Evaluating Classification Performance

Module 1 | Chapter 2 | Notebook 4

In this lesson you will learn how to assess the model quality of classification algorithms. By this end of this lesson you will have learned about:

- The accuracy metric
- Confusion matrices
- Recall, precision and the F1 score

The accuracy of categorical predictions

Scenario: You work for a *smart-building* provider. In a new pilot project, you have been asked to predict whether a person is in the room. The training data is stored in *occupancy_training.txt*. The project's management is already satisfied that you were able to make predictions so quickly. Now they want to know how good the predictions are. The independent test data for this purpose is provided in *occupancy_test.txt*.

Let's import the training data now in order to get started quickly:

In [1]:
```
# module import
import pandas as pd

# data gathering
df_train = pd.read_csv('occupancy_training.txt')

# turn date into DateTime
df_train.loc[:, 'date'] = pd.to_datetime(df_train.loc[:, 'date'])

# turn Occupancy into category
df_train.loc[:, 'Occupancy'] = df_train.loc[:, 'Occupancy'].astype('category')

# define new feature
df_train.loc[:, 'msm'] = (df_train.loc[:, 'date'].dt.hour * 60) + df_train.loc[:, 'dat
```

The data dictionary for this data is as follows:

Column number	Column name	Type	Description
0	`'date'`	continuous (`datetime`)	time of the date measurement
1	`'Temperature'`	continuous (`float`)	temperature in ° celsius

Column number	Column name	Type	Description
2	`'Humidity'`	continuous (`float`)	relative humidity in %
3	`'Light'`	continuous (`float`)	Brightness in lux
4	`'CO2'`	continuous (`float`)	Carbon dioxide content in the air in parts per million
5	`'HumidityRatio'`	continuous (`float`)	Specific humidity (mass of water in the air) in kilograms of water per kilogram of dry air
6	`'Occupancy'`	categorical	presence (0 = no one in the room, 1 = at least one person in the room)
7	`'msm'`	continuous (`int`)	time of day in minutes since midnight

Now also import the test data from *occupancy_test.txt*, process it similarly to `df_train` and save it in a new `DataFrame` called `df_test`.

In [2]:
```
# data gathering
df_test = pd.read_csv('occupancy_test.txt')

# turn date into DateTime
df_test.loc[:, 'date'] = pd.to_datetime(df_test.loc[:, 'date'])

# turn Occupancy into category
df_test.loc[:, 'Occupancy'] = df_test.loc[:, 'Occupancy'].astype('category')

# define new feature
df_test.loc[:, 'msm'] = (df_test.loc[:, 'date'].dt.hour * 60) + df_test.loc[:, 'date']
```

How large are the training and test data sets? Print the number of rows and columns and of `df_train` and `df_test`.

In [3]:
```
print(df_train.shape)
print(df_test.shape)
```

```
(8143, 8)
(2665, 8)
```

The training data contains more than 8,000 points in time, while the test data contains only 2,665 points in time. This kind of ratio between training and test data is neither unusual nor problematic. It's more important that the training and test data are comparable. In our case, for example, that the data sets were collected under similar conditions.

Now use `sklearn` to fit a k-nearest neighbors model to the data. The nearest **three** neighbors should determine the predicted classification of a data point. Use the data from `df_train`. Save the `'CO2'` and `'HumidityRatio'` columns in the feature matrix `'features_train'`, which you should standardize afterwards (`features_train_standardized`). The `'Occupancy'` column forms the target vector (`target_train`). Then fit the model to the standardized training data. You don't need to generate predictions for the time being.

So follow the first four of the five steps:

1. Select model type: (`KNeighborsClassifier`)
2. Instantiate model with k=3 (`n_neighbors=3`)
3. Divide and standardize (`StandardScaler`) data into a feature matrix and target vector
4. Fitting the model to the training data
5. Make predictions with the trained model

In [4]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

model =  KNeighborsClassifier(n_neighbors=3)
scaler = StandardScaler()

features_train = df_train.loc[:,['CO2','HumidityRatio']]
target_train= df_train.loc[:,'Occupancy']

features_train_standardized = scaler.fit_transform(features_train)

