0.000000
years_currentrole       0.000000
years_atcompany         0.000000
totalworkingyears       0.000000
trainingtimeslastyear   0.000000
distancefromhome        0.000000
standardhours           0.000000
education               0.000000
gender                  0.000000
over18                  0.000000
numcompaniesworked      0.000000
businesstravel          0.000000
percentsalaryhike       0.000000
dtype: float64
```

The most important features for us are:

Column Number	Feature	*Feature Importance*
11	`'overtime'`	33%
6	`'joblevel'`	20%
20	`'years_withmanager'`	20%
7	`'maritalstatus'`	14%
1	`'age'`	7%
8	`'monthlyincome'`	6%

This representation would also be suitable for presenting your initial results to a superior!

Congratulations: Now you know how to measure the importance of a feature for the decision tree. The company is very pleased with your results. Now they would like a nice visualization with the results.

Displaying feature importance in the decision tree with `matplotlib`

Now let's come to the last point of each model interpretation - the **data story** with a meaningful visualization.

A nice visualization makes the result even more appealing. So far, we have usually been content with visualizations as long as we can create them quickly and understand they show. We used them primarily for exploratory data analysis. Creating good visualizations is an important part of data storytelling. Humans can understand visual information very quickly and retain the context. Everybody knows that a good image sticks in the audience's mind, and that images are essential to convince somebody about a subject. Although data scientists don't have to be designers, we can significantly improve our visualizations with a few strategies. There are three relevant questions you should ask yourself:

What do I want to achieve?

This is the central question of visualization. The whole image should work towards this goal, if possible. In our example, the goal could be to convince management that a great deal of employees who are overworked are looking for a new employer. We can use a concrete statement for the title to achieve this goal. Furthermore, `overtime` should be emphasized in particular.

Who is my audience and which medium am I using?

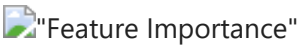
The terms we use should be based on the audience. Perhaps some technical terms or abbreviations are quite clear to you, while they are cryptic to people in another department. Is the image for an spoken presentation? Then it should be designed in such a way that it only supports your explanations and does not distract from you are saying. Is it a picture for a written report? Then, for example, there may be a little more text in the illustration because the audience has more time to look at it. What is the audience's background? Although everyone can understand a bar chart easily, understanding a box plot requires a more in-depth knowledge of statistics.

How do I make it easy for the audience?

Have you ever noticed that your attention is somewhat limited after a long meeting or report? Other people can feel the same way. It's therefore important to design the visualization in a way

that is easy to digest. This includes meaningful axis labels. The order of the components of an image is also important. The most important things should be found early when looking at the picture. People's gaze usually wanders from top to bottom and from left to right. Furthermore, using a logical order for elements (e.g. descending in importance) makes it easier to quickly understand the illustration. You can emphasize things by using colors. Colors can even be used to link elements across multiple images. This can be important if you want to shine a light on different aspects of the same categories. If possible, you should choose the colors so that they are still easily distinguishable for people with color blindness. Approx. 8% of all men have a decreased ability to see colors. The most common kind is red-green color blindness. For this reason, it's a good idea to avoid using red and green together.

Now let's look at how we can individualize a bar chart a bit with `matplotlib` so that you can take these aspects into account in the future. At the end we have the following figure:

 "Feature Importance"

Let's start with with a colorful bar chart.

Use the following cell for this

1. Import `matplotlib.pyplot` as `plt`.
2. Define the colors as a list `colors=['#99cced']*5 + ['#17415f']`. We use the same light blue color five times and a stronger dark blue to emphasize the most important feature. The list needs 6 entries, because we want to show 6 features.
3. Create *figure* and *axes* objects named `fig` and `ax`. It is best to make the image wider than it is high, as this makes it look more compact.
4. Select only the features in `feature_importance` that have an importance greater than 0. Otherwise we get a lot of column names without bars, which is just distracting.
5. Sort the values in ascending order so that the most important value is displayed at the top. Then people will see this first.
