Logistic Regression Recap

Module 2 | Chapter 3 | 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 2. The relevant lessons for each section are clearly marked.

Logistic Regression

In the last chapter you learned about a new classification algorithm: logistic regression. It's used to predict categories. This may come as a surprise now that the word "regression" would indicate that it predicts continuous values. But don't let that fool you. Logistic regression is used to predict classes, similar to k-Nearest-Neighbors (*Module 1, Chapter 2*). Logistic regression calculates the probability of a data point belonging to a class.

Probabilities have to come somewhere between 0 (0%) and 1 (100%). Logistic regression ensures this by using the following formula for in its calculation (see *Logistic Regression (Chapter 2)*):

\begin{equation*} Probability = \frac {1} {1 + e^{-(intercept + (slope \cdot Feature))}}
\end{equation*}
The Euler number $e$ in the denominator is approximately 2.72

If you try different values for feature values and arrange them on the x-axis, the corresponding y-values will result in an S-shaped curve. The y-values represent the probabilities of data points belonging to the reference category.

The Logit Function

Note that the curve does not go below `0.0` (0%) and does not go above `1.0` (100%).

Now let's look at logistic regression with a small example. To do this, we'll use medical data that has been collected to predict whether a person has heart disease.

In [1]:
```python
# import pandas and read training data
import pandas as pd

df_train = pd.read_csv("heart_data_train.csv")
df_train.head()
```

Out[1]:

	age	sex	chest pain type	resting blood pressure	serum cholestoral	fasting blood sugar	target
0	70.0	m	stabbing	130.0	322.0	normal	1
1	67.0	f	burning	115.0	564.0	normal	0
2	57.0	m	dull	124.0	261.0	normal	1
3	64.0	m	stabbing	128.0	263.0	normal	0
4	74.0	f	dull	120.0	269.0	normal	0

The code for that looks like this:

Column number	Column name	Type	Description
0	`'age'`	continuous (`int`)	The patient's age in years
1	`'sex'`	categorically	gender (`'m'` : male, `'f'` : female)
2	`'chest pain type'`	categorical	Type of chest pain
3	`'resting blood pressure'`	continuous (`float`)	Resting blood pressure in mmHg.
4	`'serum cholestoral'`	continuous (`float`)	Cholesterol value in mg/dl
5	`'fasting blood sugar'`	categorical	Blood sugar level with an empty stomach (`'high'` : over 120 mg/dl, `'normal'` : under 120 mg/dl)
6	`'target'`	categorical	Heart disease is present (`1`) or not (`0`)

Each row in `df_train` represents a patient.

Let's take a quick look at the `dtypes` of `df_train`. Several columns still need to be converted. You will notice that our target variable `'target'` is categorical according to the data dictionary. However, we need it to be in a numerical format so that we can perform calculations with it.

In [2]:
```
print(df_train.dtypes)
df_train["chest pain type"] = df_train["chest pain type"].astype('category')
df_train["fasting blood sugar"] = df_train["fasting blood sugar"].astype('category')
df_train["sex"] = df_train["sex"].astype('category')
df_train["age"] = df_train["age"].astype('int')
```

```
age                          float64
sex                           object
chest pain type               object
resting blood pressure       float64
serum cholestoral            float64
fasting blood sugar           object
target                         int64
dtype: object
```

The logistic regression is contained in `sklearn.linear_model`. Run the following cell to fit it to the numerical data.

In [3]:
```
# Step 1: Select model
from sklearn.linear_model import LogisticRegression

# Step 2: Instantiate model
model = LogisticRegression(solver='lbfgs', max_iter=1000, random_state=42)

# Step 3: Create features and target
cols_num = ['age', 'resting blood pressure', 'serum cholestoral']
features = df_train.loc[:, cols_num]
target = df_train.loc[:, 'target']

# Step 4: Fit model
model.fit(features, target)
```