model.fit(features_train_standardized, target_train)
```

Out[4]:
```
KNeighborsClassifier(n_neighbors=3)
```

Just like with a regression (see *Too Many Features: Overfitting (Module 1 Chapter 1)*), you should evaluate the model quality with independent test data. This means pretending not to know the target vector in the test data and then comparing the predicted data with the actual data.

Let's start by building a feature matrix of the test data and store it in `features_test`. Extract the `'CO2'` and `'HumidityRatio'` columns from `df_test` and store them in `features_test`. Also store the target vector of the test data in `target_test`.

In [5]:
```
features_test  = df_test.loc[:,['CO2','HumidityRatio']]
target_test    = df_test.loc[:,'Occupancy']
```

You should now standardize this feature matrix the same way you did `features_train` before. Use the scaler for this, which you probably called `scaler`. Since you have already used `scaler.fit()` with `features_train`, `scaler` already contains the relevant standardization settings. Apply these to `features_test` with `scaler.transform()`. Remember, **only ever fit to the training set!**

In [6]:
```
features_test_standardized = scaler.transform(features_test)
```

As in the previous lesson, we can now use `my_model.predict()` to predict the test data. Store the predicted classifications in the variable `target_test_pred`.

In [7]:
```
target_test_pred = model.predict(features_test_standardized)
```

As with regressions, model quality is measured by the similarity between the actual values (`target_test`) and the predicted values (`target_test_pred`).

But because we are not predicting continuous values, the metrics you've already encountered, *mean squared error* and R^2, will not help us here. These measure the similarity between measured and predicted **numerical values**. They are not useful for **categories**.

To evaluate classification models, we therefore need new model quality metrics. The simplest is accuracy. It indicates the proportion of correctly predicted classifications in relation to all classifications. The `accuracy_score()` function from the `sklearn.metrics` module will carry this out for you.

In [8]:
```python
from sklearn.metrics import accuracy_score
accuracy_score(target_test, target_test_pred)
```

Out[8]:
```
0.6626641651031895
```

It achieved an accuracy of 66%. In other words: At two out of three points in time, the model was able to detect the presence or absence of people in the room solely on the basis of CO2 values and the specific humidity. That's not bad for a first try.

Congratulations: You have recapped classification with k-nearest neighbors and have learned a first model quality metric for this kind of model: accuracy. The advantage of this is that it's very easy to understand. Unfortunately it is also almost always misleading. We'll see why now.

Confusion matrices

The accuracy metric treats all categories equally. This only makes sense if all categories appear in the data the same number of times. Is that the case here? Are there as many times when people were in the room as there are times when there was no one there?

If we look at how the target vectors are comprised by using pie charts, it becomes clear that this is not the case. There was no one in the room much more often than there was someone there.

In [9]:
```python
import matplotlib.pyplot as plt  # module import
plt.style.use('fivethirtyeight')  # set nice style

# initialise empty Figure with two Axes
fig, axs = plt.subplots(ncols=2,
                        figsize=[8, 4],
                       subplot_kw=dict(aspect="equal"))

# prepare training target vector for plotting
ct_train = pd.crosstab(index=df_train.loc[:, 'Occupancy'], columns='count')

# plot composition of training target vector
ct_train.plot(kind='pie',
              y='count',
              startangle=90,
              legend=False,
              labels=['Absence', 'Presence'],
              title='Training',
              ax=axs[0],
              colors=['red', 'blue'])
```

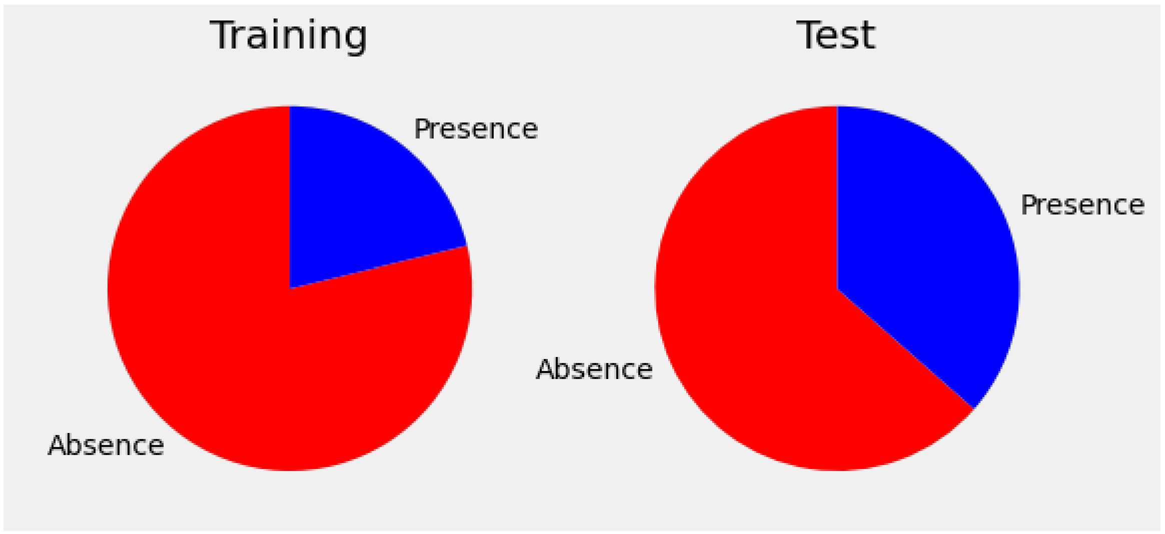
```
axs[0].set(ylabel=' ')  # remove y-label

