Logistic Regression Four Ways with Python

Logistic regression is a predictive analysis that estimates/models the probability of event occurring based on a given dataset. This dataset contains both independent variables, or predictors, and their corresponding dependent variables, or responses.

To model the probability of a particular response variable, logistic regression assumes that the log-odds for the event is a linear combination of one or more predictors. The log-odds is literally the logarithm of the odds. The odds are the probability of the event occurring divided by the probability of the event not occurring. Stated mathematically,

$$\ln \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Here, p is the probability of the response, x_n are our predictors, and β_n are our parameters. The regression estimates the parameter of each predictor such that the above linear combination is the best fit of the log-odds. The most common method of estimating these parameters is [maximum liklihood estimation](#).

Types of Logistic Regression

There are three types of logistic regression algorithms:

1. Binary Logistic Regression

Re
Se

Da
vis

Da
ac

Re
ma

- the response/dependent variable is binary in nature
- this is the **most common** type of logistic regression
- example: is a tumor benign or malignant (0 or 1) based on one or more predictor

2. Ordinal Logistic Regression
    - response variable has 3+ possible outcomes and they have a specified order
    - example: which grade is a student likely to receive on scale of A through F based on one or more predictor
3. Multinomial Logistic Regression
    - the response variable has 3 or more possible outcomes but they have no specified order
    - example: which candy are people likely to prefer out of chocolate, hard candy, sour gummies, and sweet gummies based on one or more predictor

We use binary logistic regression for the Python demonstrations below.

Quick Note on Supervised Learning

Logistic regression is sometimes classified as a supervised learning, or supervised machine learning, algorithm. For brevity, we refer to supervised learning as SL.

SL is a subcategory of machine learning that uses a dataset, sometimes called the training dataset, to teach an algorithm to accurately predict a particular outcome. SL takes a training dataset consisting of independent input variables (predictors), x, and dependent output variables (response variables), y, and figures out a function that maps the inputs to the output. This function can then be used to generate outputs from new inputs based on what it "learned" from the training dataset.

Main Goal of SL (and thus logistic regression): Approximate the mapping function well so when new input data is fed into the function it accurately predicts the outputs for these new data.

Data

In this article, we will build binary logistic regression models with Python that will predict whether a breast cancer tumor is malignant or benign (malignant or benign will be our

Wc
evc

Su

Wa
inb
mo
Ser

Ca
re
ar

Pyt
logi
sta

response variable). We will use the Breast Cancer Wisconsin (Diagnostic) Data Set available from `sklearn.datasets`.

We can import this data as follows:

```
from sklearn.datasets import load_breast_cancer
cancer1 = load_breast_cancer()
```

`sklearn.datasets` loads the data into Python as a `sklearn.utils.Bunch` object. We can take a look at the predictors (independent variables) using the `feature_names` attribute and the response variable (dependent variable) using the `target_names` attribute.

```
print("Predictors: ", cancer1.feature_names)

Predictors:  ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']

print("\nResponse: ", cancer1.target_names)

Response:  ['malignant' 'benign']
```

Here we can see that there are 30 predictors available in our dataset that can be used in our logistic regression models.

Purely for convenience, we convert this `sklearn.utils.Bunch` object to a `pandas.DataFrame` as `pandas DataFrames` are a very commonly used data structure. We then use the `shape` property to see it has 569 rows and 30 columns.

```
import pandas as pd

cancer = pd.DataFrame(cancer1.data, columns=cancer1.feature_names)
cancer.columns = cancer.columns.str.replace(' ','_')
cancer.shape
```

```
(569, 30)
```

Each tumor, or each row, contains those same 30 predictors we saw above as columns.

It is important to note that the DataFrame as is does ***not*** contain the response variable information, it only contains the predictors! As mentioned at the beginning of this article, we want our dataset to contain input-output pairs so that we can train our logistic regression model to know which predictor values will likely result in either a malignant or benign tumor. We have the inputs (predictors), so we just need to add a column onto the DataFrame containing the output (responses).