6. Create a bar chart. Pass the `colors` list to the `color` parameter. With the `width` parameter, you can vary the width of the bars. A value of 1 corresponds to the total width between the bars, so there are no more gaps. We used the value 0.8. But this is a matter of personal taste.

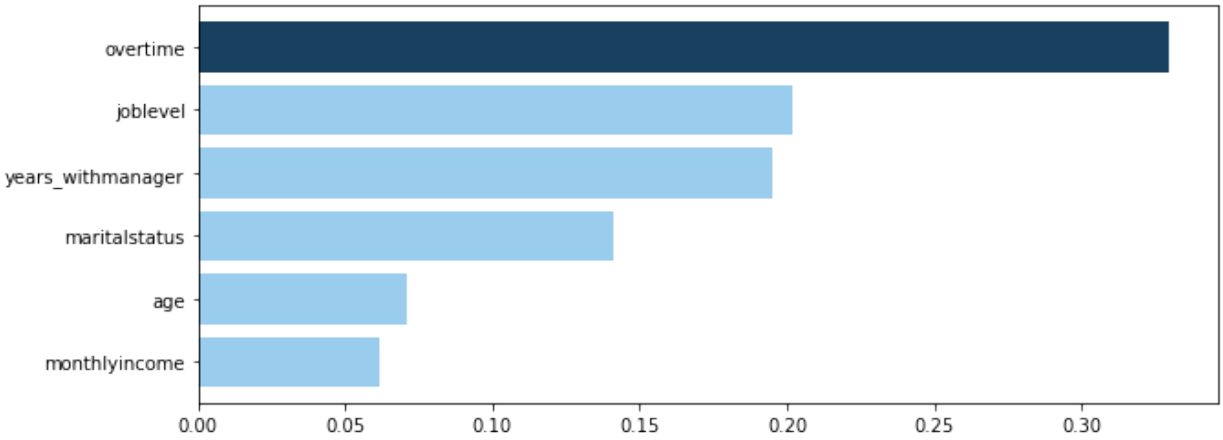
In [5]:
```
import matplotlib.pyplot as plt

colors=['#99cced']*5 + ['#17415f']
colors = ['#99cced']*5 + ['#17415f']

mask = feature_importance > 0
feature_importance = feature_importance.loc[mask].sort_values()

fig, ax = plt.subplots(figsize=(10, 4))

feature_importance.plot(kind='barh', color=colors, width=0.8);
```



Next we'll add the labels. If we want to influence the size and position of the labels, we can do this well with the methods `my_ax.set_title()`, `my_ax.set_xlabel()` and `my_ax.set_ylabel()`. They can take different parameters to style the text. For `my_ax.set_title()`, we used the following:

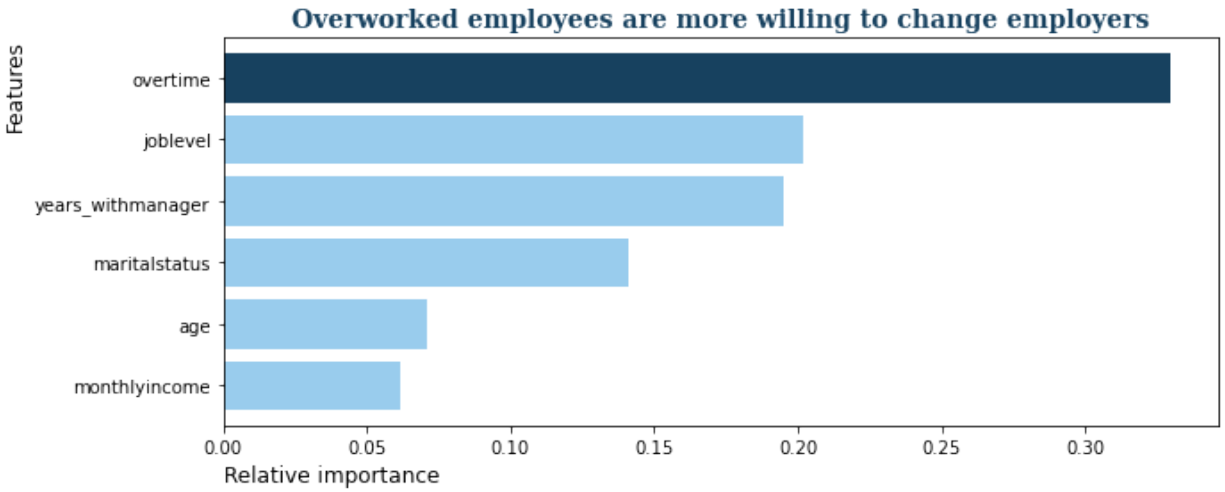
- `label='Overworked employees are more willing to change employers'`
- `family='serif'` for a font with serifs
- `color=colors[-1]`, so that the font color matches the top bar color, to make the reference to the `overtime` column clear
- `weight='semibold'` for a thicker font
- `size=14` for a larger font