Out[3]:
```
LogisticRegression(max_iter=1000, random_state=42)
```

As you can see, we'll use the `lbfgs` algorithm again for the logistic regression. This is a `solver` (also called an optimizer) that can handle problems with multiple classes in small datasets. A `solver` tries to find parameter weightings that minimize the cost function in order to find the optimal model coefficients at the end. This `max_iter` parameter determines the maximum number of iterations that the `solver` needs for convergence. To predict the categories directly, we can use the `my_model.predict()` method. Run the following cell to read in the test data set, make predictions for it and save them in `df_pred_test`:

In [4]:
```
# read test data and create features
df_test = pd.read_csv("heart_data_test.csv")
features_test = df_test.loc[:, cols_num]

# create DataFrame with prediction
pred_test = model.predict(features_test)  # predict test data

# create DataFrame with one column named prediction
df_pred_test = pd.DataFrame(pred_test, columns=['prediction'])

df_pred_test.head()
```

Out[4]:

	prediction
0	0
1	0
2	0
3	1
4	1

As you can see, the categories are predicted directly. For the data of the people at row indices 3 and 4, we get the prediction that they have heart disease.

How do we get the corresponding probabilities? It turns out that you can use the `my_model.predict_proba()` method. The result is an array containing the probability that each data point belongs to one of the categories. The probabilities of belonging to category 1 (heart disease present) are added to `df_pred_test` in the following cell.

In [5]:
```python
# predict probabilities and add them as new column
df_pred_test.loc[:, 'probability'] = model.predict_proba(features_test)[:, 1]

df_pred_test.head()
```

Out[5]:

	prediction	probability
0	0	0.459462
1	0	0.454835
2	0	0.478176
3	1	0.587920
4	1	0.537270

If the probability is less than `0.5`, then the row is assigned category `0`, otherwise category `1` is assigned.

Congratulations: You have used your second classification algorithm with logistic regression. You've recapped how you can recap the probabilities and at what threshold the predicted category changes.

ROC Curve

In principle, you could also choose a different threshold value instead of `0.5`. A lower threshold value results in more and more positive predictions and the recall increases. However, this also reduces the precision at the same time, since more and more of the predicted reference category cases turn out not to be.

You learned how to visualize this dynamic in *ROC AUC: Area under the ROC Curve (Chapter 2)*. The y-axis shows the recall. The x-axis shows a new metric: the false positive rate (FPR). This is how you calculate it:

\begin{equation*} false\ positive\ rate = \frac{false\ positive}{false\ positive + true\ negative} \end{equation*}

You can also derive this from the confusion matrix, and you calculate it like this:

confusion matrix false positive rate

Let's use the example of patient data to understand the precision, recall and false positive rate:

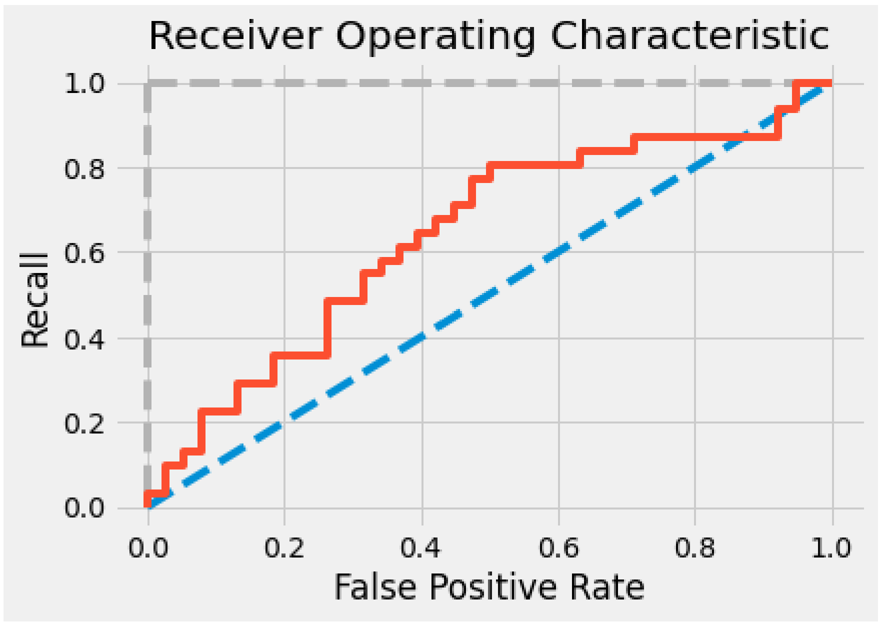
- Recall What percentage of people with heart disease was correctly identified?
- Precision What proportion of people who have been diagnosed with heart disease are actually sick?
- The false positive rate (FPR: What proportion of people who aren't ill have been incorrectly diagnosed with the disease?

