ROC AUC Area under the ROC Curve

Module 2 | Chapter 2 | Notebook 7

In this notebook we use the regression models created in the previous exercise and evaluate them with the known model quality metrics for classification models. However, you will learn about a new model quality metric based on the predicted probabilities. By this end this lesson you will have learned about:

- The False Positive Rate model quality metric
- The Receiver Operating Characteristics curve
- The Area under the ROC Curve model quality Metric (*ROC AUC*)

Model quality metrics for binary classification

Scenario: Pictaglam, a popular social media platform for sharing photos and videos, has received complaints about fake user accounts. Management at Pictaglam's have asked you to create a machine learning model that would help the platform to distinguish between real accounts and fake accounts.

In the previous exercise you created two logistic regression tests. Now you should determine which model is better suited to predicting whether user accounts are fake.

The file *social_media_train.csv* contains data on real and fake Pictaglam user accounts. Test data is provided in *social_media_test.csv*.

Let's import and prepare the training data quickly:

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

# data read in
df_train = pd.read_csv("social_media_train.csv", index_col=[0])

# label encoding
dict_label_encoding = {'Yes': 1, 'No': 0}
df_train.loc[:, 'profile_pic'] = df_train.loc[:, 'profile_pic'].replace(dict_label_enc
df_train.loc[:, 'extern_url'] = df_train.loc[:, 'extern_url'].replace(dict_label_encod
df_train.loc[:, 'private'] = df_train.loc[:, 'private'].replace(dict_label_encoding)

# one-hot encoding
onehot = pdp.OneHotEncode(["sim_name_username"], drop_first=False)
df_train = onehot.fit_transform(df_train) # fit and transform to training set
```

```
# Look at data
df_train.head()
```

```
Out[1]:
   fake  profile_pic  ratio_numlen_username  len_fullname  ratio_numlen_fullname  len_desc  extern_url
0     0            1                   0.27             0                    0.0        53           0
1     0            1                   0.00             2                    0.0        44           0
2     0            1                   0.10             2                    0.0         0           0
3     0            1                   0.00             1                    0.0        82           0
4     0            1                   0.00             2                    0.0         0           0
```

The code for that looks like this:

Column number	Column name	Type	Description
0	`'fake'`	categorical	Whether the user account is real (`0`) or fake (`1`).
1	`'profile_pic'`	categorical	Whether the account has a profile picture (`'Yes'`) or not (`'No'`)
2	`'ratio_numlen_username'`	continuous (`float`)	Ratio of numeric characters in the account username to its length
3	`'len_fullname'`	continuous (`int`)	total number of characters in the user's full name
4	`'ratio_numlen_fullname'`	continuous (`float`)	Ratio of numeric characters in the account username to its length
5	`'sim_name_username'`	categorical	Whether the user's name matches their username completely (`'Full match'`), partially (`'Partial match'`) or not at all (`'No match'`)
6	`'len_desc'`	continuous (`int`)	Number of characters in the account's description
7	`'extern_url'`	categorical	Whether the account description contains a URL (`'Yes'`) or not (`'No'`)
8	`'private'`	categorical	Whether the user's contributions are only visible to their followers (`'Yes'`) or to all Pictaglam users (`'No'`)
9	`'num_posts'`	continuous (`int`)	Number of posts by the account
10	`'num_followers'`	continuous (`int`)	Number of Pictaglam users who follow the account
11	`'num_following'`	continuous (`int`)	Number of Pictaglam users the account is following

Each row of `df` represents a user or user account.

We defined the following two logistic regression models in the last programming exercise:

- `model_log` without regularization
- `model_reg` with regularization

In the following code cell we create it again:

In [2]:
```python
# 1. Model specification
from sklearn.linear_model import LogisticRegression

###################################################################
# a) Without regularization

# 2a. Feature matrix and target vector
features_train = df_train.iloc[:, 1:]
target_train = df_train.loc[:, 'fake']

# 3a. Model instantiation
model_log = LogisticRegression(solver='lbfgs', max_iter=1e4, C=1e42, random_state=42)

# 4a. Model fitting
model_log.fit(features_train, target_train)

#####################################################################
# b) With regularization

# 2b. Feature matrix standardization
from sklearn.preprocessing import StandardScaler #use StandardScaler to adjust the fea

scaler = StandardScaler()
features_train_scaled = scaler.fit_transform(features_train) #fit to training data and

