And implementation with Scikit-learn

towardsdatascience.com/logistic-regression-ca2d070a3eee

August 19, 2019

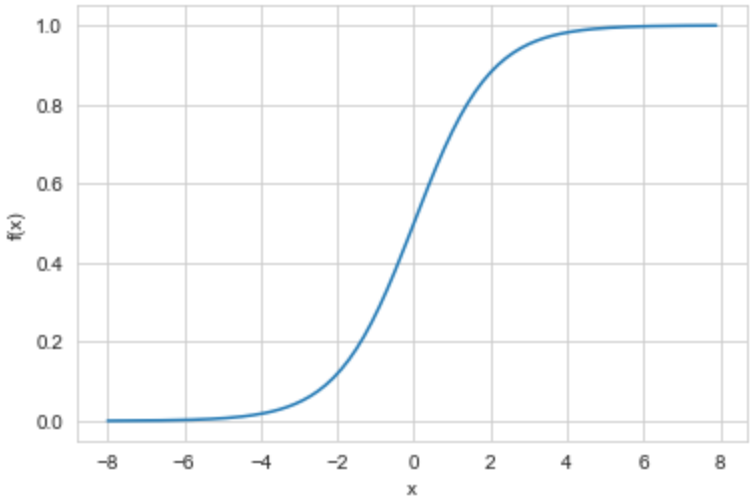
Samantha Jackson

Logistic Regression

Logistic Regression

Theory

A member of the generalized linear model (GLM) family and similar to linear regression in many ways, logistic regression (despite the confusing name) is used for classification problems with two possible outcomes.



Sigmoid function: 1/(1+exp(-x))

Logistic regression is handy for classification problems since it fits an S shaped logistic (or Sigmoid) function to the data, squishing the linear equation to an output range of 0–1. This convenient range allows logistic regression to model the probabilities of a data point belonging to a particular class, typically with the decision point at the probability of .5.

So, what does that look like in math? How does the sigmoid function squish the linear equation? As you may know, modeling the relationship between features and target variable for a linear regression looks like this:

$$\hat{y}^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_p x_p^{(i)}$$

Logistic regression simply takes that linear equation, and uses it as the parameter for the sigmoid function to come up with the probability of the data point belonging to a particular class (in the case below, class 1):

$$P(y^{(i)} = 1) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_p x_p^{(i)}))}$$

Interpreting our models

Since we've squished our linear model into a probability range, the coefficients (or weights) of each feature no longer affect the output in an additive way. So, to interpret our models, we can re-stretch our sigmoid function from output range 0–1, back to +/- infinity by converting our probabilities to log(odds). We can do this with the logit function:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

Where p is the probability of belonging to a class (i.e. class 1)

Once on a log(odds) scale, we can interpret our coefficients just as we would a linear equation. Assuming all other features and weights remain constant:

- Continuous Variables: an increase of 1 unit increases the log(odds of belonging to a class) by the amount of the coefficient
- Binary Discrete Variables: the presence of the variable increases the log(odds of belonging to a class) by the amount of the coefficient

Optimization

With linear regression, we find the best fit line by minimizing the mean squared error (MSE). However, because our line is on a log(odds) scale ranging from -infinity to +infinity, all of our labeled observations have a value of either +/- infinity. This happens because we are 100% sure which class our label data belongs to. Therefore an observation belonging to class 1, for example, has a Pr(y = 1) = 1. Putting this into the logit function results in:

Since labeled observations have a y-value of +/- infinity, the values of the residuals, no matter the prospective line fit to the data, are +/- infinity. And without reasonable values for residuals, we can't use the least squares method to optimize our line. Logistic regression instead uses Maximum Likelihood for optimization.

$$
\begin{aligned}
logit(1) &= log(\frac{1}{(1-1)}) \\
&= log(1) - log(0) \\
&= 0 - -infinity \\
&= +infinity
\end{aligned}
$$

Example showing that labeled observations are +/- infinity on the logit scale

To find the maximum likelihood for our prospective line, we project the observations onto the line on the log(odds) scale (creating a prospective log-odds value for each observation), transform the log(odds) to probabilities, and calculate the log-likelihood of the line by adding the log(probabilities that each datapoint would be classified as its label). Then, similar to the least squares method, we keep rotating the log(odds) line and projecting the data onto it, transforming the log(odds) to probabilities, and calculating the log-likelihood. We do this until we find the line with the largest maximum likelihood.

