Logistic regression from scratch in Python

COMPARING THE FUNCTION FOR LOGISTIC REGRESSION FROM SK-LEARN WITH OUR OWN FUNCTION, WRITTEN “FROM SCRATCH”

Author: M. Kolaksazov
Source: Nick Becker, data scientist (https://beckernick.github.io/logistic-regression-from-scratch/)

Short summary, describing the problem

Task: write a piece of code, describing the ***logistic regression*** model for classification of categorical data

Logistic regression: a generalized linear model, used to predict ***categorical*** variables. In logistic regression, vector parameters, called weights must be found out, that maximize the likelihood of the given data.

Procedure:

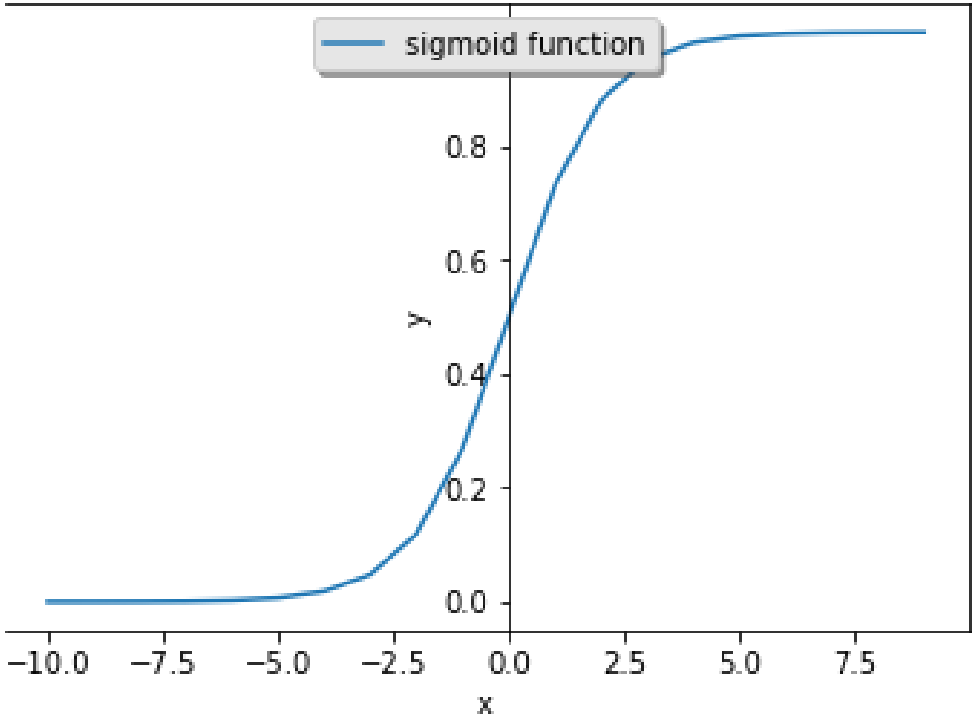
1. First, the so-called ***link function*** is used to transform the variables of the data from the continuous scale to a categorical response (0 or 1 in the simplest case). After the transformation is complete, the relationship between the predictors and the response can be modeled with linear regression.
2. Next, the ***likelihood*** of the data must be ***maximized***, which was done by the means of the ***log-likelihood function***.
3. Afterwards, the ***gradient***, as well as the ***gradient ascent*** must be calculated.
4. Finally, the function for the ***logistic regression*** can be implemented and the model can be calculated with an ***intercept***

Transformation of input data by the means of the link function

A ***sigmoid*** function was used here, as the link function: $\sigma(z) = \frac{1}{1+e^{-z}}$

The sigmoid function is often used as ***link function*** in logistic regression, when ***linear models*** must be transformed. Thus, values between ***0*** and ***1*** are given to the linear parameters.

$\tilde{y}_i = \sigma(z_i);\ \sigma(\omega x_i) = \frac{1}{1+e^{-\omega x_i}}$



Log-likelihood function

In the current task, instead of the cost of loss function was used the log-likelihood function :

Log-likelihood function:

$$\ell(\beta) = \sum_{i=1}^{N} \left\{ y_i \log p(x_i;\beta) + (1-y_i)\log(1-p(x_i;\beta)) \right\}$$

$$= \sum_{i=1}^{N} \left\{ y_i \beta^T x_i - \log(1+e^{\beta^T x_i}) \right\}.$$

In the upper formula, y is the class of the data (0 or 1), x_i is an individual data point, and β is the weights vector

Cost of loss function:

$$J = -\frac{1}{m}\sum_{i=1}^{m} y_{(i)} log(\tilde{y}_{(i)}) + (1-y_{(i)}) log(1-\tilde{y}_{(i)})$$

$\tilde{y}$ - output of the logistic regression algorithm (must be as close as possible to y)

The latter is used more often and represents the mean arithmetic of the loss error function for every input data variable. The aim is to minimize the overall cost function, which will give the optimal values for the weight parameter. In contrast, the log-likelihood function has to be maximal, in order to obtain optimal weight vector parameters.