Here we add the response variable column:

```
# Add a column for the response variable: malignant or benign
cancer['Target'] = cancer1.target

# Take a look at the DataFrame again to double check we added the column proper
cancer.shape
```

```
(569, 31)
```

We can now see that there is a new column containing the output/response information; whether each tumor is malignant (1) or benign (0).

Next, we will split up our predictor and response data into training datasets and testing datasets. Recall, we will use the training dataset to train our logistic regression models and then use the testing dataset to test the accuracy of model predictions. There is a nice function from `sklearn.model_selection` called `train_test_split` that splits a given dataset into 75% training and 25% testing data. Stetting `random_state=123` allows you to generate the same random train and test subsets used in this article. It's not strictly necessary to split data into training and testing sets when performing logistic regression. In fact if you have limited data it's not wise to do. However, we do it in this article to demonstrate how each method leads to the same results.

For the logistic regression examples, we will model malignant or benign as a function of the first 10 predictors (columns) in our dataset. These first 10 correspond to mean measurements of each tumor; mean radius, mean texture, mean perimeter, mean area, etc. (We selected these 10 columns purely for convenience to limit output. The goal of this article is to present different ways of performing logistic regression in Python, not how to select variables.)

```
from sklearn.model_selection import train_test_split

# Select the first 10 columns of our DataFrame that we will use as the predicto
x = cancer.iloc[:,:10]

# Select the response column
y = cancer.Target

# Split these data into training and testing datasets
x_train, x_test, y_train, y_test = train_test_split(x,y, random_state=123)
```

Determining Model Accuracy

In our examples below, we will need to assess the how well the models work at correctly classifying the test data. We will use two tools to assess the accuracy of the models: the **confusion matrix** and the **accuracy score**.

The **confusion matrix** is a matrix/table layout that provides a visualization of how well a certain algorithm (usually a supervised learning algorithm) performed.

In our examples below we will use the `sklearn.metrics.confusion_matrix` function to generate the confusion matrices.

For a binary classification model like logistic regression, the confusion matrix will be a 2x2 matrix with each row representing the counts of actual conditions and each column representing the counts of predicted conditions. Essentially, a confusion matrix is a contingency table with two dimensions: predicted and actual. Let's call the confusion matrix itself C. C$_{00}$ (the 00^{th} element or the top left matrix element) will show the count of true negatives, C$_{11}$ (the bottom right element) will show the count of true positives, C$_{01}$ (the top right element) will show the count of false positives, and C$_{10}$ (the bottom left element) will show the count of false negatives.

It's important to note that logistic regression returns *predicted probabilities*, not classifications. In order to make a classification we have to set a threshold. The most common threshold is 0.5. If a prediction is 0.5 or below, the classification is a 0. If it's higher than 0.5, the classification is a 1. Clearly a confusion matrix is not a perfect measure of model accuracy, especially when dealing with probabilities close to 0.5.

If the model performed perfectly, there would be no off-diagonal elements in the confusion matrix — it would only contain true negatives and true positives. The model is likely imperfect, so there will be off-diagonal elements in the confusion matrix. Having the majority of counts (larger numbers) in the true negative and true positive (diagonal) elements is a good indication that the model is working well.

The **accuracy score** is a metric given as the fraction of correct predictions generated by the given model. To calculate the accuracy score of our logistic regression models below we will use the `sklearn.metrics.accuracy_score` function. It computes the accuracy score as follows:

$$\texttt{accuracy} = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y_i} = y_i)$$

where $1(\dots)$ is the indicator function.

If the model performed perfectly and was able to correctly classify every sample in the test dataset, the accuracy score would return a 1.0 (100%). So the closer to 1.0 or 100% the better the model performance.

Now we are ready to build our logistic regression models!

