Logistic Regression with Python

José Luis Higuera Caraveo

Febrero 28 2022

1 Logistic Regression

Logistic regression is a type of regression analysis used to predict the result of a categorical variable, that is, used to classify a data set according to the possible categories given by the variable to be predicted.

1.0.1 Hypothesis

The algorithm predicts the probability that a certain example belongs to a certain category. But, for each case study, an approval threshold have to be specified, it is a number from 0 to 1, given by the user, who will determine, if the probability is greater than this threshold, then this example belongs to a certain category.

Having this clear, our hypothesis is as follows:

We are going to use a threshold of 0.5, therefore:

- If $h_0(x) \geq 0.5$, the prediction will be "$y = 1$"
- If $h_0(x) \leq 0.5$, the prediction will be "$y = 0$"

Note that $0 \leq h_0 \leq 1$.

And how to get h_0:

$$h_0(x) = g(\theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \cdots + \theta_n \cdot x_n) = g(z)$$

where:

- $g(z) = \frac{1}{1+e^{-\theta^T \cdot X}}$

Once we analyze the formulas. First, the value of $\theta^T \cdot X$ is obtained and a function is applied to this result which can convert the result of $\theta^T \cdot X$ to one that ranges from 0 to 1, and thus, given the threshold, we can decide if we can classify that example as "$y = 1$" or "$y = 0$".

This function is called the sigmoid function:

$$Sigmoid = \frac{1}{1 + e^{-X}}$$

1.0.2 Cost Function

The cost function for a linear regression is as follows:

$$J(\theta_0,\ \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_0(x^i) - y^i)^2$$

For logistic regression we cannot use this one, since this will result in a non-convex function, so it would never be possible to find an optimal global minimum to minimize it.

We have to modify this function in order to work with the two possible values "$y = 1$" and "$y = 0$". Our cost function would look like this:

$$Cost(h_0(x), y) = \begin{cases} -log(h_0(x)) & if\ y = 1 \\ -log(1 - h_0(x)) & if\ y = 0 \end{cases}$$

But the above may be difficult to understand, for this we simplify the function as follows:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_0(x_i), y_i)$$

Taking as reference the equations for "$y = 1$" and "$y = 0$", we have a formula where we can work with the two possible results:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y_i log(h_0(x_i)) + (1 - y_i) log(1 - h_0(x_i))\right]$$

To make the prediction of a new element x:

$$Output\ h_0(x) = \frac{1}{1 + e^{-\theta^T x}}$$

1.0.3 Gradient Descent

How do we get the optimal θ?

As in the linear regression algorithm, we will use gradient descent to help us find these parameters. The algorithm is as follows:

Repeat until converge {

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_0(x^i) - y^i) \cdot x_j^i$$

}

where: * α is the learning rate.

We notice that the algorithm looks identical to the linear regression algorithm, but we must make something clear, now, the function to calculate $h_0(x)$ is different, you have to use:

$$h_0(x) = \frac{1}{1 + e^{-\theta^T x}}$$

1.0.4 Code Implementation

An example dataset that helps us understand how the logistic regression algorithm is used is the dataset for predicting malignant or benign tumors based on certain characteristics.

The dataset that will be used provides us with characteristics of the tumors such as

- identification
- diagnosis
- average_radius
- medium_texture
- mean_perimeter
- mean_area
- medium_smooth
- medium_compactness
- half_concavity
- concave_mean points
- mean_symmetry

Among other important information. For this, we will try to predict the diagnosis (M = Malignant, B = Benign) according to the specifications of each tumor.