Provide a meaningful title and understandable labels for the x and y axis. For `my_ax.set_xlabel()` and `my_ax.set_ylabel()` also use the parameters `position=[0, 0], horizontalalignment='left'` and `position=[0, 1], horizontalalignment='right'` respectively. In this way, the axis labels are flush with the edges of the image. This supports the reading flow a little. Then output `fig`.

Important: If you execute the code cell more often, `matplotlib` will "draw over" the old image without deleting it. If you want to change something, execute all cells starting from the cell where `fig` is defined.

In [6]:
```
ax.set_title(label='Overworked employees are more willing to change employers',
             family='serif',
             color=colors[-1],
             weight='semibold',
             size=14)
ax.set_xlabel('Relative importance',
              size=12, position=[0, -2],
              horizontalalignment='left')
ax.set_ylabel('Features',
              size=12, position=[0, 1],
              horizontalalignment='right')
fig
```

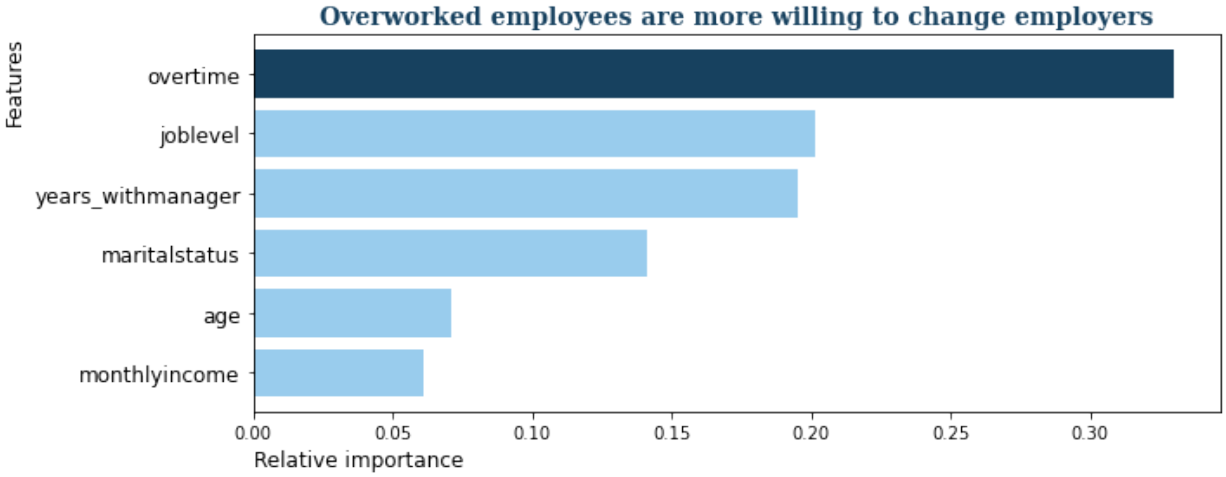
Out[6]:



Now execute the following code cell to make the y-axis tick labels a bit bigger and more readable. We can change the labels of the individual bars with `my_ax.set_yticklabels()`. With `my_ax.get_yticklabels()` you get a list of the existing labels.

