Finding the Best Logistic Regression Model with Grid Search and ROC-AUC

Module 2 | Chapter 2 | Notebook 8

In this notebook, we'll combine the knowledge you've gained in this chapter. We'll use the *ROC-AUC* metric to find the best logistic regression model and then use it to predict the fake status of Pictaglam accounts.

Preparing data

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. The company would then use your model to identify fake Pictaglam accounts so that they can then be deleted from the platform.

You will be asked to find the best model for the task, train it, evaluate it and then predict the fake status of Pictaglam accounts.

The file *social_media_train.csv* contains data on real and fake Pictaglam user accounts. Test data is provided in *social_media_test.csv*. You've been asked to classify the Pictaglam accounts in *social_media_aim.csv*.

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'`),

Column number	Column name	Type	Description
			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.

Import the training, test and target data and save each set in `df_train`, `df_test` and `df_aim` respectively. Use label encoding on the binary categorical columns (`'profile_pic'`, `'extern_url'` , `'private'`). Use one-hot encoding on the categorical columns with more than two categories (`'sim_name_username'`), see *Preparing Categorical Features*.

Tip: The data preparation process is identical for the 3 different files `['social_media_train.csv', 'social_media_test.csv', 'social_media_aim.csv']`. We recommend using a loop and saving the 3 DataFrames in a list first. Then you can give the DataFrame the normal names from the list and adjust the one-hot encoder from `pdpipe` to the training set and transform all the data sets with it.

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

# label encoding dictionary
dict_label_encoding = {'Yes': 1, 'No': 0}

# will save DataFrames
df_list = []

for df_str in ['social_media_train.csv',
               'social_media_test.csv',
               'social_media_aim.csv']:

    # data read in
    df = pd.read_csv(df_str, index_col=[0])

    # label encoding
    df.loc[:, 'profile_pic'] = df.loc[:, 'profile_pic'].replace(dict_label_encoding)
    df.loc[:, 'extern_url'] = df.loc[:, 'extern_url'].replace(dict_label_encoding)
    df.loc[:, 'private'] = df.loc[:, 'private'].replace(dict_label_encoding)

    # append to list
    df_list.append(df)


# creating data sets
df_train = df_list[0]
df_test = df_list[1]
df_aim = df_list[2]

# one-hot encoding
onehot = pdp.OneHotEncode(["sim_name_username"], drop_first=False)
df_train = onehot.fit_transform(df_train) # only ever fit to training set!
df_test = onehot.transform(df_test)
df_aim = onehot.transform(df_aim)

# 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

Congratulations: You imported and cleaned the data. Then we can now turn use a grid search to find the best logistic regression model.

Grid search

So far we've evaluated two models to find the better one. However, in the lesson *Grid Search (Module 1, Chapter 2)* you learned that Python can do that for us as well. Using a grid search automatically finds the optimal hyperparameters.

But for that we need a few things, like a scaler and a `Pipeline`.

Let's prepare the grid search by importing the most important things:

- `LogisticRegression` from `sklearn.linear_model`
- `StandardScaler` from `sklearn.preprocessing`
- `Pipeline` from `sklearn.pipeline`
- `GridSearchCV` from `sklearn.model_selection`

