Logistic Regression with and without Regularization

Module 2 | Chapter 2 | Notebook 6

In this notebook we'll use the features we prepared in the previous lessons and train a logistic regression model with them. As with linear regression and its ridge and Lasso versions, logistic regression also allows for regularization. In this notebook you'll see what you need to consider when using regularization. By the end of this lesson you will have learned about:

- The `C` parameter of `LogisticRegression()` for specifying regularization
- `my_model.predict_proba()` to predict probabilities.

Logistic regression without regularization

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 file *social_media_train.csv* contains data on real and fake Pictaglam user accounts. You have 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'`), partially (`'Partial match'`) or not at all (`'No match'`)

Column number	Column name	Type	Description
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.

In the previous lessons, you learned how to prepare the categorical features. Use this knowledge in the next code cell. Carry out the following steps:

- Import `pandas` and `pdpipe`.
- Read the data from *social_media_train.csv* Store the data in a `DataFrame` called `df_train` and use the first column for row names. See *Logistic Regression*.
- Encode the labels of the columns `['profile_pic']`, `'extern_url'`, `'private']`. Represent a `'Yes'` with `1` and a `'No'` with `0`, see *Preparing Categorical Features*. Replace the existing columns rather than creating new ones. This makes it easier for you to select features to train your model on.
- Use one-hot encoding for the `'sim_name_username'` column, see *Prepare categorical features*.
- Print the first five rows of `df_train` to view the data.

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_encoc
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

Now we can set up a logistic regression model, see *Linear Regression versus Logistic Regression*. By default, the logistic regression algorithm of `sklearn` already uses regularization - with the regularization parameter `C` = 1.0 by default. If we assign an extremely large value to `C`, such as a 1 followed by 42 zeros (`1e42`), it doesn't perform any regularization. That is what we want to achieve here first.

Unfortunately the algorithm needs a lot of attempts to solve the problem. The standard 100 iterations are not enough. Therefore we should also assign a relatively large number to `max_iter`. This parameter sets the maximum number of iterations that the `solvers` need for convergence. 10000 (`1e4`) should be enough here.

Attention: The logistic regression model without regularization doesn't need you to scale the features, because without regularization they have no influence on the results of the logistic regression. So in this case we don't have to transform the features.

Carry out the following steps:

1. Import `LogisticRegression` from `sklearn.linear_model`
2. Instantiate the model and save it in the variable `model_log`. Use the following hyper parameter settings: `solver='lbfgs'`, `C=1e42` and `max_iter=1e4`. To be able to reproduce the model, you should also specify `random_state=42`.
3. Split the data into feature matrix (`features_train`) and target vector (`target_train`). Note that you should use all the numerical features. The target vector is the `'fake'` column of `df_train`.
4. Fit the model to the data.

In [2]:

```
# 1. Model specification
from sklearn.linear_model import LogisticRegression

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

# 3. Features matrix and target vector
features_train = df_train.iloc[:, 1:]
target_train = df_train.loc[:, 'fake']
```

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

Out[2]:
```
LogisticRegression(C=1e+42, max_iter=10000.0, random_state=42)
```

Now we've fitted the model to the data. But how do we know now that we've not overfitted the model to the data? It's best to be sure and and check this by performing a 5-fold cross-validation. To do this, apply `cross_val_score` and calculate the average `accuracy`.

In [5]:
```
from sklearn.model_selection import cross_val_score
cv_results = cross_val_score(estimator=model_log,
                             X=features_train,
                             y=target_train,
                             cv=5,
                             scoring='accuracy')
cv_results.mean()
```

Out[5]:
```
0.9252923538230885
```

We get an accuracy score of about 93%, which is pretty good.

We can now import *social_media_aim.csv* to make predictions with the trained model. Follow a similar process as you did above with *social_media_train.csv*:

- Import the data from *social_media_aim.csv* Store the data in a `DataFrame` called `df_aim` and use the first column for row names.
- Encode the labels of the `'profile_pic'`, `'extern_url'` und `'private'` columns. Represent a `'Yes'` with `1` and a `'No'` with `0`. Replace the existing columns rather than creating new ones. This makes it easier for you to select features to train your model on.
- Use one-hot encoding for the `'sim_name_username'` column.
- Print the first five rows of `df_aim` to look at the data.