# 3b. Model instantiation
model_reg = LogisticRegression(solver='lbfgs', max_iter=1e4, C=0.5, random_state=42)

# 4b. Model fitting
model_reg.fit(features_train_scaled, target_train)
```

Out[2]:
```
LogisticRegression(C=0.5, max_iter=10000.0, random_state=42)
```

So which model is better? In the lesson *Evaluating Classification Performance (Module 1 Chapter 2)*, you learned about 4 model quality metrics for classification algorithms:

- Accuracy: `sklearn.metrics.accuracy_score()`
- Precision: `sklearn.metrics.precision_score()`
- Recall: `sklearn.metrics.recall_score()`
- F1 score: `sklearn.metrics.f1_score()`

You can derive all these model quality metrics from a confusion matrix (`sklearn.metrics.confusion_matrix()`):

Precision

confusion matrix precision

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

Recall

confusion matrix recall

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

In our case, the reference category (`1`) is the fake account category. So if a fake account (actually: `1`) is identified as such (predicted to be `1`), it really is a positive case (*hit* in the lower right corner of the confusion matrix). On the other hand, if a fake account (actually: `1`) is not identified as such (predicted to be `0`), it is a false negative case (*miss* in the lower left corner of the truth table).

To avoid *overfitting*, you shouldn't use the training data to evaluate the model. Either you separate validation data from training data (*Evaluating Models with Cross Validation (Module 1, Chapter 2)*) or you use test data. One way or another, you always evaluate a model with data that the model didn't have access to in the training phase.

In this case we'll use the test data from *social_media_test.csv*. Import this data as a `DataFrame` named `df_test`. Use column `0` for the row names again. Then prepare the test data with label encoding and one-hot encoding and print the first few rows.

In [3]:
```
# import data
df_test = pd.read_csv("social_media_test.csv", index_col=[0])

# label encoding
dict_label_encoding = {'Yes': 1, 'No': 0}
df_test.loc[:, 'profile_pic'] = df_test.loc[:, 'profile_pic'].replace(dict_label_encod
df_test.loc[:, 'extern_url'] = df_test.loc[:, 'extern_url'].replace(dict_label_encodir
df_test.loc[:, 'private'] = df_test.loc[:, 'private'].replace(dict_label_encoding)

# one-hot encoding
df_test = onehot.transform(df_test) #transform to test set

# look at data
df_test.head()
```

Out[3]:

	fake	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url
0	0	1	0.33	1	0.33	30	0
1	0	1	0.00	5	0.00	64	0
2	0	1	0.00	2	0.00	82	0
3	0	1	0.00	1	0.00	143	0
4	0	1	0.50	1	0.00	76	0

Now we have 120 test data points available to evaluate `model_log` and `model_reg` with independent data. Divide them into a feature matrix (`features_test`) and a target vector (`target_test`).

Then use `model_log` to calculate the categorical target values (`0` or `1`) for the test data and store these in `target_test_pred_log` . Then calculate the precision and recall model quality metrics, and print them afterwards.

In [4]:
```
# features matrix and target vector
features_test = df_test.iloc[:, 1:]
target_test = df_test.loc[:, 'fake']

# predict target values from model
target_test_pred_log = model_log.predict(features_test)

# model evaluation
from sklearn.metrics import precision_score, recall_score
precision_log = precision_score(target_test, target_test_pred_log)
recall_log = recall_score(target_test, target_test_pred_log)

# print
print('Precision of model without regularisation: ', precision_log)
print('Recall of model without regularisation: ', recall_log)
```

```
Precision of model without regularisation:  0.8666666666666667
Recall of model without regularisation:  0.8666666666666667
```

The model performs quite well. Both the precision and the recall are over 80%:

Model	*precision*	*recall*
`model_log`	86.7%	86.7%

However, we should always compare with other models to see if we can possibly get better metrics.

So how does it look for `model_reg` ? So calculate the precision and recall for this model, save them in `precision_reg` and `recall_reg` , and print them afterwards.

Tip: Don't forget to standardize the feature values of the test data with `scaler` first, just like the feature values in the training data. Use `my_scaler.transform()` .