For a better conceptual understanding of maximum likelihood applied to Logistic Regression, I recommend checking out this StatQuest video.

Implementation with Scikit-learn

I have some data about Twitter accounts, and I'll use logistic regression to predict if the accounts are real or not. My features for each Twitter account are:

- Follower Count
- Following Count
- Whether the account is private or not
- Media Count (# of tweets)
- Whether the account holder has populated the bio
- Whether the account holder offers an external URL
- And various multiplicative combinations of these

I've split my dataset into a training set and a testing set, and have transformed the continuous variables to a scale range of 0–1. I've also taken care of any null values. So, it's time to see if I can create a model that will predict whether an account is fake or not.

First step is importing scikit-learn:

```
from sklearn.linear_model import LogisticRegression
```

```
In [147]: C = [10, 1, .1, .01, .001]

for c in C:
    clf = LogisticRegression(penalty='l1', C=c, solver='liblinear')
    clf.fit(X_train_mm, y_train)
    print('C:', c)
    print('Training accuracy:', clf.score(X_train_mm, y_train))
    print('Test accuracy:', clf.score(X_test_mm, y_test))
    print('')
```

```
C: 10
Training accuracy: 0.9250439624853458
Test accuracy: 0.9264576618810431

C: 1
Training accuracy: 0.9231389214536928
Test accuracy: 0.9252856724289481

C: 0.1
Training accuracy: 0.9218200468933178
Test accuracy: 0.9249926750659244

C: 0.01
Training accuracy: 0.914932590855803
Test accuracy: 0.9188397304424261

C: 0.001
Training accuracy: 0.8574882766705745
Test accuracy: 0.8614122472897744
```

Because I have quite a few features and my intuition (after looking at a correlation matrix) is that not all will be useful, I opted for L1 regularization so that features can be penalized to 0. However, I wanted to tune the hyperparameter, C, so that regularization was useful but wasn't decreasing the accuracy of my model by too much.

You'll notice that my first step is to instantiate a LogisticRegression object. Here, I select `'l1'` regularization, my hyperparameter `C=c` and my solver. This is a pretty standard format for scikit-learn across the machine learning algorithms: you need create an object of the algorithm before fitting it to your data.

After some more testing, I opted for C=.1 because I felt it eliminated a good chunk of my features (reducing my model complexity) without too large of a compromise on accuracy.

Because scikit-learn syntax is almost exactly the same for many machine learning algorithms, I tend to use this function to fit classification models:

View entire gist

So, creating a LogisticRegression object and using my function looks like this:

```
In [116]: lr = LogisticRegression(penalty='l1', C=.1, n_jobs=-1)

In [117]: classify_users(lr, X_train_mm, X_test_mm, y_train, y_test)
```