# prepare test target vector for plotting
ct_test = pd.crosstab(index=df_test.loc[:, 'Occupancy'], columns='count')

# plot composition of test target vector
ct_test.plot(kind='pie',
             y='count',
             startangle=90,
             legend=False,
             labels=['Absence', 'Presence'],
             title='Test',
             ax=axs[1],
             colors=['red', 'blue'])
axs[1].set(ylabel=' ')  # remove y-label
```

Out[9]:

```
[Text(0, 0.5, ' ')]
```



There was no one in the room for about two thirds of the time in the test data. So if the k-nearest neighbors model simply **always** predicted a total absence of people, it would still be correct for two out of three points in time in the test data set.

The measured accuracy of 66% could therefore be due simply to the fact that the model has detected that there are more instances of absence than presence in the training data. Surprisingly often, classification models make use of these differences between categories and appear quite good. However, in reality they are then no better than classifications based on a pie chart.

In order not to be fooled by the model, we can use other model quality metrics to determine accuracy. They are based on what's called a confusion matrix. Let's take a look at it directly for our example:

Import the `confusion_matrix()` function directly from the `sklearn.metrics` module.

In [10]:

```
from sklearn.metrics import confusion_matrix
```

Like most model quality metrics, confusion matrices use the actual target values (`target_test`) and their predicted values (`target_test_pred`). Use these two variables together with `confusion_matrix()` to print the confusion matrix.

In [11]:
```
confusion_matrix(target_test, target_test_pred)
```

Out[11]:
```
array([[1049,  644],
       [ 255,  717]])
```

A confusion matrix is structured as follows:

The rows represent the actual categories. At the top there are the values of the classes that are actually "cross" and at the bottom are the values that are actually "tick". In our case, the top line shows points in time without people in the room and the bottom line shows points in time with people in the room.

The columns represent the predicted categories. In the left column you will find the prediction "cross" and in the right column the prediction "tick". In our case, the left column shows times when it is predicted that no one is present in the room, and the right column shows times when it is predicted that someone is in the room.

With this knowledge we can now fill the confusion matrix with our categories and values:

In the confusion matrix it is apparent that the row of actual instances where there was nobody there contains a lot more data points (1,049 + 644) than the row representing instances when people were actually present (255 + 717). This reflects the impression of the right pie chart in the upper graph: There are more times without people in the room than with.

We also see that the dark blue diagonal cells contain more data points (1,049 + 717) than the light blue cells off the diagonal (255 + 644). This means that 1,766 points in time were correctly classified and 899 were not. The correct classifications are also not concentrated in just one cell. So the model very often correctly predicts the absence **and presence** of people. So our fear that the model had tricked us by simply always predicting that no one was there was unfounded in this case.

As we mentioned earlier, all model quality metrics in this lesson can be derived from the confusion matrix. The accuracy, for example, is the sum of the diagonal blue cells divided by the sum of all cells.

\begin{equation*} accuracy = \frac{correct\ classified}{correct\ classified + false\ classified}
\end{equation*}
So you could derive the accuracy from the confusion matrix (`cm`) like this:

In [12]:
```
cm = confusion_matrix(target_test, target_test_pred)
accuracy = (cm[0][0] + cm[1][1]) / (cm[0][0] + cm[1][1] + cm[1][0] + cm[0][1])
accuracy
```

```
Out[12]: 0.6626641651031895
```

Congratulations: Now you know that accuracy is not a perfect model quality metric for classification models. Confusion matrices show the quality of all classifications in detail. You can typically get three values from them. We'll look at them next.

Precision, recall and the F1 score

In general, people are most interested in the positive predictions. They claim that a characteristic is present ("tick"). This could be an illness, a machine failure or even the presence of a person in the room. You then quickly need an impression of how to understand these positive predictions. Of course, it would be best if all positive features were predicted by the algorithm and all positive predictions were correct.

Figuratively speaking, you want to maximize the hits, or actual positive predictions in the lower right corner of the confusion matrix. confusion matrix hit

On the one hand, you want all the actually positive features to be recognized as such. This is expressed by the hit rate, which we refer to as *recall*.

On the other hand, you want the proportion of characteristics predicted as positive that are actually positive to be very high. This is expressed by the *precision*.

It's not surprising if you're a little confused at this point. Let's take a close look at *recall* and *precision* again. Pay attention to how they differ only in the population. Both deal with correctly positive predictions. You either look at the proportion of these predictions in the actually positive cases (*recall*) or in the positively predicted cases (*precision*).

In our case, the recall is the percentage of data points with people in the room where a person in the room was also correctly predicted.

\begin{equation*} recall = \frac{correct\ positive}{false\ negative + correct\ positive} \end{equation*}

confusion matrix recall

It is relatively easy to derive *recall* from the confusion matrix.