```
In [11]: # features matrix and target vector
features_test = df_test.iloc[:, 1:]
target_test = df_test.loc[:, 'fake']

features_standarized_test = scaler.transform(features_test)

# predict target values from model
target_test_pred_reg = model_reg.predict(features_standarized_test)

# model evaluation
from sklearn.metrics import precision_score, recall_score
precision_reg = precision_score(target_test, target_test_pred_reg)
recall_reg = recall_score(target_test, target_test_pred_reg)

# print
print('Precision of model without regularisation: ', precision_reg)
print('Recall of model without regularisation: ', recall_reg)
```

```
Precision of model without regularisation:  0.8813559322033898
Recall of model without regularisation:  0.8666666666666667
```

This model is also not bad at all, but it has slightly different strengths and weaknesses:

Model	precision	recall
`model_log`	86.7%	86.7%
`model_reg`	88.1%	86.7%

`model_log` is equally good at identifying fake accounts as such (recall), while `model_reg` had more accounts that were actually fake among the accounts that were predicted to be fake (precision). In other words, the predictions from `model_log` are slightly less reliable than those from `model_reg`.

Therefore, it is best to consider at the beginning of each analysis which model quality metric is the decisive one. You should also find out in advance whether there are any minimum reference values for your approach that need to be achieved. In any case, the evaluation of whether a model's performance can be considered "good" or "bad" depends on the underlying aim of your analysis.

Congratulations: We have recapped the model quality metrics for binary classification and have seen that the models differ mainly in precision. The model quality metrics we have used so far are based on matching actual binary categories (`1` vs `0`) with predicted categories.

However, in the last lesson we learned that logistic regression can also be used to predict the probability of a category. This opens up the possibility of using a new model quality metric: the area under the ROC curve. Let's look at that next.

ROC Curve

In the last lesson we learned that `LogisticRegression` uses a threshold of `0.5` (50%) by default to determine the predicted categories. If this threshold is lowered, there are 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 cases predicted to be in the reference category turn out not to be.

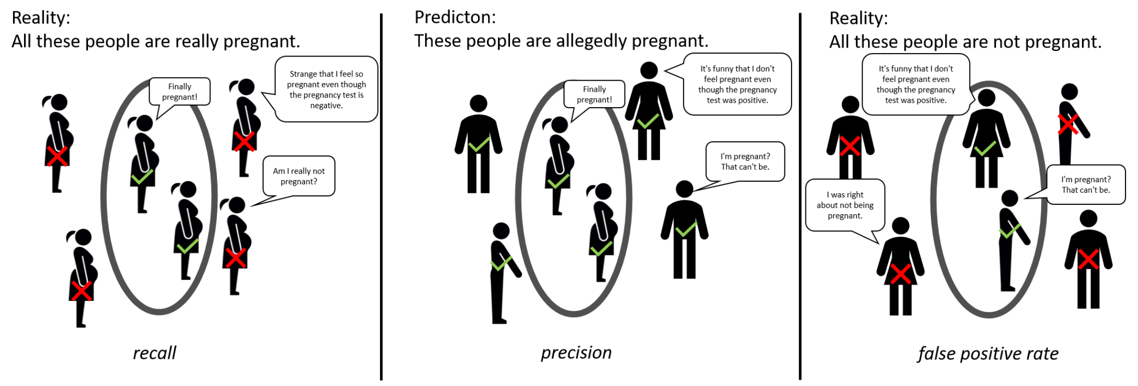
In our case, lowering the threshold would mean that more accounts would be classified as fake. This means you miss fewer fake accounts (recall), but at the same time there are a lot more real accounts among the predicted fake accounts (precision). An extreme case would be a threshold of `0`, where all accounts are predicted to be fake. This way all fake accounts are identified as such (perfect recall), but the precision is extremely poor.

You can visualize these dynamics well with a Receiver Operating Characteristic curve - also known as an ROC curve). The y-axis shows the recall. The x-axis shows a new metric: The false positive rate (FPR). This can also be derived from the truth matrix and is calculated as follows:

False positive rate

confusion matrix false positive rate

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


Reality:
All these people are really pregnant.
Finally pregnant!
Strange that I feel so pregnant even though the pregnancy test is negative.
Am I really not pregnant?
recall
Predicton:
These people are allegedly pregnant.
Finally pregnant!
It's funny that I don't feel pregnant even though the pregnancy test was positive.
I'm pregnant? That can't be.
precision
Reality:
All these people are not pregnant.
It's funny that I don't feel pregnant even though the pregnancy test was positive.
I was right about not being pregnant.
I'm pregnant? That can't be.
false positive rate

If you imagine all this for a pregnancy test, the pregnancy category is the reference category. You can understand the precision, recall and FPR like this:

pregnancy visualisation

- Recall: the percentage of pregnant women who correctly received a positive pregnancy test result
- Precision: the percentage of people with a positive test result who are actually pregnant
- False positive rate (FPR): the percentage of people who are not pregnant, who have received a positive pregnancy test result by mistake

What are the false positive rates for `model_log` and `model_reg`? They can be calculated by using the confusion matrix.

Import `confusion_matrix()` directly from `sklearn.metrics`. Then calculate the truth matrix for the model without regularization (`model_log`) by passing `target_test` and `target_test_pred_log` to `confusion_matrix`. Store the matrix in `cm_log`. The false positive rate is then the value in row 0 column 1 divided by the sum of all values in row 0. Print this value and then proceed in the same way for the model with regularization (`model_reg`).

In [16]:
```
# module import
from sklearn.metrics import confusion_matrix