The ROC curve describes how the recall and false positive rate change with the threshold value. The function `roc_curve()` from `sklearn.metrics` calculates the values of the curve. It needs the measured categories and the predicted probabilities. We get false positive rates, recall rates and the corresponding thresholds.

In [6]:
```
# calculate roc-curve
from sklearn.metrics import roc_curve

false_positive_rate, recall, threshold = roc_curve(df_test.loc[:, 'target'], df_pred_t
```

In order to get an impression of how different threshold values affect the model, we should draw the ROC curve.

In [7]:
```
# print roc-curve and curves for comparison

# module import and style setting
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')

# figure and axes intialization
fig, ax = plt.subplots()

# reference lines
ax.plot([0, 1], ls = "--")  # blue diagonal
ax.plot([0, 0], [1, 0], c=".7", ls='--')  # grey vertical
ax.plot([1, 1], c=".7", ls='--')  # grey horizontal

# roc curve
ax.plot(false_positive_rate, recall)

# labels
ax.set_title("Receiver Operating Characteristic")
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("Recall");
```



The curve gives us an idea of how good our model is at separating the classes. It becomes clearer how it does this when we know what the reference curves mean. The dotted grey reference line is the ideal of what you are aiming for: The recall (y-axis) is always 100%, no matter how high or low the false positive rate (x-axis) is. So we can identify all the people who are sick this way.

The blue dotted reference line is the performance if you are just guessing at random: If the hit rate increases, the false positive rate increases to the same extent. The proportion of correctly diagnosed diseases is therefore just as high as the proportion of falsely diagnosed diseases.

The red line belongs to our model. Our model is better than just guessing, as the curve is above the blue line. But she it's still far from perfect.

The question of how to set the threshold always depend on the specific application. In our example, we would like to detect people who are sick as accurately as possible. False positive diagnoses lead to healthy people undergoing therapy that isn't suitable for them and could cause health problems. To make this decision, it's important to know what the consequences are.

But in any case, we want to find the clearest possible distinction between the two categories. So that we don't always have to evaluate this visually, we use the area under the receiver operator curve: (ROC AUC)*.

The grey ideal line has a value of `1.0` because the entire area is filled in. The curve of the random guess algorithm in blue has a value of `0.5` because only half of the area is filled. A value below `0.5` suggests that the predictions will be better if you swap them round (`0` becomes `1` and `1` becomes `0`).

To determine the value of the red curve, we need `roc_auc_score()` from `sklearn.metrics` . The `roc_auc_score()` function needs the actual categories and the predicted reference category probabilities for them.

In [8]:
```
# calculate area under the curve
from sklearn.metrics import roc_auc_score

roc_auc_score(df_test.loc[:, 'target'], df_pred_test.loc[:, 'probability'])
```

Out[8]:
```
0.6400679117147707
```

This value is above `0.5` (64% to be precise), which means that it's already better than guessing at random. However, it's a long way from the ideal curve. This means that our model has some difficulty in distinguishing between healthy and sick patients. You probably wouldn't be able to use this model for a medical diagnosis.

Congratulations: You've recapped another way of evaluating category predictions with the false positive rate. This forms the basis for the receiver operator curve, which gives you a visual impression of your model's quality. To determine how sharply the model separates between categories, it's helpful to calculate the area under this curve.

Label encoding

So far we've only used the numerical columns as features. Because as long as the categories with `str` values such as `'high'` or `'normal'` as indicated in the `'fasting blood sugar'` column, they are not usable for logistic regression. Texts can't be interpreted numerically, so we can't assign any weight to them. But if we represent the categories with numbers like `0` and `1`, we can use them to make predictions. Converting categories into numbers is called label encoding.