Calculation of gradient, as well as the gradient ascent

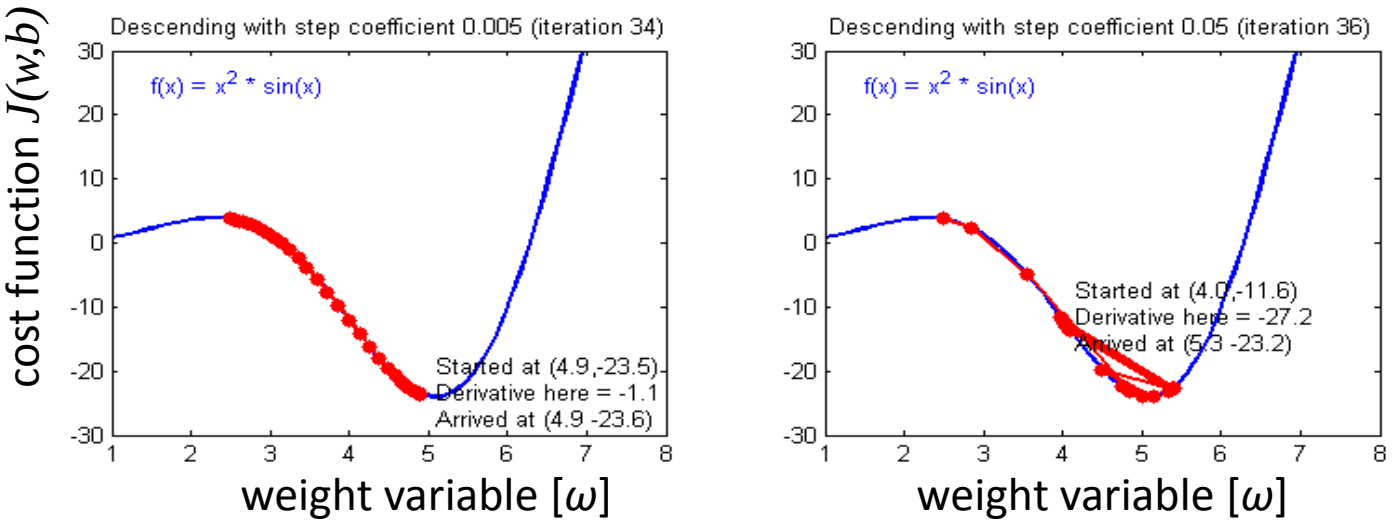
The optimal values for the weight parameters can be found, when the gradient becomes equal to 0. At this point the likelihood of the data becomes maximal. When enough time and sufficiently small learning rate are given, the gradient ascent on a concave function will always reach the global optimum. The learning rate must be chosen in such way, that it is not too small, which will slow down the calculation, or too high, leading to diverging of the gradient ascent algorithm and wrong results.

Gradients:

$$dz = y_i - \tilde{y}_i$$
$$d\omega = x_i^{\mathrm{T}} dz$$

Gradient ascent:

$$\omega = \omega + \alpha d\omega$$



A gradient descent algorithm with a good learning rate (converging) and a bad learning rate (diverging). In this case it is gradient descent, because the cost of loss function is calculated, which must me minimal. (Images: Adam Harley)