```
In [13]: recall = cm[1][1]/(cm[1][0] + cm[1][1])
         recall
```

```
Out[13]: 0.7376543209876543
```

74% of the instances with people in the room are recognized as such. The *recall* expresses whether you are missing something if you only focus on the positive predictions.

The opposite of that is the *precision*, which is the proportion of characteristics predicted as positive that are actually positive. In our case the relevance is therefore the proportion of data points with people in the room out of all the data points where people were predicted to be in the room.

\begin{equation*} precision = \frac{correct\ positive}{false\ positive + correct\ positive} \end{equation*}

confusion matrix precision

It is relatively easy to derive *precision* from the confusion matrix.

In [14]:
```
precision = cm[1][1]/(cm[0][1] + cm[1][1])
precision
```

Out[14]:
```
0.526818515797208
```

There was only actually at least one person in the room 53% of the times when a person was predicted to be in the room.

If you're confused at this point, let us put your mind at rest. Almost all budding data scientists feel this way. Here's a brief summary again:

\begin{equation*} recall = \frac{correct\ positive}{actually\ positive} \end{equation*}\begin{equation*} precision = \frac{correct\ positive}{positive\ predicted} \end{equation*}
It might help you to imagine recall and precision in terms of a pregnancy test. The *recall* describes the percentage of pregnant women who got a positive result on the pregnancy test. The *precision* describes the percentage of people with a positive test result who are actually pregnant.

precision recall pregnancy

Now, if you think that the pregnancy test in this image is not very good, you've understood what *recall* and *precision* mean.

You can learn more on this subject on the relevant Wikipedia page. In our case it is not about whether or not someone is pregnant, but about whether or not someone is in a room. But the principles of *recall* and *precision* are the same.

Decision makers often want a value that expresses the quality of the predictions. Since accuracy is not suitable, a combination of recall and precision has become established. This is called the *F1 score*. This value is the harmonic mean of *recall* and *precision*. The harmonic mean is a kind of average for relative frequencies. The formula looks like this:

\begin{equation*} F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \end{equation*}
And you can calculate the F1 value with Python like this:

In [16]:
```
F1 = 2 * (precision * recall)/(precision + recall)
F1
```

Out[16]:
```
0.6146592370338619
```

It is noticeable that the F1 score (61%) is lower than the accuracy (66%). This is a sign that the k-nearest neighbors model has at least partially exploited the imbalance in category sizes.

Although the model quality metrics *precision*, *recall* and F1 score can be derived from the confusion matrix, most data scientists use the corresponding functions from the `sklearn.metrics` module: `precision_score()`, `recall_score()`, `f1_score()`. Import them directly.

In [18]:
```
from sklearn.metrics import precision_score, recall_score, f1_score
```

All three need the true categories (`target_test`) and the predicted categories (`target_test_pred`). Use `recall_score()`, `precision_score()` and `f1_score()` to calculate the model quality metrics. Do the values match those we calculated earlier?

In [19]:
```
print(recall_score(target_test, target_test_pred))
print(precision_score(target_test, target_test_pred))
print(f1_score(target_test, target_test_pred) )
```

```
0.7376543209876543
0.526818515797208
0.6146592370338619
```

Congratulations: You have learned about four of the most important model quality metrics for classification models:

- Accuracy
- Recall
- Precision
- F1 score

You should remember these values when evaluating classification models. You don't necessarily need your own test data set for this. Cross-validation also allows you to create an independent validation data set from the training data. We'll look at that in the next lesson.

Remember:

- For unequal category sizes, the *accuracy* is not a good model quality metric.
- *Recall* and *precision* both focus on the cases that are actually positive.
- *Recall* and *precision* only differ in the population: either all cases that are actually positive (*recall*) or all positively predicted cases (*precision*)
- The F1 score is a robust metric that considers both false positives and false negatives

Literature:

- https://en.wikipedia.org/wiki/Confusion_matrix

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

The data was published here first: Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. Luis M. Candanedo, Véronique Feldheim. Energy and Buildings. Volume 112, 15 January 2016, Pages 28-39.