If a categorical feature only has two categories like `'m'` and `'f'`, you can represent one category with `1` and the other with `0`. The easiest way to do this is to use the `my_series.replace()` method, which we described in *Categorical Features (Chapter 2)*. This takes a `dict`, where the *keys* are the entries to be replaced and the *values* are the new value entries. Convert all the binary categories to the numbers 0 and 1 by executing the following cell.

In [9]:
```
# replace values in column 'sex'
replace_dict_sex = {'f': 1, 'm': 0}
df_train.loc[:, 'sex'] = df_train.loc[:, 'sex'].replace(replace_dict_sex)
df_test.loc[:, 'sex'] = df_test.loc[:, 'sex'].replace(replace_dict_sex)

# replace values in column 'fasting blood sugar'
replace_dict_sugar = {'high': 1, 'normal': 0}
df_train.loc[:, 'fasting blood sugar'] = df_train.loc[:, 'fasting blood sugar'].replac
df_test.loc[:, 'fasting blood sugar'] = df_test.loc[:, 'fasting blood sugar'].replace(

df_train.head()
```

Out[9]:

	age	sex	chest pain type	resting blood pressure	serum cholestoral	fasting blood sugar	target
0	70	0	stabbing	130.0	322.0	0	1
1	67	1	burning	115.0	564.0	0	0
2	57	0	dull	124.0	261.0	0	1
3	64	0	stabbing	128.0	263.0	0	0
4	74	1	dull	120.0	269.0	0	0

Now there's a `1` where there used to be a `'high'` or an `'f'`. All `'m'` and `'normal'` values are now `0` values.

The *label encoding* for the `'chest pain type'` column could look like this:

- `"sharp"` -- `3` >
- `"stabbing"` --> `2`
- `"Burning"` --> `1`
- `"Dull"` --> `0`

If you use a feature like this for logistic regression, the categories would not be interpreted as such, but as continuous values. The categories of `'chest pain type'` do actually have an order. So this is ordinal data. In the lesson *Polynomial Regression (Module 1, Chapter 4)*, you saw that sometimes it can be advantageous to interpret them as measured values.

But this approach is not optimal if there is no real order. Then it is better to create a separate column for each category, containing only a `0` or a `1`. This formatting step is called *one-hot encoding*. We can implement this using `OneHotEncode()` of `pdpipe` (see *Preparing Categorical Features (Chapter 2)*). You pass a `DataFrame` to this function and specify which columns to encode. We should also set the parameter `drop_first=True` to avoid collinearity. Afterwards we just adapt the model to the training data and then transform it. This transformation should be carried out in a similar way without being fitted to the test set.

In [10]:
```
import pdpipe as pdp

# use one-hot encoding to separate chest pain types
onehot = pdp.OneHotEncode(["chest pain type"], drop_first=True)
df_train = onehot.fit_transform(df_train) #fit and transform train set
df_test = onehot.transform(df_test)

df_train.head()
```

Out[10]:

	age	sex	resting blood pressure	serum cholestoral	fasting blood sugar	target	chest pain type_dull	chest pain type_sharp	chest pain type_stabbing
0	70	0	130.0	322.0	0	1	0	0	1
1	67	1	115.0	564.0	0	0	0	0	0
2	57	0	124.0	261.0	0	1	1	0	0
3	64	0	128.0	263.0	0	0	0	0	1
4	74	1	120.0	269.0	0	0	1	0	0