The data is found in the following link: https://www.kaggle.com/yasserh/breast-cancer-dataset

[1]:
```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.style.use('ggplot')
```

[2]:
```
data = pd.read_csv('../data/breast-cancer.csv')
data.head()
```

[2]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	\
0	842302	M	17.99	10.38	122.80	1001.0	
1	842517	M	20.57	17.77	132.90	1326.0	
2	84300903	M	19.69	21.25	130.00	1203.0	
3	84348301	M	11.42	20.38	77.58	386.1	
4	84358402	M	20.29	14.34	135.10	1297.0	

	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	\
0	0.11840	0.27760	0.3001	0.14710	
1	0.08474	0.07864	0.0869	0.07017	
2	0.10960	0.15990	0.1974	0.12790	
3	0.14250	0.28390	0.2414	0.10520	
4	0.10030	0.13280	0.1980	0.10430	

```
   …  radius_worst  texture_worst  perimeter_worst  area_worst  \
0  …         25.38          17.33           184.60      2019.0
1  …         24.99          23.41           158.80      1956.0
2  …         23.57          25.53           152.50      1709.0
3  …         14.91          26.50            98.87       567.7
4  …         22.54          16.67           152.20      1575.0

   smoothness_worst  compactness_worst  concavity_worst  concave points_worst  \
0            0.1622             0.6656           0.7119                0.2654
1            0.1238             0.1866           0.2416                0.1860
2            0.1444             0.4245           0.4504                0.2430
3            0.2098             0.8663           0.6869                0.2575
4            0.1374             0.2050           0.4000                0.1625

   symmetry_worst  fractal_dimension_worst
0          0.4601                  0.11890
1          0.2750                  0.08902
2          0.3613                  0.08758
3          0.6638                  0.17300
4          0.2364                  0.07678

[5 rows x 32 columns]
```

We have 569 examples and 30 characteristics, we do not take into account the ID, nor the diagnosis.

[3]:
```
data['diagnosis'][data['diagnosis'] == 'M'].count(), \
data['diagnosis'][data['diagnosis'] == 'B'].count()
```

[3]:
```
(212, 357)
```

The dataset contains 212 cases of malignant tumors and 357 cases of benign tumors.

When a regression algorithm is going to be applied, it is important to verify that the data which it is going to work is numerical. Otherwise, some transformation work will have to be done to help us convert the categorical variables to numeric.

[57]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 32 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   id                      569 non-null    int64
 1   diagnosis               569 non-null    object
 2   radius_mean             569 non-null    float64
 3   texture_mean            569 non-null    float64
 4   perimeter_mean          569 non-null    float64
```

```
 5   area_mean                569 non-null    float64
 6   smoothness_mean          569 non-null    float64
 7   compactness_mean         569 non-null    float64
 8   concavity_mean           569 non-null    float64
 9   concave points_mean      569 non-null    float64
 10  symmetry_mean            569 non-null    float64
 11  fractal_dimension_mean   569 non-null    float64
 12  radius_se                569 non-null    float64
 13  texture_se               569 non-null    float64
 14  perimeter_se             569 non-null    float64
 15  area_se                  569 non-null    float64
 16  smoothness_se            569 non-null    float64
 17  compactness_se           569 non-null    float64
 18  concavity_se             569 non-null    float64
 19  concave points_se        569 non-null    float64
 20  symmetry_se              569 non-null    float64
 21  fractal_dimension_se     569 non-null    float64
 22  radius_worst             569 non-null    float64
 23  texture_worst            569 non-null    float64
 24  perimeter_worst          569 non-null    float64
 25  area_worst               569 non-null    float64
 26  smoothness_worst         569 non-null    float64
 27  compactness_worst        569 non-null    float64
 28  concavity_worst          569 non-null    float64
 29  concave points_worst     569 non-null    float64
 30  symmetry_worst           569 non-null    float64
 31  fractal_dimension_worst  569 non-null    float64
dtypes: float64(30), int64(1), object(1)
memory usage: 142.4+ KB
```

All of our data is numeric, so no transformation is necessary before applying the algorithm, also we do not have null data.

[4]:
```
# Variables to be used for the algorithm
X = data.iloc[:, 2:]
m = len(X)
y = data['diagnosis']
```

The variable to predict is categorical, so it has to be transformed to numeric. Normally an encoding technique is used, but in this case study we have two possible outcomes, therefore:

- $y = 1$ when the diagnosis is "M".
- $y = 0$ when the diagnosis is "B".

[5]:
```
y = y.apply(lambda x: 0 if x == 'B' else 1)
```

In order to handle a vectorized solution, it is necessary to add a column of ones at the beginning of the X variables, this helps us that θ_0 is not modified.

[6]:
```
ones = [1] * len(X)

