Logistic Regression from scratch in Python

Martín Pellarolo Feb 23, 2018 · 4 min read

While Python's scikit-learn library provides the easy-to-use and efficient LogisticRegression class, the objective of this post is to create an own implementation using NumPy. Implementing basic models is a great idea to improve your comprehension about how they work.

Data set

We will use the well known Iris data set. It contains 3 classes of 50 instances each, where each class refers to a type of iris plant. To simplify things, we take just the first two feature columns. Also, the two non-linearly separable classes are labeled with the same category, ending up with a binary classification problem.

```
iris = sklearn.datasets.load_iris()
X = iris.data[:, :2]
y = (iris.target != 0) * 1
```

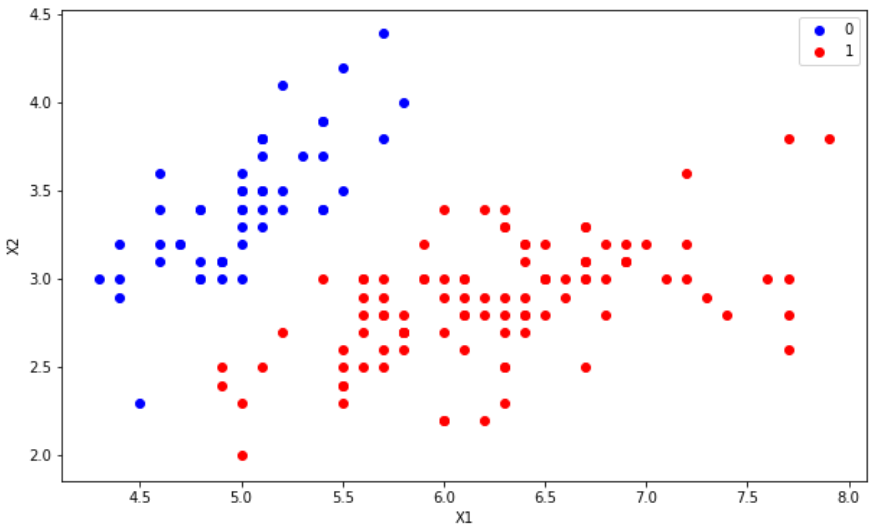


Fig. 1 — Training data

Algorithm

Given a set of inputs X, we want to assign them to one of two possible categories (0 or 1). Logistic regression models the probability that each input belongs to a particular category.

Hypothesis

A function takes inputs and returns outputs. To generate probabilities, logistic regression uses a function that gives outputs between 0 and 1 for all values of X. There are many functions that meet this description, but the used in this case is the *logistic function*. From here we will refer to it as *sigmoid.*

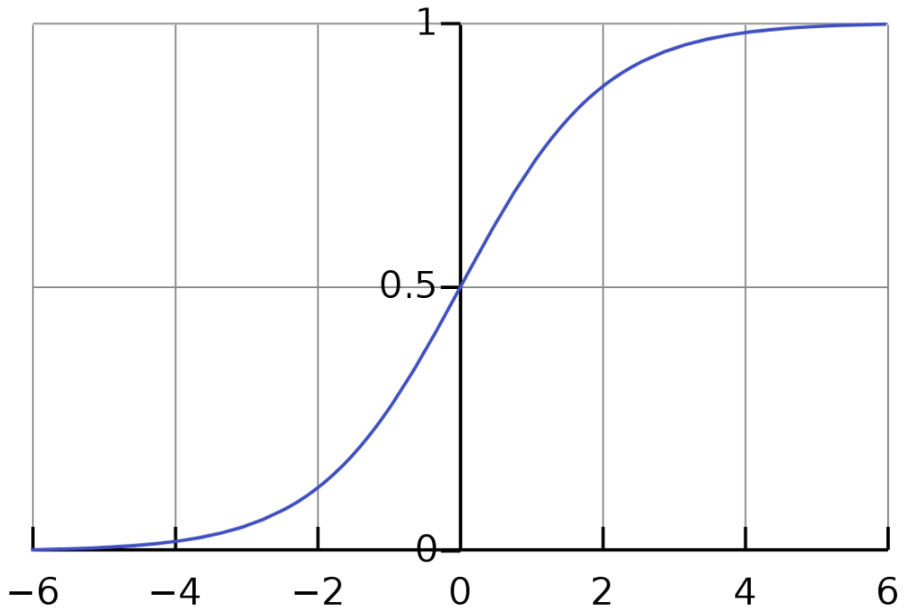


Fig. 2 — Logistic function

$$h_\theta(x) = g(\theta^T x)$$

$$z = \theta^T x$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Fig. 3— Hypothesis

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

z = np.dot(X, theta)
h = sigmoid(z)
```

Loss function

Functions have parameters/weights (represented by theta in our notation) and we want to find the best values for them. To start we pick random values and we need a way to measure how well the algorithm performs using those random weights. That measure is computed using the loss function, defined as:

$$h = g(X\theta)$$

$$J(\theta) = \frac{1}{m} \cdot \left(-y^T \log(h) - (1-y)^T \log(1-h)\right)$$

Fig. 4— Loss function

```
def loss(h, y):
    return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()
```

Gradient descent

Our goal is to minimize the loss function and the way we have to achive it is by increasing/decreasing the weights, i.e. fitting them. The question is, how do we know what parameters should be biggers and what parameters should be smallers? The answer is given by the derivative of the loss function with respect to each weight. It tells us how loss would change if we modified the parameters.

$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m} X^T \left(g(X\theta) - y\right)$$

Fig. 4 — Partial derivative

```
gradient = np.dot(X.T, (h - y)) / y.shape[0]
```

Then we update the weights by substracting to them the derivative times the learning rate.