Implementing the task in Python 3

Generating the input data

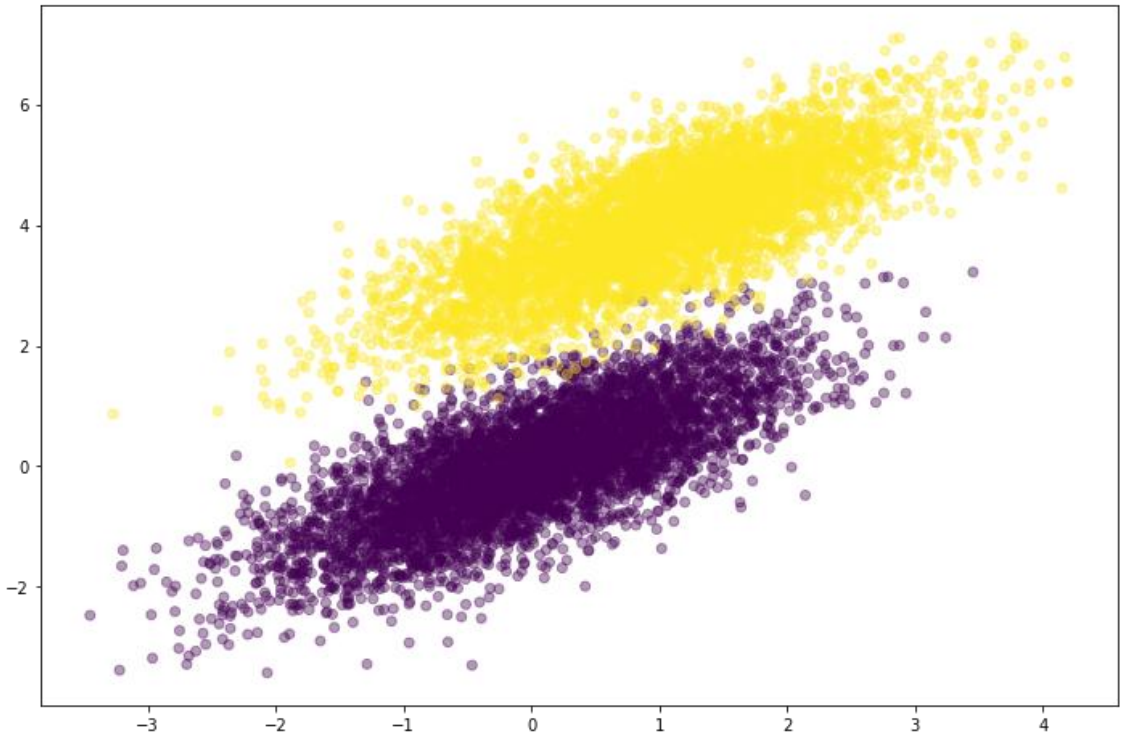
The initial input data was be generated. The random number generator of Python 3 was used for this purpose.

two samples of random data, with the same deviation, but different mean, the first labelled as 0, the second labelled as 1. The Python function (**multivariative_normal**) was used to generate random numbers with normal (Gaussian) distribution.

```
np.random.seed(12)
num_observations = 5000

x1 = np.random.multivariate_normal([0, 0], [[1, .75],[.75, 1]], num_observations)
x2 = np.random.multivariate_normal([1, 4], [[1, .75],[.75, 1]], num_observations)

simulated_separableish_features = np.vstack((x1, x2)).astype(np.float32)
simulated_labels = np.hstack((np.zeros(num_observations),
                              np.ones(num_observations)))
plt.figure(figsize=(12,8))
plt.scatter(simulated_separableish_features[:, 0],
            simulated_separableish_features[:, 1],
            c = simulated_labels, alpha = .4)
```



Defining functions for the logistic regression algorithm

Implementing the logistic regression as function in Python

Defining the sigmoid function:

```
def sigmoid(z):
    s = 1/(1+np.exp(-z))
    return s
```

Defining the log-likelihood function:

```
def log_likelihood(X, Y, w):
    A = np.dot(w, X)
    ll = np.sum( Y*A - np.log(1 + np.exp(A)) )
    return ll
```

Defining the function of the logistic regression:

```
def logistic_regression(X, Y, num_iterations, alpha, add_intercept = False):
    if add_intercept:
        intercept = np.ones((X.shape[0], 1))
        X = np.hstack((intercept, X))
    w = np.zeros(X.shape[1])

    for i in (num_iterations):
        z = np.dot(w, X)
        A = sigmoid(z)
        # gradients
        dz = Y - A
        dw = np.dot(X.T, dz)
        w += alpha * dw

        if i % 10000 == 0:
            print log_likelihood(X, Y, w)
    return w
```

It is important to vectorize the variables by the means of the NumPy functions `np.dot(), np.sum(), np.prod()` etc., as compared with using "`for loop`", because loops slow down the code. In addition, it is recommended to initialize the weight variables using the random number generator, when implementing logistic regression in neural networks, as compared with the initialization with zeros because the functions in the nodes will become symmetric and will calculate the same function

Comparing our written function "from scratch" with the function from the sk-learn library