# calculate confucion matrices
cm_log = confusion_matrix(target_test, target_test_pred_log)
cm_reg = confusion_matrix(target_test, target_test_pred_reg)

print('False positive rate of model without regularisation: ',
      round(cm_log[0, 1]/(cm_log[0, 1] + cm_log[0, 0]), 3))

print('False positive rate of model with regularisation: ',
      round(cm_reg[0, 1]/(cm_reg[0, 1] + cm_reg[0, 0]), 3))
```

```
False positive rate of model without regularisation:  0.133
False positive rate of model with regularisation:  0.117
```

`model_reg` has a slightly lower false positive rate:

Model	Precision	Recall	False positive rate
`model_log`	86.7%	86.7%	13,3%
`model_reg`	88.1%	86.7%	11.7%

Where can you find this value in an ROC curve?

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 predicted probabilities.

As an example, let's look at the ROC curve for the logistic regression model without regularization `model_log`. Since we're only using the probability of the reference category, we'll only use the right column of `target_test_pred_proba_log`.

In [20]:
```
# calculate probability
target_test_pred_proba_log = model_log.predict_proba(features_test)

# module import
from sklearn.metrics import roc_curve

# calculate roc curve values
false_positive_rate_log, recall_log, threshold_log = roc_curve(target_test,
                                                               target_test_pred_proba_
                                                               drop_intermediate=False
```

`roc_curve()` returns three vectors as arrays:

- the false positive rate (FPR) for each threshold
- the recall for each threshold
- the thresholds

This means, for example, that the values with index `49` of `false_positive_rate_log` and `recall_log` are the corresponding model quality metrics at the threshold value set in `threshold` with the same index. Print the value at index `49` of `threshold_log`, `recall_log` and `false_positive_rate_log`.

In [22]:
```
print("Threshold (model_log): ", threshold_log[49])
print("Recall (model_log): ", recall_log[49])
print("False positive rate (model_log): ", false_positive_rate_log[49])
```

```
Threshold (model_log):  0.8814056500610079
Recall (model_log):  0.7666666666666667
False positive rate (model_log):  0.06666666666666667
```

With a threshold value of about `0.88` the model is very strict. It only classifies a user account as fake if it's certain. As a result, the false-positive rate is extremely low (approx. 7%), i.e. hardly any accounts classified as fake are actually real accounts. At the same time, however, the recall is worse than before with the standard threshold value `0.5` (approx. 77%).

Let's try this again with the corresponding values at index `70`. Print the value at this index from `threshold_log`, `recall_log` and `false_positive_rate_log`.

In [23]:
```
print("Threshold (model_log): ", threshold_log[70])
print("Recall (model_log): ", recall_log[70])
print("False positive rate (model_log): ", false_positive_rate_log[70])
```