```
lr = 0.01
theta -= lr * gradient
```

We should repeat this steps several times until we reach the optimal solution.

Predictions

By calling the sigmoid function we get the probability that some input x belongs to class 1. Let's take all probabilities ≥ 0.5 = class 1 and all probabilities < 0 = class 0. This threshold should be defined depending on the business problem we were working.

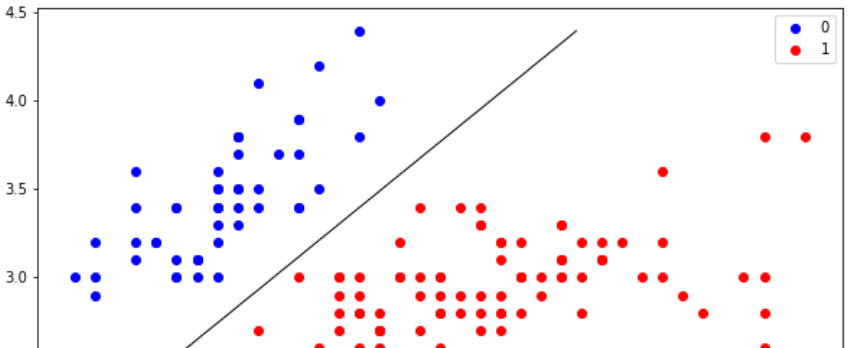
```
def predict_probs(X, theta):
    return sigmoid(np.dot(X, theta))

def predict(X, theta, threshold=0.5):
    return predict_probs(X, theta) >= threshold
```

Putting it all together

```
class LogisticRegression:
    def __init__(self, lr=0.01, num_iter=100000, fit_intercept=True,
verbose=False):
        self.lr = lr
        self.num_iter = num_iter
        self.fit_intercept = fit_intercept

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))


    def __loss(self, h, y):
        return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

    def fit(self, X, y):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        # weights initialization
        self.theta = np.zeros(X.shape[1])

        for i in range(self.num_iter):
            z = np.dot(X, self.theta)
            h = self.__sigmoid(z)
            gradient = np.dot(X.T, (h - y)) / y.size
            self.theta -= self.lr * gradient

            if(self.verbose == True and i % 10000 == 0):
                z = np.dot(X, self.theta)
                h = self.__sigmoid(z)
                print(f'loss: {self.__loss(h, y)} \t')

    def predict_prob(self, X):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        return self.__sigmoid(np.dot(X, self.theta))

    def predict(self, X, threshold):
        return self.predict_prob(X) >= threshold
```

Evaluation

```
model = LogisticRegression(lr=0.1, num_iter=300000)
%time model.fit(X, y)


CPU times: user 13.8 s, sys: 84 ms, total: 13.9 s
Wall time: 13.8 s


preds = model.predict(X)
# accuracy
(preds == y).mean()
1.0
```



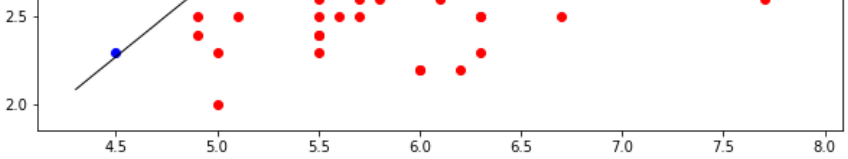


Fig. 5 — Decision boundary

Picking a learning rate = 0.1 and number of iterations = 300000 the algorithm classified all instances successfully. 13.8 seconds were needed. These are the resulting weights:

```
model.theta
array([-25.96818124,  12.56179068, -13.44549335])
```

LogisticRegression from sklearn:

```
model = LogisticRegression(C=1e20)
%time model.fit(X, y)

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 854 µs

preds = model.predict(X)
# accuracy
(preds == y).mean()
1.0

model.intercept_, model.coef_
(array([-80.62725491]), array([[ 31.61988897, -28.31500665]]))
```

If we trained our implementation with smaller learning rate and more iterations we would find approximately equal weights. But the more remarkably difference is about training time, sklearn is order of magnitude faster. Anyway, is not the intention to put this code on production, this is just a toy exercice with teaching objectives.

Further steps could be the addition of l2 regularization and multiclass classification.

Code available here.

Machine Learning | Data Science | Python | Numpy | Classification

Medium

About Write Help Legal