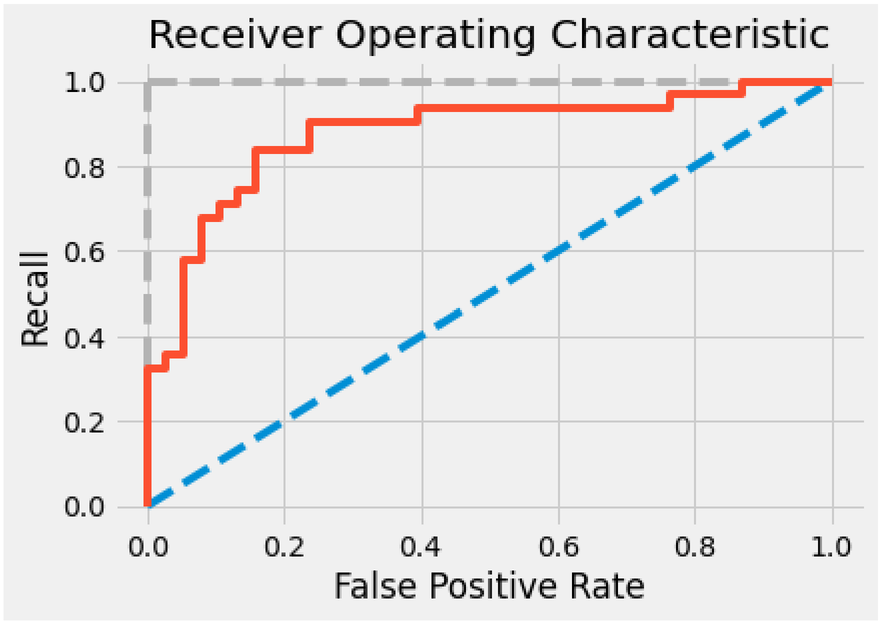
Now we can use all the columns except the target variable as features for logistic regression.

In [11]:
```
cols_features = [col for col in df_train.columns if not col == 'target']  # select all
features_train = df_train.loc[:, cols_features]  # select new features for training da
target_train =df_train.loc[:, 'target']

model.fit(features_train, target_train)
```

Out[11]:
```
LogisticRegression(max_iter=1000, random_state=42)
```

Has this improved the model?

In [12]:
```
# select new features for test data and calculate the roc-curve
features_test = df_test.loc[:, cols_features]
false_positive_rate, recall, threshold = roc_curve(df_test.loc[:, 'target'], model.pre

# figure and axes intialisation
fig, ax = plt.subplots()

# reference lines
ax.plot([0, 1], ls = "--")  # blue diagonal
ax.plot([0, 0], [1, 0], c=".7", ls='--')  # grey vertical
ax.plot([1, 1], c=".7", ls='--')  # grey horizontal

# roc curve
ax.plot(false_positive_rate, recall)

# labels
ax.set_title("Receiver Operating Characteristic")
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("Recall");
```



By using the categorical features, the red curve has got a lot closer to the ideal. You can see this numerically in the area under the curve:

```
In [13]: roc_auc_score(df_test.loc[:, 'target'], model.predict_proba(features_test)[:, 1])  # c
```

```
Out[13]: 0.8760611205432937
```

We get a better ROC AUC metric of about 88%.

Congratulations: You have recapped how to prepare categorical features for use. The one-hot encoding method enables you to prevents categories from being interpreted ordinally.

Regularization with logistic regression

It's very common to have a lot of features, especially after using *one-hot encoding*. This can very quickly lead to overfitting. You regularization can counteract this. `LogisticRegression` uses the `C` parameter for this. If we assign an extremely large value to `C`, such as a 1 followed by 42 zeros (`1e42`), it doesn't perform any regularization. If the `C` is small, the regularization has a high weight.

To understand how regularization works in logistic regression, it is worth looking back at ridge regression. In the lesson *Regularization (Module 1 Chapter 1)* you learned that a linear regression with regularization has two goals. For a ridge regression, they would be something like this:

- Keep the difference between predicted and actual target values as small as possible.
- Keep the sum of the squared slopes (e.g. $(slope_1)^2 + (slope_2)^2$) as small as possible.

The second objective is called regularization, or *shrinkage penalty*. This means that the model is punished for when the slopes are too big. The `alpha` parameter of `Ridge()` controls how much the second goal should be pursued.

With `alpha=0` the second objective (regularization) is ignored. Then the ridge regression would be a normal linear regression. With an infinitely high `alpha`, the first objective is disregarded. In this case all slopes are zero.