In [7]:
```
ax.set_yticklabels(ax.get_yticklabels(), size=12)
fig
```

Out[7]:

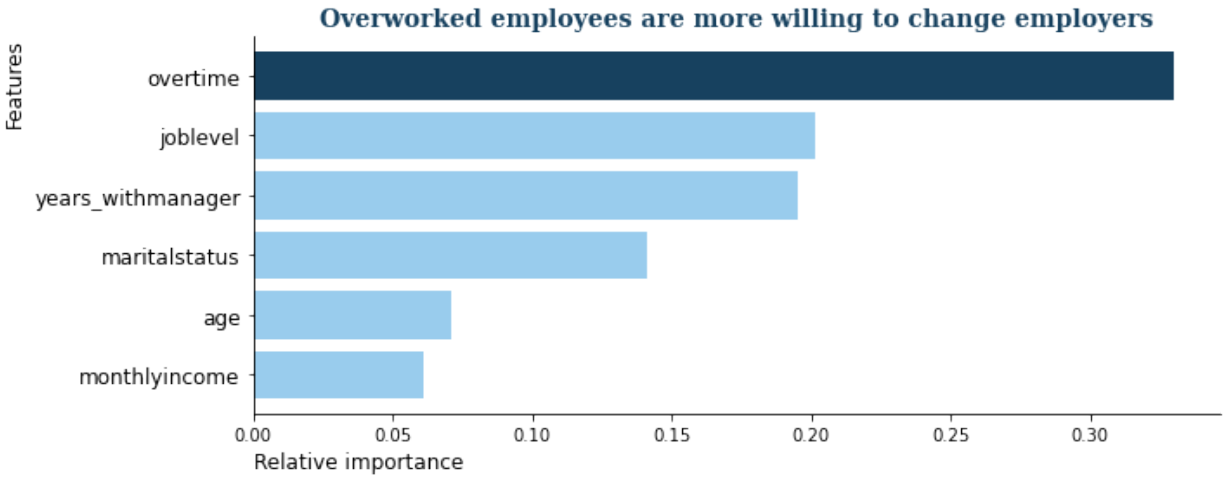


The frames of the picture are called `spines` in `matplotlib`. With `my_ax.spines` you get a `dict` with the keys `'top'`, `'bottom'`, `'left'` and `'right'`. The values of the `dict` are the objects of the corresponding frame lines. Use its `my_spine.set_visible(False)` method to remove the top and right border lines. End the code cell with `fig` to see the change in the image.

In [8]:
```
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

fig
```

Out[8]:

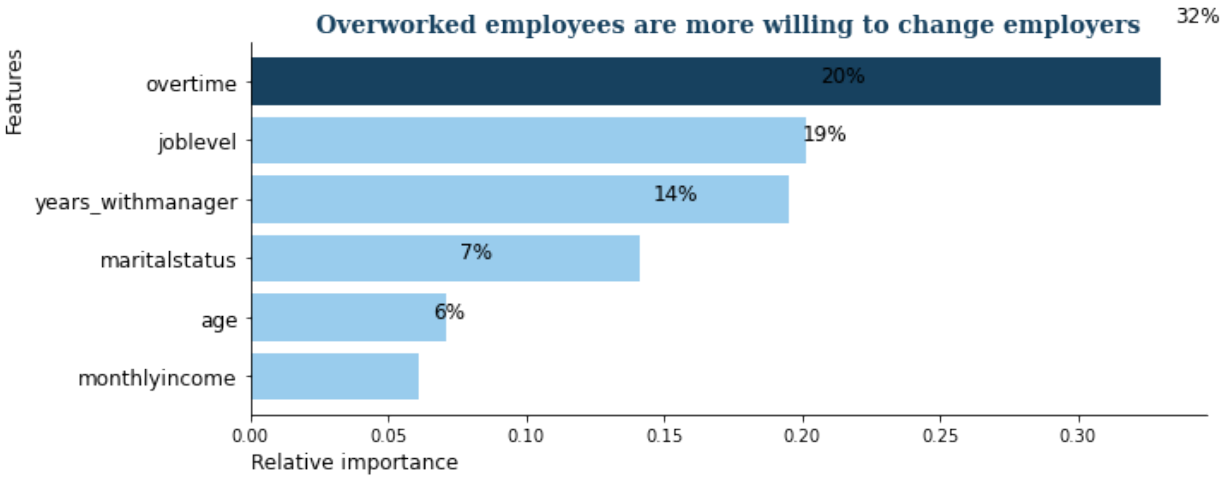


Now only the labels next to the bars are missing. They make sure that your eyes don't have to look downwards when you want to read the size for the bar.