```
Threshold (model_log):  0.2636253460649329
Recall (model_log):  1.0
False positive rate (model_log):  0.18333333333333332
```

Now the model is no longer as strict (threshold value at `0.27`) and all fake accounts are identified as such (recall). At the same time, the proportion of accounts falsely predicted as fake (false positive rate) rises to 18%. But all fake accounts are identified (recall `1.0`)

In summary, this is the picture we get for `model_log`:

Threshold value	recall	false positive rate
`0.27`	100%	18.3%
`0.5` (standard)	86.7%	13.3%
`0.82`	76.7%	6.7%

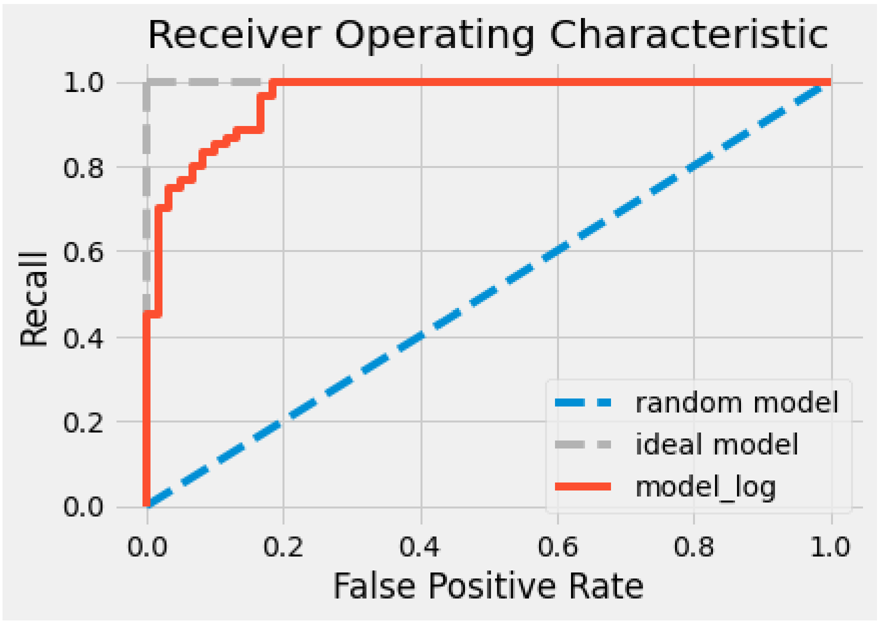
You can see that as the threshold value increases, both the hit rate and the false positive rate decrease. This means that if the model becomes stricter and increasingly only predicts fake accounts where it is very certain, more and more fake accounts will not be identified as such (recall) and the number of accounts incorrectly classified as fake will decrease (false positive rate).

Think about this dilemma from the perspective of the decision-makers at Pictaglam: If you want to delete fake accounts, you can either be strict and incorrectly leave a lot of fake accounts as they are, so that as few real customer accounts as possible are deleted by mistake. Alternatively, you want to delete as many fake accounts as possible, but then you have to live with the fact that many real customers will lose their accounts by mistake.

The ROC curve visualizes this dilemma:

```
In [28]: # module import and style setting
         import matplotlib.pyplot as plt
         plt.style.use('fivethirtyeight')

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

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

         # roc curve
         ax.plot(false_positive_rate_log, recall_log, label='model_log')

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

```
Out[28]: <matplotlib.legend.Legend at 0x7fc6abe51400>
```



Dotted you can see two reference curves. 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. 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. In our case, this would mean that with every fake account that is identified as fake, a real account would also be classified as such.

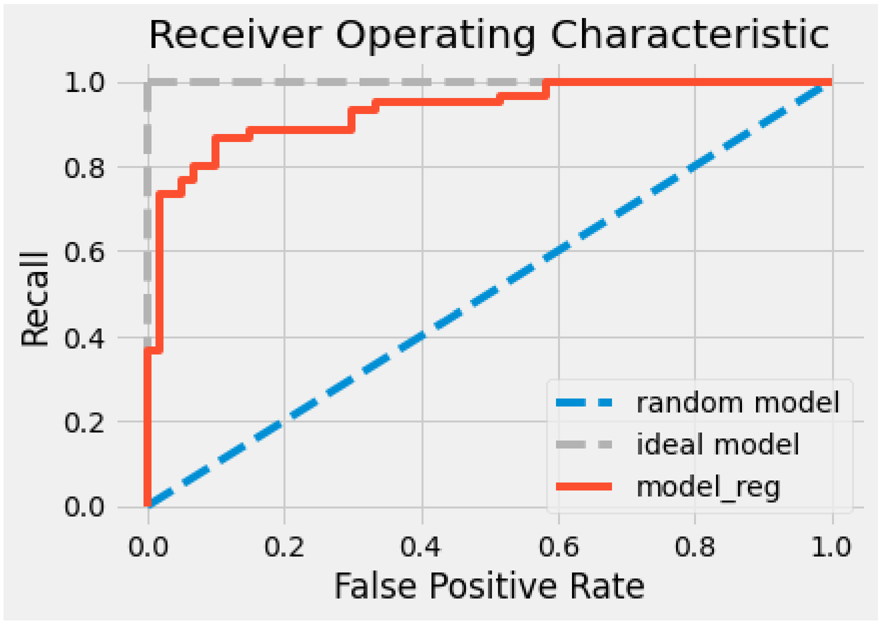
The red line belongs to our model. Up to a hit rate of 40% and from a false-positive rate of approx. 19%, it barely deviates from the ideal. So the model appears to be generally very good, as it follows the grey ideal rather than the blue random guess line. But it's not perfect.