In [8]:
```
# read in data
df_aim = pd.read_csv("social_media_aim.csv", index_col=[0])

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

# one-hot encoding
df_aim = onehot.transform(df_aim) #transform to target set

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

Out[8]:

	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url	priva
0	1	0.33	1	0.33	30	0	
1	1	0.22	2	0.00	63	0	
2	0	0.00	2	0.00	0	0	
3	1	0.33	1	0.00	0	0	
4	1	0.00	1	0.00	137	1	

`df_aim` does not contain a `'fake'` column, which indicates whether the respective user account is fake or not. It's best to use `model_log` to predict this classification for each account.

The following code first stores the feature matrix of the target data (`features_aim`). This will then allow `model_log` to predict target categories. These are stored directly in the `fake_pred_log` column.

In [9]:
```
features_aim = df_aim.copy()
df_aim.loc[:, 'fake_pred_log'] = model_log.predict(features_aim)
df_aim
```

Out[9]:

	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url	priva
0	1	0.33	1	0.33	30	0	
1	1	0.22	2	0.00	63	0	
2	0	0.00	2	0.00	0	0	
3	1	0.33	1	0.00	0	0	
4	1	0.00	1	0.00	137	1	
5	1	0.27	1	0.00	0	0	
6	0	0.44	1	0.44	112	0	

Of the seven accounts we needed a prediction for, five were classified as fake accounts, namely the accounts with row indexes 0, 2, 3, 5 and 6.

Congratulations: You have used a logistic regression model without regularization to predict whether a Pictaglam account is fake or not.

In the lesson *Regularization (Module 1, Chapter 1)* you learned that it can be helpful if a linear regression model tries to minimize the slope values. This can help to minimize overfitting, for example. Does this kind of approach lead to different predictions? Let's look at that next.

Logistic regression with regularization

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 would be `punished` if 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.

Let's start by instantiating the model. The `C` parameter of `LogisticRegression()` controls the balance between the top two targets, similar to the `alpha` parameter of `Ridge()`. Although they have the same function - confusingly, 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`. At the end you have to find the optimal value for `C` again with a grid search. In this case we'll try `0.5`. Instantiate `LogisticRegression` and store the model in the variable `model_reg`. Use the following hyperparameter settings: `solver='lbfgs'`, `C=0.5` and `max_iter=1e4`. To be able to reproduce this model, you should also specify `random_state=42`.

In [10]:
```
model_reg = LogisticRegression(solver='lbfgs', C=0.5, max_iter=1e4,random_state=42)
```

Attention: In contrast to the logistic regression without regularization, the features of the model with regularization definitely have to be standardized! This is due to the `penalty` parameter (`l2` by default) in logistic regression: as with linear regression, logistic regression with regularization makes the prediction dependent on feature scaling, with $L1$ (lasso) and

$L2$ (ridge) penalizing large coefficients more heavily. In order for the coefficients to be penalized equally, we have to standardize them.

You should always consider which scaling method is better for your data set(s). However, for logistic regression with regularization, the Z-transformation (`StandardScaler`) is recommended for the majority of cases, see also the references at the end of this lesson.

Now we have to standardize the features for the regularization. In our case, the best approach is a Z-transformation (`StandardScaler`) of our features for the two datasets `social_media_train.csv` and `social_media_train.csv` . Do this in the following cell for the features of the training set as well as the target data set.

In [11]:
```
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 a
features_aim_scaled = scaler.transform(features_aim)  #scale target data
```

`features_train_scaled` is now an array. So that we know exactly what each feature means, we'll quickly put it backt into a `DataFrame` .

In [12]:
```
features_train_scaled = pd.DataFrame(features_train_scaled, columns=features_train.col
```

Now use `features_train_scaled` and `target_train` to fit the model `model_reg` to the data.

In [13]:
```
model_reg.fit(features_train_scaled,target_train)
```

Out[13]:
```
LogisticRegression(C=0.5, max_iter=10000.0, random_state=42)
```

Use `features_aim_scaled` to predict which user account is probably fake. Store the predictions in the new `df_aim` column `'fake_pred_reg'` . Are they the same accounts as before with `model_log` ? Then print `df_aim` .

In [16]:
```
df_aim.loc[:,'fake_pred_reg'] =  model_reg.predict(features_aim_scaled)
df_aim
```

Out[16]:

	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url	priva
0	1	0.33	1	0.33	30	0	
1	1	0.22	2	0.00	63	0	
2	0	0.00	2	0.00	0	0	
3	1	0.33	1	0.00	0	0	
4	1	0.00	1	0.00	137	1	
5	1	0.27	1	0.00	0	0	
6	0	0.44	1	0.44	112	0	

The regularized logistic regression model predicts that exactly the same accounts will be fake as the non-regularized logistic regression model. However, since we haven't yet evaluated the models, we don't know which one we should trust more.

Congratulations: You have learned how to use regularization for a logistic regression model. It's important here again to standardize the feature data to the same scale. In most cases `StandardScaler` is suitable for this purpose.

The logistic regression model with regularization provided the same predictions as the model without regularization. However, the predictions might not be as similar if you look at the probability instead of the binary prediction (fake or not). This "fake probability" is what we're going to investigate next.

Predicted probabilities

If you train a classification model, you always predict the categories with `my_model.predict()`. But if you want to know the probability of something belonging to one category or another, you should use `my_model.predict_proba()`. It's probably best to try that out now.

In [17]:
```
target_aim_pred_proba = model_log.predict_proba(features_aim)
target_aim_pred_proba
```

Out[17]:
```
array([[2.08856615e-003, 9.97911434e-001],
       [8.45854099e-001, 1.54145901e-001],
       [2.97334427e-003, 9.97026656e-001],
       [2.68534687e-001, 7.31465313e-001],
       [1.00000000e+000, 9.35292218e-208],
       [1.82285494e-001, 8.17714506e-001],
       [3.59204692e-007, 9.99999641e-001]])
```

Due to the scientific notation, it's not immediately obvious how to interpret this. This is why we prefer to use the standard notation.

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

# suppress scientific notation in pandas
pd.options.display.float_format = '{:.2f}'.format

target_aim_pred_proba  # print array
```

Out[18]:
```
array([[0.00208857, 0.99791143],
       [0.8458541 , 0.1541459 ],
       [0.00297334, 0.99702666],
       [0.26853469, 0.73146531],
       [1.        , 0.        ],
       [0.18228549, 0.81771451],
       [0.00000036, 0.99999964]])
```

The numpy array (`ndarray`) is made up of several lists. Each list contains two values that add up to 1.0, which represents the overall probability of 100%. The left value in each list reflects the probability of the account not being fake. The right-hand value, on the other hand, reflects the probability of the account being classified as a fake account.