X.insert(0, 'ones', ones)
X = X.values
```

[8]:
```
import warnings
warnings.filterwarnings('ignore')
```

[10]:
```
# Sigmoid function
def sigmoid(x):
    return 1/(1 + np.e**(-x))

# Getting Predictions
def get_y_pred(x, thetas):
    return sigmoid(x.dot(thetas.T))

# Getting the cost
def get_cost(y, y_pred):
    cost = 0
    for i in range(m):
        cost += (y[i] * np.log(y_pred[i])) + ((1-y[i]) * np.log(1-y_pred[i]))
    return -1 * (1/m) * cost

# Gradient Descent
def get_gradient(x, y, n_iter, thetas, alpha=0.01):
    for i in range(n_iter):
        y_pred = get_y_pred(x, thetas)

        thetas = thetas - (alpha * ((1 / m)*((y_pred - y).T.dot(x))))

    return thetas


# Optimal theta
initial_thetas = np.zeros([X.shape[1]]).T
n_iter = 100000
thetas = get_gradient(X, y, n_iter, initial_thetas)

# Obtaining the predictions according to the optimal theta.
y_pred = get_y_pred(X, thetas)

# Get 0 or 1 according to the threshold.
threshold = 0.5
y_pred2 = [1 if pred > threshold else 0 for pred in y_pred]
```

The best way to check how well our model is predicting is to use a confusion matrix, this helps us check for true positives, true negatives, false positives and false negatives.

The Sklearn library provides us with a function to obtain this matrix. Which results in an array as follows:

$$\begin{pmatrix} True\ Positives & False\ Positives \\ False\ Negatives & True\ Negatives \end{pmatrix}$$

This matrix also helps us calculate:

- Accuracy: Of the predictions, what is the proportion that the algorithm predicts correctly.

$$Accuracy = \frac{True\ Positives\ +\ True\ Negatives}{m}$$

- Precision: Of the total number of positive predictions, what proportion is actually true positive.

$$Precision = \frac{True\ Positives}{True\ Positives\ +\ False\ Positives}$$

- Recall: Of the total number of positive predictions, what proportion actually predicts as a true positive.

$$Recall = \frac{True\ Positives}{True\ Positives\ +\ False\ Negatives}$$

[11]:
```
from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(y, y_pred2)

VP = conf_matrix[0][0]
FP = conf_matrix[0][1]
VN = conf_matrix[1][1]
FN = conf_matrix[1][0]

accuracy = (VP + VN) / m
presicion = VP / (VP + FP)
recall = VP / (VP + FN)

print('Accuracy: {}, Precision: {}, Recall: {}'.format(accuracy, presicion,␣
 ↪recall))
```

```
Accuracy: 0.9121265377855887, Presicion: 0.9971988795518207, Recall:
0.8790123456790123
```

Our algorithm obtains an accuracy of almost 100%, with high precision and recall. Therefore, it is concluded that this algorithm helps to predict with almost 100% accuracy a malignant or benign tumor.

1.0.5 Algorithm with Sklearn

[12]:
```
from sklearn.linear_model import LogisticRegression

X = data.iloc[:, 2:].values
m = len(X)
y = data['diagnosis']

# Converting categorical data to numeric
y = y.apply(lambda x: 0 if x == 'B' else 1)

# Training the model
model = LogisticRegression()
model.fit(X, y)

# Getting predictions
y_pred = model.predict(X)

# Confusion Matrix
conf_matrix = confusion_matrix(y, y_pred2)

VP = conf_matrix[0][0]
FP = conf_matrix[0][1]
VN = conf_matrix[1][1]
FN = conf_matrix[1][0]

accuracy = (VP + VN) / m
presicion = VP / (VP + FP)
recall = VP / (VP + FN)

print('Accuracy: {}, Precision: {}, Recall: {}'.format(accuracy, presicion,␣
↪recall))
```

```
Accuracy: 0.9121265377855887, Precision: 0.9971988795518207, Recall:
0.8790123456790123
```

We can see how we obtain the same results, but with the advantage that now the procedure is applied with a few lines of code.