In [2]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
```

Next, split the training data into the feature matrix (all the columns of `df_train` except `'fake'`) and the target vector (the `'fake'` column of `'f_train'`). Store these in the variables `features_train` and `target_train`.

In [3]:
```
features_train = df_train.iloc[:,1:]
target_train =  df_train.iloc[:,0]
```

Then we can now turn to the `Pipeline`, which contains the steps from the data to the forecast, see *Cross Validation with Pipelines (Module 1, Chapter 2)*. Since we then want to try out different regularization parameters in the grid search, we should standardize the data so it's on a uniform scale. Use `StandardScaler` to do this.

Now define a `Pipeline` called `pipeline_log` with two steps:

1. A standardization step called `scaler`, which consists of `StandardScaler()`
2. A modelling step called `'classifier'`, which consists of `'LogisticRegression(solver='saga', max_iter=1e4)`. We should also set `random_state=42` so that the results are reproducible.

Important: Not all `solver` settings support all options. Some also take longer or need more memory. You can find more on the documentation page.

Since we want to try out the regularization of both ridge regression and the LASSO regression in this notebook, we can't use `'lbfgs'` as before. `'saga'` is the only setting that supports both types of regularization.

For the algorithm to find a good result, it has to make quite a lot of attempts. So you should increase `max_iter` from the standard 100 iterations. 10000 iterations (`1e4`) should be enough.

In [5]:
```
pipeline_log = Pipeline([('scaler', StandardScaler()),
                         ('classifier', LogisticRegression(solver='saga',
                                                           max_iter=1e4,
                                                           random_state=42))])
```

Before we can start the grid search, we have to define the grid of hyperparameters that it will search for. If you look at [the documentation of LogisticRegression, you'll see some hyper parameters that you could try:

- `penalty`: whether the regularization proceeds like with Ridge (`'l2'`) or like with LASSO (`'l1'`)
- `C`: Regularization weakness, see *Logistical Regression with and without Regularization*

Which parameter values do we want to look for under `C`? For the difference between the `C` parameter values tried out to be meaningful, it shouldn't be linear. For example, if you try `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`, then you test a very large difference between maximum regularization (`C=0`) and standard regularization (`C=1`) at the beginning, but then at the top end you test insignificantly small differences between relatively little regularization (`C=9`) and also relatively little regularization (`C=10`).

Therefore, it's better to include increasingly large distances between the parameter values you are trying out. In the lesson *Grid Search (Module 1 Chapter 2)*, you learned about the `numpy` function `np.geomspace()` . It creates a series of numbers with increasingly large distances between the numbers from a start value (the `start` parameter) to a target value (the `stop` parameter). The number of numbers generated is specified by the `num` parameter.

Use `np.geomspace()` to generate `14` values from `0.001` to `1000` with increasingly large distances between the values. Name the resulting one-dimensional array `C_values` .

In [18]:
```
import numpy as np
np.set_printoptions(suppress=True)  # avoid scientific notation

C_values = np.geomspace(start=0.001, stop=1000, num=14)

print(C_values)
```

```
[   0.001         0.00289427    0.00837678    0.02424462    0.07017038
    0.20309176    0.58780161    1.70125428    4.92388263   14.2510267
   41.24626383  119.37766417  345.51072946 1000.        ]
```

Together with the two arguments for the `penalty` parameter, the grid of hyperparameter values that are tried out would look like this

grid search grid

Define a `search_space_grid` variable, which we can use later to tell `GridSearchCV` to try out these hyperparameter values.

Tip: `search_space_grid` should output a list with just one entry. This entry is a `dict` with two keys. The first key should indicate the `penalty` parameter of the `'classifier'` step of `pipeline_log` (`'classifier__penalty'`). The key's value, i.e. the values to be tried out, is the list `['l1', 'l2']` . The second key should refer to the `C` parameter of the `classifier` step of `model` and the value should contain the search space of the regularization parameter (`C_values`) predefined here.

In [9]:
```
search_space_grid = [{'classifier__penalty': ['l1', 'l2'],
                      'classifier__C': C_values}]
```

Now we can instantiate the grid search: Name the resulting variable `model_grid` . The grid search should select the best model according to the *ROC-AUC* metric. Assign the `scoring` parameter of `GridSearchCV()` to `'roc_auc'` . Specify a 5-fold cross validation (`cv` parameter of `GridSearchCV()`). Also, use `n_jobs=-1` to access all CPU cores to speed up the calculation.

In [19]:
```
model_grid = GridSearchCV(estimator=pipeline_log,
                          param_grid=search_space_grid,
                                   scoring='roc_auc',
                                   cv=5,
                                   n_jobs=-1)
```

Before we do the grid search, we'll deactivate the `DataConversionWarning`, which would otherwise occur very often although you can safely ignore it.

In [11]:
```
from sklearn.exceptions import DataConversionWarning
import warnings

warnings.filterwarnings(action='ignore', category=DataConversionWarning)
```

Now run the following cell. Which hyperparameter settings produces the largest area under the ROC curve? How large is this area? Print the `my_GridSearchCV.best_estimator_` attribute and the `my_GridSearchCV.best_score_` attribute after the grid search to answer these questions.

In [20]:
```
model_grid.fit(features_train, target_train)

print(model_grid.best_estimator_)
print(model_grid.best_score_)
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('classifier',
                 LogisticRegression(C=4.923882631706742, max_iter=10000.0,
                                    random_state=42, solver='saga'))])
0.9657175042242944
```

You should get a *ROC AUC* value of about 96.6%. This is an excellent value.

The logistic regression model that produced this value uses a regularization similar to the ridge regression (`penalty='l2'`) and regularizes slightly less than the standard model (`C=4.92`).

Congratulations: You have found the best logistic regression model by using a grid search. You could of course extend the grid search with polynomials or principal component analysis, for example. However, we'll not do this now as it would take too long, and we'll move on to the next step. Next, we'll evaluate the best model with the test data.

Evaluating the model with test data

It is best to check whether the extraordinarily good model quality you get from cross validation can also be achieved with the test data. Store the feature matrix and target vector of the test data in new variables (`features_test`, `target_test`) and use the best model from the grid search to predict probabilities of the different category assignments. Assign this to the variable `target_test_pred_proba`.

Tip: At the end of the grid search, the best model is automatically fitted to the data and then returned. So you don't need to instantiate the best model again. If you use `model_grid` to make predictions, it automatically uses the best model.

In [22]:
```
features_test = df_test.iloc[:, 1:]
target_test = df_test.iloc[:, 0]

