
The Little Machine Learner

Release 0.0.1

The Little Machine Learner team

February 18, 2010

CONTENTS

1	Logistic regression	3
1.1	1. Odds, Logit and Logistic function	3
1.2	2. Prediction	4
1.3	3. Training	5
1.4	4. Evaluation	7
1.5	5. Cross-validation	7
1.6	6. Data set	8
1.7	7. Wrapping up	9
1.8	References	9
2	Indices and tables	11

Contents:

LOGISTIC REGRESSION

Logistic regression is a statistical technique often used to predict the value taken by a categorical variable y given several predictor variables, either numerical or categorical, $\vec{x} = (x_1, \dots, x_K)$. Here we focus on the case when y is binary, i.e. $y = +1$ or $y = -1$. This often happens in medicine (diseased/healthy) and decision making (yes/no).

Let's start by importing Numpy (Numerical Python). For convenience, we alias it to np , as is common in the Numpy community.

```
import numpy as np
```

1.1 1. Odds, Logit and Logistic function

The odds in favor of an event of probability p are defined as $\frac{p}{1-p}$. The logit function is defined as $\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$. If p is a probability, the logit is thus the logarithm of the odds.

In logistic regression, the logarithm of the odds in favor of $y=+1$ given \vec{x} is modeled as a weighted sum:

$$\text{logit}(p(y = +1 | \vec{x})) = w_0 + w_1x_1 + \dots + w_Kx_K$$

The weights describe the size of the contribution of the corresponding predictor variables. A positive weight means that the variable increases the probability of the outcome, while a negative weight means that the variable decreases the probability of the outcome. A large weight means that the variable strongly influences the probability of the outcome while a near-zero weight means that the variable has little influence on the probability of the outcome.

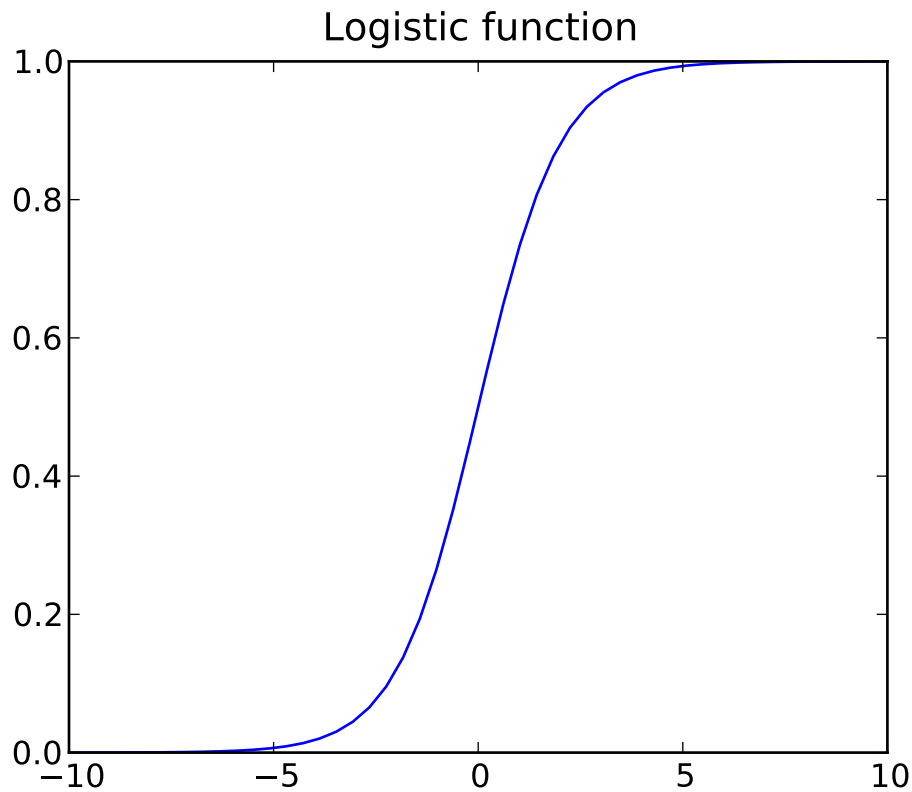
By introducing an imaginary attribute $x_0 \equiv 1$ for all instances, we can rewrite the sum as a dot product:

$$\text{logit}(p(y = +1 | \vec{x})) = \sum_{k=0}^K w_k x_k = \vec{w} \cdot \vec{x}$$