Method 1: statsmodels.formulas.api.Logit( )

For this first example, we will use the `Logit()` function from the `statsmodels.formula.api` package to fit our model. This function requires two arguments: a string formula detailing which columns are predictors and response variables, and another specifying the `pandas.DataFrame` dataset.

The formula should be input in a format similar to R’s formula syntax: `"output ~ predictor1 + predictor2 + predictor3 + ... + predictorN"`.

```
import statsmodels.formula.api as smf
import pandas as pd

# Create the formula string
```

```
all_columns = ' + '.join(cancer.columns[:10])
formula = "Target ~ " + all_columns
print("Formula: ", formula, "\n")

Formula:  Target ~ mean_radius + mean_texture + mean_perimeter + mean_area + mea
mean_concavity + mean_concave_points + mean_symmetry + mean_fractal_dimension

# Put the training predictors and responses into one DataFrame to be input into
trainingdata = pd.concat([x_train,y_train], axis = 1)

# Build the model
log_reg_1 = smf.logit(formula, data=trainingdata).fit()

Optimization terminated successfully.
         Current function value: 0.131593
         Iterations 11
```

We can look at the model summary using the summary() method:

```
log_reg_1.summary()
```

Logit Regression Results

- Dep. Variable: Target
- Model: Logit
- Method: MLE
- Date: Thu, 01 Jun 2023
- Time: 0.503020833333333
- converged: TRUE
- Covariance Type: nonrobust
- No. Observations: 426
- Df Residuals: 415
- Df Model: 10
- Pseudo R-squ.: 0.8004

- Log-Likelihood: -56.059
- LL-Null: -280.92
- LLR p-value: 2.402E-90

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	17.1132	15.041	1.138	0.255	-12.368	46.594
mean_radius	-0.1786	4.151	-0.043	0.966	-8.315	7.958
mean_texture	-0.4333	0.079	-5.479	0.000	-0.588	-0.278
mean_perimeter	0.2724	0.569	0.479	0.632	-0.842	1.387
mean_area	-0.0301	0.019	-1.617	0.106	-0.067	0.006
mean_smoothness	-90.0668	38.119	-2.363	0.018	-164.778	-15.356
mean_compactness	-9.5962	23.305	-0.412	0.681	-55.273	36.080
mean_concavity	-11.5117	9.451	-1.218	0.223	-30.035	7.011
mean_concave_points	-47.8953	32.398	-1.478	0.139	-111.394	15.603
mean_symmetry	-23.1823	12.730	-1.821	0.069	-48.133	1.768
mean_fractal_dimension	91.5592	98.116	0.933	0.351	-100.745	283.863

Now we can test our model on the testing data using the predict() method. We can also determine the accuracy of the model by looking at the accuracy score and confusion matrix from the sklearn.metric package.

```
from sklearn.metrics import confusion_matrix, accuracy_score

# Predict responses
pred_1 = log_reg_1.predict(x_test)
# round() rounds to nearest integer;
# 0.5 rounds to 0; 0.501 rounds to 1
prediction_1 = list(map(round, pred_1))

# Accuracy score
print('\nTest accuracy = ', accuracy_score(y_test, prediction_1))

Test accuracy =  0.958041958041958

# Confusion matrix
```

```
cm = confusion_matrix(y_test, prediction_1)
print ("\nConfusion Matrix : \n", cm)

Confusion Matrix :
 [[49  5]
 [ 1 88]]
```

Method 2: sklearn.linear_model.LogisticRegression()

In this example, we will use the `LogisticRegression()` function from `sklearn.linear_model` to build our logistic regression model. The `LogisticRegression()` function implements regularized logistic regression by default, which is different from traditional estimation procedures. To get estimates similar to the other methods presented in this article we need to set `penalty = None` and `solver = 'newton-cg'`. The former turns off regularization and the latter uses the common Newton-Raphson algorithm for estimating parameters. The default maximum number of solver iterations is 100. We need to slightly increase this number to avoid convergence warnings, hence the setting `max_iter = 150`.