target_test_pred_proba = model_grid.predict_proba(features_test)
```

With the predicted probabilities we can now determine the *ROC-AUC* value for this model and the test data. First import `roc_auc_score()` directly from `sklearn.metrics`. Then calculate the model quality metric. Is it as good as it was for the validation data during cross validation?

In [24]:
```
from sklearn.metrics import roc_auc_score
roc_auc_score(target_test, target_test_pred_proba[:, 1])
```

Out[24]:
```
0.9391666666666667
```

This value (*ROC-AUC measure* of 93.9%) is very good, but unfortunately slightly lower than the model quality measure calculated from the validation data during the grid search. There is often a discrepancy between the model performance with test data and validation data. Here are three common reasons for this:

1. The training data is guaranteed to be representative of the validation data because it comes from the same data set. However, the is not necessarily the case with the test data. The trained model therefore matches the validation data better (which is similar to the training data) than the test data (which is not necessarily as similar to the training data). This makes the model look better when evaluating with validation data than when evaluating with test data.
2. A grid search optimizes the hyperparameter settings for the training data. In extreme cases, this can result in the hyperparameter settings being overfitted to the training data. This problem is usually not as pronounced as the overfitting of the model coefficients to the training data during model fitting (see *The Overfitting Problem (Module 1, Chapter 1)*). However, this can still result in slightly lower model performance with test data than with validation data.
3. There is typically only one test data set and this is often relatively small. This could be a better or worse match to the trained model by coincidence. The calculated model quality can therefore go up or down more easily than with the validation data. For the validation data, the model quality measures are calculated at each step of the cross-validation and then averaged. This compensates for random upward or downward deviations.

Congratulations: The best model according to the grid search also performs well with the test data. The difference in performance with the validation data is not so big that we should be worried. So we can now move on to making the predictions.

Predictions

Use the target data in `df_aim` to create `features_aim`. Then calculate the fake probabilities of the user accounts in it. Store them in the new `df_aim` column `'fake_pred_proba'`. Also calculate the predicted fake category (`1` or `0`) of each Pictaglam account. Store this value in `max_whisk` . Then print these.

Tip: Create the features before you add the predictions as new columns. Otherwise, you could end up defining the predictions as features.

In [25]:
```
features_aim = df_aim.copy()
df_aim.loc[:, 'fake_pred_proba'] = model_grid.predict_proba(features_aim)[:, 1]
df_aim.loc[:, 'fake_pred'] = model_grid.predict(features_aim)

# avoid scientific notation
pd.set_option('display.float_format', lambda x: '%.3f' % x)
df_aim
```

Out[25]:

	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url	priv
0	1	0.330	1	0.330	30	0	
1	1	0.220	2	0.000	63	0	
2	0	0.000	2	0.000	0	0	
3	1	0.330	1	0.000	0	0	
4	1	0.000	1	0.000	137	1	
5	1	0.270	1	0.000	0	0	
6	0	0.440	1	0.440	112	0	

There are two Pictaglam accounts where the model is absolutely sure about the classification. The predicted fake probability is either exactly `1.0` or `0.0` .

The model is less certain for the other accounts. Management can decide for for themselves which threshold value they want to use to block accounts here. The classification with the default threshold value `0.5` is shown in the `'fake_pred'` column. This is a good starting point.

Congratulations: You have helped Pictaglam categorize user accounts into real and fake based on their features. You've even exceeded management's expectations because now they can decide for themselves how certain the classification needs to be before an account is deleted. Pictaglam's new management is very grateful and hopes you enjoy the rest of this course.

Remember:

- You can output the best model and the corresponding best metric for a grid search with the `sklearn` attributes `my_model.best_estimator_` and `my_model.best_score_` (always ending with an underscore).
- With `np.geomspace()` you can generate a series of numbers with increasing distances between the numbers

- There can be several reasons why model performance with validation data differs slightly from model performance with test data.

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.