Furthermore, the logistic function is defined as:

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

It is a kind of sigmoid and maps values between -infinity and infinity to values between 0 and 1.



We implement it in the **logistic** function.

```
def logistic(z):
    return 1.0 / (1.0 + np.exp(-z))
```

The logistic function is the inverse function of the logit so

$$p(y = +1 | \vec{x}) = g(\vec{w} \cdot \vec{x})$$

Our weighted sum is therefore conveniently mapped to a value between 0 and 1, as required to make it a probability.

1.2 2. Prediction

Given the model parameters \vec{w} and a new vector \vec{x} , we can make a prediction about its class y as follows.

$$y = \begin{cases} +1, & \text{if } p(y = +1 | \vec{x}) \geq 0.5 \\ -1, & \text{otherwise} \end{cases}$$

We implement it in the **predict** function.

```
def predict(w, x):
    return logistic(np.dot(w, x)) > 0.5 or -1
```

1.3 3. Training

1.3.1 3.1. Maximum Likelihood estimation

In order to estimate the parameters $\vec{w} = (w_0, w_1, \dots, w_K)$ from a training data set $X = (\vec{x}_1, \dots, \vec{x}_N)$ and the corresponding labels $Y = (y_1, \dots, y_N)$, we can use Maximum Likelihood Estimation, i.e., finding the \vec{w} that maximizes the log-likelihood of the data.

$$L(\vec{w}) = \sum_{i=1}^N \log \begin{cases} p(y_i = +1 | \vec{x}_i) = g(\vec{w} \cdot \vec{x}_i), & \text{if } y_i = +1 \\ 1 - p(y_i = +1 | \vec{x}_i) = 1 - g(\vec{w} \cdot \vec{x}_i), & \text{if } y_i = -1 \end{cases}$$

Thankfully, the logistic function g has the nice property that $1 - g(z) = g(-z)$. By using the fact that $y=+1$ or $y=-1$, we can rewrite L more concisely.

$$L(\vec{w}) = \sum_{i=1}^N \log g(y_i \vec{w} \cdot \vec{x}_i) = \sum_{i=1}^N \log g(y_i \sum_{k=0}^K w_k x_{ik})$$

The problem with Maximum Likelihood Estimation is that it can lead to *overfitting*: when the parameter vector \vec{w} fits too well the training data and fails to generalize to new, unseen data. This can happen especially if the number of dimensions K is high or if the training data is sparse. To help with this problem, we can introduce the L2-regularized log-likelihood.

$$L_R(\vec{w}) = \sum_{i=1}^N \log g(y_i \vec{w} \cdot \vec{x}_i) - \frac{C}{2} \|\vec{w}\|^2 = \sum_{i=1}^N \log g(y_i \vec{w} \cdot \vec{x}_i) - \frac{C}{2} \sum_{k=0}^K w_k^2$$

The basic idea is to penalize large values of \vec{w} . The penalty is proportional to the squared magnitude of \vec{w} and its strength is determined by the constant C .

We implement it in the `log_likelihood` function.

```
def log_likelihood(X, Y, w, C=0.1):
    return np.sum(np.log(logistic(Y * np.dot(X, w)))) - C/2 * np.dot(w, w)
```

Note that by choosing $C=0$, we get the non-regularized log likelihood.