The performance of both functions can be compared. The accuracy is the same, as well as the weight parameters, used in calculation. The time for calculation, however, is clearly much lower for the sk-learn function (about 300 times lower)

```
tic = tm.process_time()
weights = logistic_regression(simulated_separableish_features, simulated_labels,
                      num_steps = 300000, learning_rate = 5e-5, add_intercept=True)
toc = tm.process_time()
t = (toc - tic)
print('time in seconds from scratch: {0}'.format(t))

print(clf.intercept_, clf.coef_)
print(weights)

tic = tm.clock()
clf = LogisticRegression(fit_intercept=True, C = 1e15)
clf.fit(simulated_separableish_features, simulated_labels)
toc = tm.clock()
t = (toc - tic)
print('time in seconds from sk-learn: {0}'.format(t))

print('Accuracy from scratch: {0}'.format((predictions
    == simulated_labels).sum().astype(float) / len(preds)))
print('Accuracy from sk-learn: {0}'.format(clf.score(
    simulated_separableish_features, simulated_labels)))
```

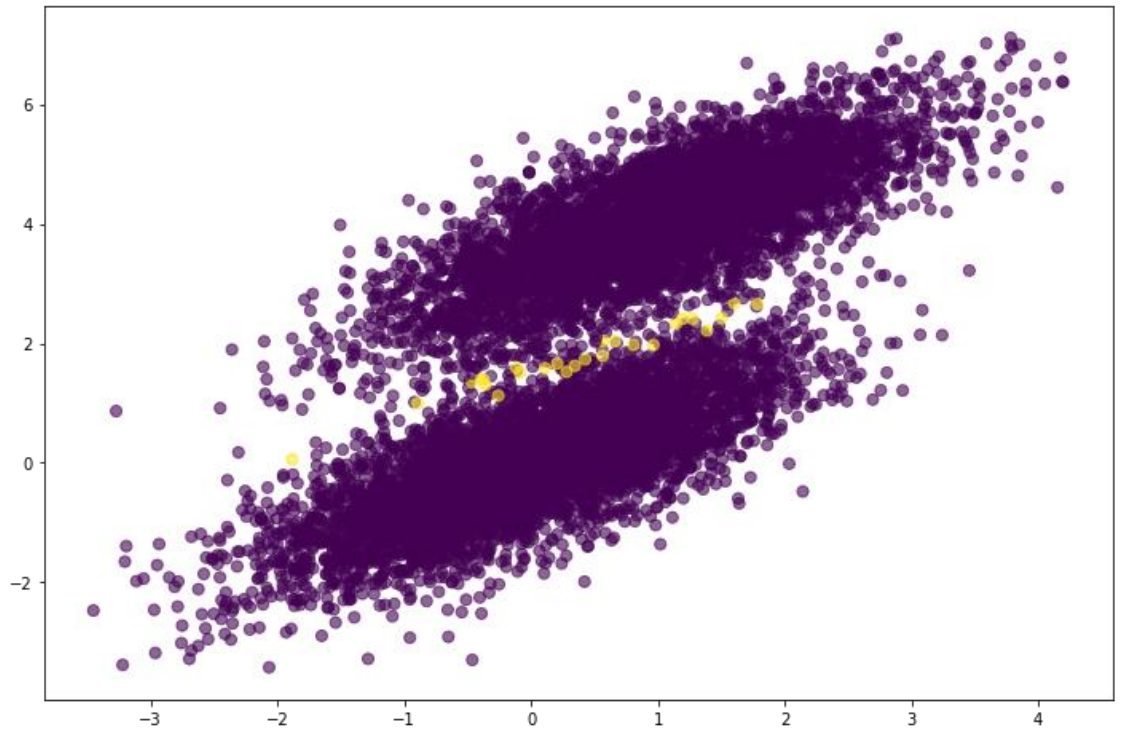
```
time in seconds from scratch: 72.6875
time in seconds from sk-learn: 0.024149615957867354
[-13.99400797] [[-5.02712572  8.23286799]]
[-14.09225541  -5.05899648   8.28955762]
Accuracy from scratch: 0.9948
Accuracy from sk-learn: 0.9948
```

} weight parameters

Graphical representation of the results

```
plt.figure(figsize = (12, 8))
plt.scatter(simulated_separableish_features[:, 0],
            simulated_separableish_features[:, 1],
            c = predictions == simulated_labels - 1,
            alpha = .6, s = 50)
```

Graph, showing the two samples of data, both in blue, whereas the yellow dots show the data, that was not predicted by the means of the function



Thank you for your attention!