Now visualize the ROC curve for `model_reg`.

First calculate the fake probabilities of the test data according to `model_reg` and `features_test_scaled` and save them in `target_test_pred_proba_reg`. Then calculate the values for the false positive rate and the recall at different thresholds using `roc_curve()`. Use the `drop_intermediate=False` parameter, otherwise not all limit values are generated. Then visualize these key figures with the false positive rate on the x-axis and the recall on the y-axis.

In [27]:
```
# calculate probability
target_test_pred_proba_reg = model_reg.predict_proba(features_test_scaled)

# calculate roc curve values
false_positive_rate_reg, recall_reg, threshold_reg = roc_curve(target_test,
                                                               target_test_pred_proba_
                                                               drop_intermediate=False

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

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

# roc curve
ax.plot(false_positive_rate_reg, recall_reg, label='model_reg')

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

Out[27]:
```
<matplotlib.legend.Legend at 0x7fc6abed1730>
```



You should get the following ROC curve:

roc curve

The ROC curve of the model with regularization (`model_reg`) deviates slightly more from the ideal of the grey dashed line than the model without regularization (`model_log`). This suggests that the logistic regression model with regularization has a slightly worse hit rate than the non-regularized model.

Congratulations: You have learned about the ROC curve. It visualizes a dilemma a lot of decision makers face. As the classification becomes more and more strict, you become more certain that the accounts predicted to be fake are actually fake, while at the same time overlooking a lot of accounts that are actually fake. Next we'll look at how to get a model quality metric from the visual impression of the ROC curves.

ROC AUC metric

To create a quantifiable model quality metric from the visual impression we have just used, we calculate the area under the curve. This is why the model quality metric is called: *receiver operator characteristic area under the curve (ROC AUC)*. It's also referred to as AUROC.

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` . `import` die function directly.

In [29]:
```
from sklearn.metrics import roc_auc_score
```

The `roc_auc_score()` function needs the actual categories (`target_test`) and the predicted reference category probabilities for them (`target_test_pred_proba_log[:,1]`).

```
In [30]: roc_auc_score(target_test, target_test_pred_proba_log[:, 1])
```

```
Out[30]: 0.9624999999999999
```

A value of `0.96` is very good and confirms our visual impression from above. The model is very close to the ideal. We can also interpret the value directly.

Let's assume that from all the Pictaglam accounts that exist, we take one fake account and one real account. If the model has to choose which of the two is probably fake, then in 96% of the cases it will correctly select the fake account.

Also calculate the *ROC AUC* value for `model_reg`.

```
In [31]: roc_auc_score(target_test, target_test_pred_proba_reg[:,1])
```

```
Out[31]: 0.9322222222222222
```

The value is slightly lower than for `model_log`, at around 93%. This again confirms our visual impression above: The model with regularization follows the ideal less than the model without regularization.

So in this case, regularization has reduced rather than improved the quality of the model. Regardless of the decision thresholds we choose, we can conclude that we have overfitted our model!

So you should always try to find the balance between the accuracy and the possibility to generalize a model.

Congratulations: You have recapped the model quality metrics for classification models you learned in the last module and got to know a new one: *ROC AUC*. This metric differs from the ones you learned previously because it's independent of how strict your classification is. It can thus be used to evaluate the potential of the model to implement strict and non-strict thresholds.

Remember:

- The confusion matrix (`sklearn.metrics.confusion_matrix()`) and all model quality metrics derived from it are based on predicted categories
- You can calculate false positive and false negative rates from the confusion matrix
- The ROC curve and the *ROC AUC* metric are based on the predicted probabilities
- `sklearn` function of the model quality metric *ROC AUC*: `sklearn.metrics.roc_auc_score()`

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.