1.3.2 3.2. Gradient

Unfortunately, there's no closed form solution to maximizing $L(\vec{w})$ or $L_R(\vec{w})$ with respect to \vec{w} . One common approach is to use the gradient ascent, which works with the gradient (vector of partial derivatives).

$$\frac{dL}{d\vec{w}} = \left(\frac{dL}{dw_0}, \dots, \frac{dL}{dw_k} \right)$$

By using the chain rule and the fact that $\frac{d}{dz}g(z) = g(z)g(-z)$, it can be found that

$$\frac{dL}{dw_k} = \sum_{i=1}^n y_i x_{ik} g(-y_i \vec{w} \cdot \vec{x}_i)$$

Likewise,

$$\frac{dL_R}{dw_k} = \sum_{i=1}^n y_i x_{ik} g(-y_i \vec{w} \cdot \vec{x}_i) - C w_k$$

We implement it in the `log_likelihood_grad` function.

```
def log_likelihood_grad(X, Y, w, C=0.1):
    K = len(w)
    N = len(X)
    s = np.zeros(K)

    for i in range(N):
        s += Y[i] * X[i] * logistic(-Y[i] * np.dot(X[i], w))

    s -= C * w

    return s
```

1.3.3 3.3. Numerical gradient

To test whether our gradient function works, we can also compute them numerically and compare with the analytical results. For that purpose, we use the *finite central difference*. For a function f , it is defined as

$$\frac{d}{dz}f(z) \approx \frac{f(z + \epsilon) - f(z - \epsilon)}{2\epsilon}$$

where ϵ is a small enough value. The error made is in the order of $O(\epsilon^2)$.

To apply this to the gradient, we make vary one w_k at a time. This is implemented in `grad_num`. The parameter f allows to pass any log likelihood function.

```
def grad_num(X, Y, w, f, eps=0.00001):
    K = len(w)
    ident = np.identity(K)
    g = np.zeros(K)

    for i in range(K):
        g[i] += f(X, Y, w + eps * ident[i])
        g[i] -= f(X, Y, w - eps * ident[i])
        g[i] /= 2 * eps

    return g
```

The results of the analytical and numerical gradient calculations should be about the same.

```
def test_log_likelihood_grad(X, Y):
    n_attr = X.shape[1]
    w = np.array([1.0 / n_attr] * n_attr)

    print "with regularization"
    print log_likelihood_grad(X, Y, w)
    print grad_num(X, Y, w, log_likelihood)

    print "without regularization"
    print log_likelihood_grad(X, Y, w, C=0)
    print grad_num(X, Y, w, lambda X,Y,w: log_likelihood(X,Y,w,C=0))
```

1.3.4 3.4. Gradient ascent

To find \vec{w} , the `train_w` function uses the optimization package from Scipy. We use the closures `f` and `fprime` because we are maximizing the loglikelihood with respect to \vec{w} , while `X` and `Y` remain constant. The minus sign is due to the fact that Scipy performs minimization instead of maximization.

```
import scipy.optimize

def train_w(X, Y, C=0.1):
    def f(w):
        return -log_likelihood(X, Y, w, C)

    def fprime(w):
        return -log_likelihood_grad(X, Y, w, C)

    K = X.shape[1]
    initial_guess = np.zeros(K)

    return scipy.optimize.fmin_bfgs(f, initial_guess, fprime, disp=False)
```

1.4 4. Evaluation

In order to evaluate the accuracy of our model, we use a test data set and count how many predictions we get correct. This is implemented in `accuracy`.

```
def accuracy(X, Y, w):
    n_correct = 0
    for i in range(len(X)):
        if predict(w, X[i]) == Y[i]:
            n_correct += 1
    return n_correct * 1.0 / len(X)
```

1.5 5. Cross-validation

Ideally, the regularization penalty `C` should be evaluated using a third, cross-validation dataset. Since we don't have one, we use `K`-fold cross-validation.