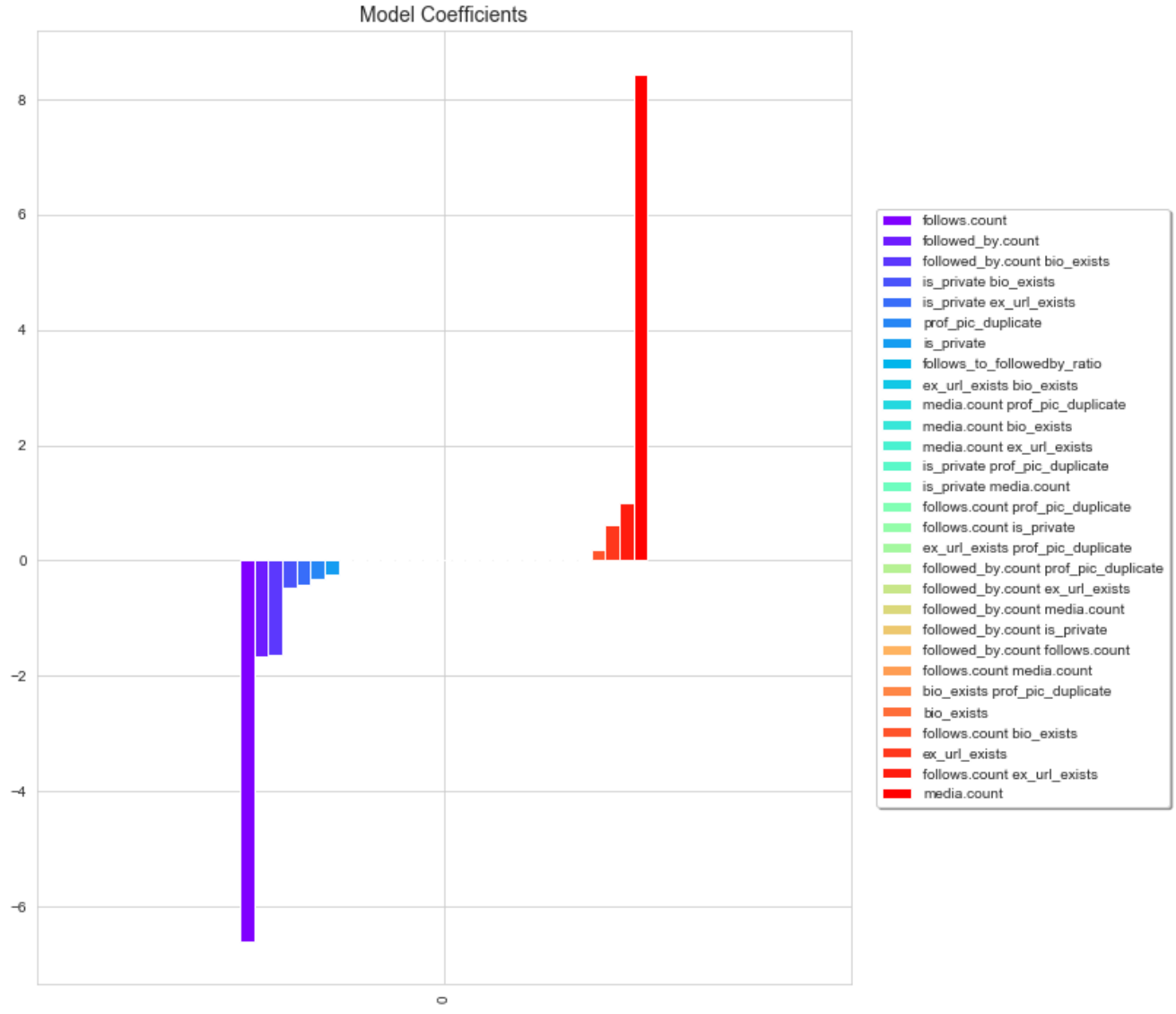
```
Training accuracy:  0.9218933177022274
Testing accuracy:  0.9249926750659244
Testing Report:
              precision    recall  f1-score   support

           0       0.95      0.88      0.91      1556
           1       0.90      0.96      0.93      1857

   micro avg       0.92      0.92      0.92      3413
   macro avg       0.93      0.92      0.92      3413
weighted avg       0.93      0.92      0.92      3413
```

As you can see, my test accuracy is a pretty decent 92.5%. This means that my model accurately classified 92.5% of the testing data. To find the weights for my features, I can call `lr.coef_`. After putting the coefficients into a data frame and sorting, we can see that a number of the features were not terribly useful to the accuracy of our model.

follows.count	-6.597470
followed_by.count	-1.665824
followed_by.count bio_exists	-1.630478
is_private bio_exists	-0.462082
is_private ex_url_exists	-0.420536
prof_pic_duplicate	-0.306358
is_private	-0.248918
follows_to_followedby_ratio	0.000000
ex_url_exists bio_exists	0.000000
media.count prof_pic_duplicate	0.000000
media.count bio_exists	0.000000
media.count ex_url_exists	0.000000
is_private prof_pic_duplicate	0.000000
is_private media.count	0.000000
follows.count prof_pic_duplicate	0.000000
follows.count is_private	0.000000
ex_url_exists prof_pic_duplicate	0.000000
followed_by.count prof_pic_duplicate	0.000000
followed_by.count ex_url_exists	0.000000
followed_by.count media.count	0.000000
followed_by.count is_private	0.000000
followed_by.count follows.count	0.000000
follows.count media.count	0.000000
bio_exists prof_pic_duplicate	0.000000
bio_exists	0.021367

follows.count bio_exists	0.182646
ex_url_exists	0.606525
follows.count ex_url_exists	0.991343
media.count	8.436876

And plotting the coefficients looks like this:



Conclusion

Logistic Regression is a useful classification algorithm that is easy to implement with scikit-learn. A regularized logistic regression can also useful for feature selection.

In addition to the code snippets here, my full Jupyter Notebooks can be found on my Github.

Please feel free to comment for any reason! I'd love to discuss your thoughts.

Resources

StatQuest with Josh Starmer: Logistic Regression Details Pt 1: Coefficients

Interpretable ML Book

Chris Albon

The Hundred-Page Machine Learning Book