The second column of `target_aim_pred_proba` represents the "fake probability" of a user account, and the first column represents the opposite.

Here's a reminder of what management asked you to do: Print the contents of the `fake_pred_log' column of` df_aim`.

In [19]:
```
df_aim.loc[:, 'fake_pred_log']
```

Out[19]:
```
0    1
1    0
2    1
3    1
4    0
5    1
6    1
Name: fake_pred_log, dtype: int64
```

Apparently, high probability values in the second column of `target_aim_pred_proba` were converted to `1` values in the `'fake_pred_log'` column. The opposite is true for the `0` values in the `'fake_pred_log'` column.

Add the second column of `target_aim_pred_proba` to `df_aim` as a new column `'fake_pred_log_proba'`. Then print `df_aim` .

In [20]:
```
df_aim.loc[:, 'fake_pred_log_proba'] = target_aim_pred_proba[:, 1]
df_aim
```

Out[20]:

	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url	priva
0	1	0.33	1	0.33	30	0	
1	1	0.22	2	0.00	63	0	
2	0	0.00	2	0.00	0	0	
3	1	0.33	1	0.00	0	0	
4	1	0.00	1	0.00	137	1	
5	1	0.27	1	0.00	0	0	
6	0	0.44	1	0.44	112	0	

If the logistic regression models, with and without regularization (see `'fake_pred_log'` and `'fake_pred_reg'`), consistently predict that an account is fake, the probability of an account being fake is particularly high according to `model_log`. This indicates that you can use the stated probability as a measure of the uncertainty of the binary classification.

Now also calculate the predicted probabilities of accounts being fake using `model_reg` and add them to `df_aim` as a new column `'fake_pred_reg_proba'`. Then print `df_aim`.

In [21]:
```
df_aim.loc[:, 'fake_pred_reg_proba'] = model_reg.predict_proba(features_aim_scaled)[:,
df_aim
```

Out[21]:

	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	extern_url	priva
0	1	0.33	1	0.33	30	0	
1	1	0.22	2	0.00	63	0	
2	0	0.00	2	0.00	0	0	
3	1	0.33	1	0.00	0	0	
4	1	0.00	1	0.00	137	1	
5	1	0.27	1	0.00	0	0	
6	0	0.44	1	0.44	112	0	

If you look at the predicted probabilities of an account being fake with `model_reg` and `model_log`, you'll notice that both logistic regression models predict almost the same probabilities of accounts being fake. The only time the two models arrive at quite different predicted probabilities of an account being fake is with the account at row index 5: 81% for the logistic regression model without regularization and 65% for the model with regularization. Nevertheless, they are both certain that this is a fake account.

By default, `LogisticRegression` uses `0.5` as the threshold value. For example, in our case if the probability of an account being fake is less than `0.5` (50%), it is predicted that the account

is not fake. If it is higher, it is predicted to be fake.

However, it's worth questioning whether this threshold is optimal for every case. A lower threshold leads to more predicted reference category classifications. For example, with a pregnancy test, more people would think they were pregnant. The advantage is that women who are actually pregnant are then identified correctly. On the other hand, a lot of women who are not pregnant would then wonder why the non-pregnant people wonder why the test said that they were pregnant.

A higher threshold leads to fewer data points being classified as the reference category. This is good if you only want to classify things in the reference category if the model is really certain. For example, a good criminal justice system would use a high threshold because only people who are undoubtably criminals should be classified as criminals.

Congratulations: You have learned how to use regularization for a logistic regression model. You can now also interpret the predicted probabilities instead of just looking at the predicted categories.

Which prediction should we trust more now? It's time to evaluate the models. Next, we use these probabilities to evaluate logistic regression models with a new model quality metric.

Remember:

- `LogisticRegression` uses regularization as standard.
- While it's not necessary to standardize features for the logistic regression model without regularization, it is always necessary for a logistic regression model with regularization! A Z-transformation of the features using `StandardScaler` is suitable for most cases.
- The higher the `C` parameter of `LogisticRegression()`, the weaker the regularization.
- With `my_model.predict_proba()` predict probabilities.

Literature: If you would like to delve deeper into the subject matter of this chapter, we recommend the following source(s):

- Hastie T., Tibshirani R., Friedman J. (2009) Linear Methods for Classification. In: The Elements of Statistical Learning. Springer Series in Statistics. Springer, New York, NY. pp. 101-137.

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.