In the lesson *Optional: Logistic Regression as Linear Regression with Log Odds* you saw that logistic regression is a kind of alternative version of linear regression. So you can also think of logistic regression with regularization as a ridge regression that predicts log chances.

Confusingly, although `C` and `alpha` have the same function, they have the exact opposite effect. The following settings are equivalent:

Description	`alpha`	`C`
no regularization	`0`	`Inf`
little regularization	`0.5`	`2`
Standard regularization	`1`	`1`
a lot of regularization	`0.2	`5`
maximum regularization	`Inf`	`0`

So `C` and `alpha` are related to each other in the following way: `C = 1/alpha`. You can use a grid search to find the best value for `C`.

In *Regularization (Module 1, Chapter 1)*, you saw that it's essential to standardize the features so that they are treated equally during regularization. They will only be regularized equally if they are on the same scale.

For this we used `StandardScaler` from the `sklearn.preprocessing`. By default, it ensures that numerical and categorical features of different lengths are on the same scale.

In [14]:
```
# use StandardScaler to adjust the features
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
features_train_scaled = scaler.fit_transform(features_train)  # fit on training data c
features_test_scaled = scaler.transform(features_test)  # scale test data
```

Standardization and more higher regularization may help to improve our metrics a bit more. In this example, however, we see that with `C=0.5` we weren't able to improve our model significantly, but we actually made it worse due to overfitting. We get a ROC AUC metric of about 87%.

In [15]:
```
# define model with higher regularization
model = LogisticRegression(solver='lbfgs', C=0.5, max_iter=1000, random_state=42)

# fit model to training data
model.fit(features_train_scaled, target_train)
```

```
# calculate area under the curve
roc_auc_score(df_test.loc[:, 'target'], model.predict_proba(features_test_scaled)[:, 1
```

Out[15]:
```
0.8709677419354839
```

Congratulations: You've recapped how regularization for ridge and logistic regression is linked. We recommend using `StandardScaler` to bring categorical and numerical features onto the same scale.

Sanity checks

We can check how plausible our solutions are with what we call sanity checks. These help us interpret the models.

How can you be sure that you can trust your predictions? Both data science knowledge and domain knowledge are suitable here.

There are a few laws to observe in data science, and these should be reflected in the data:

- A machine learning model makes better predictions for its own training data better than for independent test data.
- If you change the training data, the predictions should also change. If they don't change, you may have forgotten to retrain the model.
- The predicted target values should look similar to the target values of the training data. For example, with classification the ratio between the categories should be approximately the same. If this is not the case, the training data may not be representative of the data used for the predictions.

On domain knowledge: A lot of industries experience seasonal fluctuations.

- For example in the medical field, there are flu epidemics in winter. A machine learning model should predict this kind of known seasonal effects.
- Another familiar example would be correlations between variables. For example, if you look at the time of use and the cost of calls, there should be a positive correlation between the two. If this is not reflected in the data, there may be an error.
- The third example would be the typical values of a data series. If you work on weather data, you can measure the temperature in degrees Celsius, Fahrenheit or Kelvin. If we use normal room temperature as an example, you should know the typical value range of a data series in a certain measurement unit thanks to your domain knowledge. Make sure that this area is actually range in the data. This way you know which unit of measurement is used and whether your data is plausible at all.

There are no limits on how you use sanity checks. The main thing is to check from time to time whether your analysis is going as you would expect it to. This way you can trust the results more in the end.

Remember:

- Logistic regression for the probabilities of target categories (classification)
- Prepare categories for use with `my_df.replace()` or `pdp.OneHotEncode()`
- `LogisticRegression` uses regularization by default (`C=1.0`) * The higher the `C` parameter in `LogisticRegression()` , the weaker the regularization
- For logistic regression models with regularization, you should always standardize the features in advance
- Solvers are algorithms that can help minimize the cost function.
- The ROC curve and the *ROC-AUC* metric are based on the predicted probabilities
- Use sanity checks to quickly see whether your analyses are plausible

---

The data is based on the published data from Dua, D. and Graff, C. (2019). UCI Machine Learning Repository http://archive.ics.uci.edu/ml. Irvine, CA: University of California, School of Information and Computer Science.