Here we import the function and fit the model to the training data:

```
from sklearn.linear_model import LogisticRegression

log_reg_2 = LogisticRegression(penalty=None, solver = 'newton-cg', max_iter= 15
```

sklearn does not provide a built-in function to show the summary of a regression model. It does however provide functions to extract the model coefficients and intercept:

```
print("Model Coefficients: ", log_reg_2.coef_)

Model Coefficients:  [[-1.78640821e-01 -4.33339411e-01  2.72448462e-01 -3.00983
  -9.00667340e+01 -9.59622388e+00 -1.15116969e+01 -4.78953170e+01
  -2.31822640e+01  9.15591120e+01]]

print("\nModel Intercept: ",log_reg_2.intercept_)

Model Intercept:  [17.1131667]
```

Notice the results are slightly different from the previous method.

Now we input our testing predictors into the model to predict response variables using the `predict()` method. We then determine how accurate the model was using the `score()`

method and a confusion matrix. Notice the result is identical to the previous method.

```
# Predict responses
pred_2 = log_reg_2.predict(x_test)
prediction_2 = list(map(round, pred_2))

# Accuracy score
print('\nTest accuracy = ', accuracy_score(y_test, prediction_2))

Test accuracy =  0.958041958041958

# Confusion matrix
cm = confusion_matrix(y_test, prediction_2)
print ("\nConfusion Matrix : \n", cm)

Confusion Matrix :
 [[49  5]
 [ 1 88]]
```

Method 3: statsmodels.api.Logit()

For this example, we will use the `Logit()` function from `statsmodels.api` to build our logistic regression model. This method and the next one require that a constant be added to the training set in order to estimate an intercept. This is simply a column of ones. We use the `add_constant()` function from `statsmodels.tools` to do this for us.

```
import statsmodels.api as sm
import statsmodels.tools as tools

# add constant to training data
x_train_const = tools.add_constant(x_train)
# Train the logistic regression with the training data
log_reg_3 = sm.Logit(y_train, x_train_const).fit()

Optimization terminated successfully.
         Current function value: 0.131593
         Iterations 11
```

We can look at the model summary using the summary() method:

```
print(log_reg_3.summary())
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                 Target   No. Observations:                  426
Model:                          Logit   Df Residuals:                      415
Method:                           MLE   Df Model:                           10
Date:                Thu, 01 Jun 2023   Pseudo R-squ.:                  0.8004
Time:                        12:04:29   Log-Likelihood:                -56.059
converged:                       True   LL-Null:                       -280.92
Covariance Type:            nonrobust   LLR p-value:                 2.402e-90
================================================================================
                           coef    std err          z      P>|z|      [0.025
--------------------------------------------------------------------------------
const                   17.1132     15.041      1.138      0.255     -12.368
mean_radius             -0.1786      4.151     -0.043      0.966      -8.315
mean_texture            -0.4333      0.079     -5.479      0.000      -0.588
mean_perimeter           0.2724      0.569      0.479      0.632      -0.842
mean_area               -0.0301      0.019     -1.617      0.106      -0.067
mean_smoothness        -90.0668     38.119     -2.363      0.018    -164.778
mean_compactness        -9.5962     23.305     -0.412      0.681     -55.273
mean_concavity         -11.5117      9.451     -1.218      0.223     -30.035
mean_concave_points    -47.8953     32.398     -1.478      0.139    -111.394
mean_symmetry          -23.1823     12.730     -1.821      0.069     -48.133
mean_fractal_dimension  91.5592     98.116      0.933      0.351    -100.745
================================================================================
```