Create a loop with 6 iterations, because we have 6 features. Create a text in each iteration with `my_ax.text()`. Pass the importance of the bar to the parameter `s`. You can determine the position with the parameters `x` and `y`. The y-values of the bars are integers from 0 to 5. The end of a bar is determined on the x-axis by its value. Add a small number to place the text a little further to the right. The `size` and `color` parameters control the size and color of the text. You should specify `color='black'` for all bars (but you are welcome to experiment with different combinations here).

In [9]:
```
for idx in range(len(feature_importance.index)):
    ax.text(s='{}%'.format(int(100*feature_importance.iloc[idx])),
            x=feature_importance.iloc[idx]+0.005,
            y=idx+1,
            size=12,
            color='black')
fig
```

Out[9]:

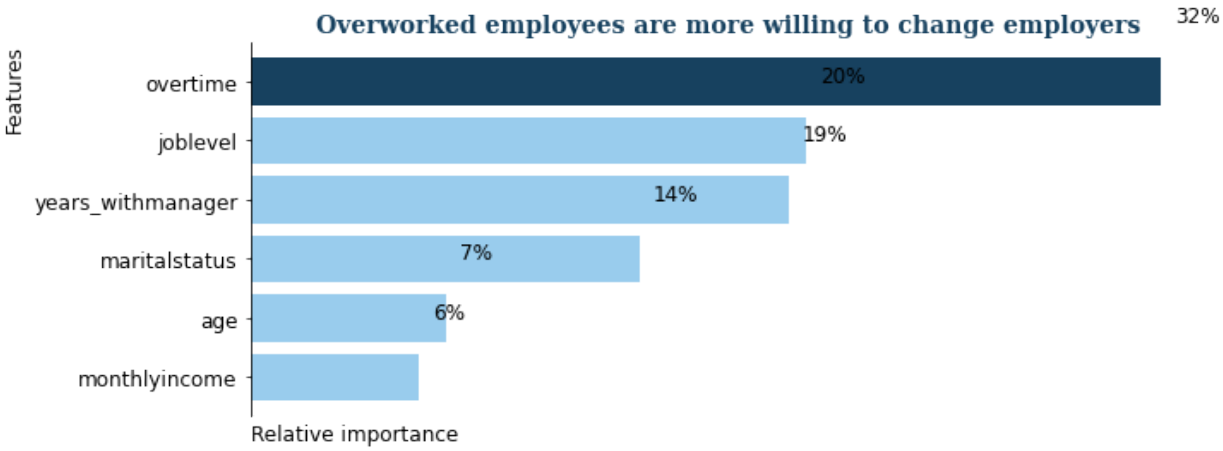


Now the x-axis is redundant and we can remove it.

In [10]:
```
#remove x-ticks and spine
ax.spines['bottom'].set_visible(False)
ax.set_xticks([])
```

```
ax.set_xticklabels([])
fig
```

Out[10]:



Congratulations: You have refreshed your `matplotlib` skills! `matplotlib` is a good foundation for creating images with Python. But there are also other visualization modules, and we're going to look at one more. With the help of the graphic, we can give the logistics company convincing recommendations!

Remember:

- The *Gini importance* measures the weighted contribution a feature makes in reducing false predictions.
- You can get them from the decision tree `my_model.feature_importance_`.
- Make your images as user-friendly as possible.

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.