We set aside $1/K * N$ samples from the training data. This is called held-out data. **fold** splits a vector or matrix between a held-out portion and the remaining portion. i (between 0 and $k-1$) is the index of the held-out portion.

```
def fold(arr, K, i):
    N = len(arr)
    size = np.ceil(1.0 * N / K)
    arange = np.arange(N) # all indices
    heldout = np.logical_and(i * size <= arange, arange < (i+1) * size)
    rest = np.logical_not(heldout)
    return arr[heldout], arr[rest]
```

We repeat this process K times for different positions of the held-out data in **kfold**.

```
def kfold(arr, K):
    return [fold(arr, K, i) for i in range(K)]
```

For a fixed C , we fit \vec{w} to the remaining of the data and measure the accuracy using the held-out data. **avg_accuracy** returns the average accuracy over the K folds.

```
def avg_accuracy(all_X, all_Y, C):
    s = 0
    K = len(all_X)
    for i in range(K):
        X_heldout, X_rest = all_X[i]
        Y_heldout, Y_rest = all_Y[i]
        w = train_w(X_rest, Y_rest, C)
        s += accuracy(X_heldout, Y_heldout, w)
    return s * 1.0 / K
```

It's important here to use **avg_accuracy** on the training data because the test data would lead to overly optimistic accuracy rates.

train_C simply tries several values of C and returns the one with overall best accuracy.

```
def train_C(X, Y, K=10):
    all_C = np.arange(0, 1, 0.1) # the values of C to try out
    all_X = kfold(X, K)
    all_Y = kfold(Y, K)
    all_acc = np.array([avg_accuracy(all_X, all_Y, C) for C in all_C])
    return all_C[all_acc.argmax()]
```

1.6 6. Data set

To try our logistic regression, we use the [SPECT Heart Data Set](#). Patients are classified into two categories (normal and abnormal) and are described by 22 binary attributes. The data is divided into a training set ("SPECT.train", 80 instances) and a testing set ("SPECT.test", 187 instances). 84.0% accuracy (as compared with cardiologists' diagnoses) was reported with the CLIP3 algorithm.

The file is in a comma-separated CSV format. The first column is the overall diagnosis y .

```
1,0,0,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0
1,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,0,0,0,0,0,1
1,1,0,1,0,1,0,0,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0
[... ]
0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Note that negatives are marked by 0.

The `read_data` function reads a data file and outputs a $N \times K$ matrix $X = (\vec{x}_1, \dots, \vec{x}_N)$ (rows are patients, columns are attributes) and a N -vector of the corresponding diagnoses (class labels) $Y = (y_1, \dots, y_N)$.

```
def read_data(filename, sep=",", filt=int):

    def split_line(line):
        return line.split(sep)

    def apply_filt(values):
        return map(filt, values)

    def process_line(line):
        return apply_filt(split_line(line))

    f = open(filename)
    lines = map(process_line, f.readlines())
    # "[1]" below corresponds to x0
    X = np.array([[1] + l[1:] for l in lines])
    # "or -1" converts 0 values to -1
    Y = np.array([l[0] or -1 for l in lines])
    f.close()

    return X, Y
```

1.7 7. Wrapping up

Once we now C , as the last training step, we can fit \vec{w} to the entire training set. This is what we do in the `main` function.

```
def main():
    X_train, Y_train = read_data("logreg/spect/SPECT.train")

    # Uncomment the line below to check the gradient calculations
    #test_log_likelihood_grad(X_train, Y_train); exit()

    C = train_C(X_train, Y_train)
    print "C was", C
    w = train_w(X_train, Y_train, C)
    print "w was", w

    X_test, Y_test = read_data("logreg/spect/SPECT.test")
    print "accuracy was", accuracy(X_test, Y_test, w)

if __name__ == "__main__": main()
```

We got 73% predictions correct.

1.8 References

Logistic regression, Jason Rennie

Generative and Discriminative Classifiers: Naive Bayes and Logistic Regression, Tom Mitchell

Cross validation, Regularization and Information Criteria, Michael I. Jordan

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*