Now we can test our model on the testing data using the `predict()` method and then determine the accuracy of the model by looking at the accuracy score and confusion matrix. Once again the result is identical to the previous methods

```
# add constant to test data
x_test_const = tools.add_constant(x_test)
# Predict responses
pred_3 = log_reg_3.predict(x_test_const)
prediction_3 = list(map(round, pred_3))

# Accuracy score
print('\nTest accuracy = ', accuracy_score(y_test, prediction_3))
```

```
Test accuracy =  0.958041958041958

# Confusion matrix
cm = confusion_matrix(y_test, prediction_3)
print ("\nConfusion Matrix : \n", cm)

Confusion Matrix :
 [[49  5]
 [ 1 88]]
```

Method 4: statmodels.api.GLM()

For the final example we will use the generalized linear model `GLM()` function from `statsmodels.api` to build our logistic regression model. `GLM()` can be used to build a number of different models, so in addition to providing the training data to `GML()`, we also specify that we want to build a logistic regression model by setting `family=sm.families.Binomial()` in the argument of the function:

```
import statsmodels.api as sm

log_reg_4 = sm.GLM(y_train, x_train, family=sm.families.Binomial()).fit()
```

Now we can look at the model summary:

```
log_reg_4.summary()
```

Generalized Linear Model Regression Results

- Dep. Variable: Target
- Model: GLM
- Model Family: Binomial
- Link Function: Logit
- Method: IRLS
- Date: Thu, 01 Jun 2023
- Time: 0.503125

- No. Iterations: 9
- Covariance Type: nonrobust
- No. Observations: 426
- Df Residuals: 416
- Df Model: 9
- Scale: 1
- Log-Likelihood: -56.766
- Deviance: 113.53
- Pearson chi2: 186
- Pseudo R-squ. (CS): 0.6509

	coef	std err	z	P>|z|	[0.025	0.975]
mean_radius	1.8376	3.836	0.479	0.632	-5.681	9.356
mean_texture	-0.4124	0.076	-5.455	0.000	-0.561	-0.264
mean_perimeter	0.2176	0.567	0.384	0.701	-0.894	1.329
mean_area	-0.0473	0.011	-4.227	0.000	-0.069	-0.025
mean_smoothness	-82.2700	34.902	-2.357	0.018	-150.676	-13.864
mean_compactness	-18.0799	22.109	-0.818	0.413	-61.412	25.253
mean_concavity	-8.8574	8.658	-1.023	0.306	-25.826	8.112
mean_concave_points	-56.3369	30.533	-1.845	0.065	-116.180	3.506
mean_symmetry	-19.6757	11.902	-1.653	0.098	-43.004	3.652
mean_fractal_dimension	150.5408	82.905	1.816	0.069	-11.949	313.031

And once again we can test our model on the testing data using the same `predict()` method as above and examine the accuracy score and confusion matrix.

```
# Predict responses
pred_4 = log_reg_4.predict(x_test)
prediction_4 = list(map(round, pred_4))
```

```
# Accuracy score
print('\nTest accuracy = ', accuracy_score(y_test, prediction_4))

Test accuracy =  0.958041958041958

# Confusion matrix
cm = confusion_matrix(y_test, prediction_4)
print ("\nConfusion Matrix : \n", cm)

Confusion Matrix :
 [[49  5]
 [ 1 88]]
```

Quick Summary of the Logistic Regression Process

Throughout this article we worked through four ways to carry out a logistic regression with Python. While these methods were all done with different packages, they all followed the same general steps:

1. **Organize** the dataset such that it contains both predictors and responses (input-output pairs)
2. **Split** the dataset into training and testing datasets
3. **Fit** the logistic regression model to the training dataset
4. Use the testing dataset with the model to **predict** testing dataset outcomes
5. Determine the **accuracy** of the model from these predictions

Again, it’s not always necessary to split your data into training and test sets, but it can be an effective way to compare the performance of different models as we did in this article.

References

- sklearn.dataset Breast Cancer data set: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html#sklearn.datas
- the actual site for the Breast Cancer data set: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

Samantha Lomuscio
StatLab Associate
University of Virginia Library
September 22, 2022
Updated June 01, 2023
