Neural Networks

Matt R

February 21, 2022

Contents

1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose we have we have training examples $x \in \mathbb{R}^{m \times n}$ with binary labels $y \in \{0,1\}^{1 \times n}$. We desire to train a model which yields an output a which represents

$$a = \mathbb{P}(y = 1|x).$$

To this end, let $\sigma : \mathbb{R} \to (0,1)$ denote the sigmoid function, i.e.,

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

and let $w \in \mathbb{R}^m$, $b \in \mathbb{R}$, and let

$$a = \sigma(w^T x + b).$$

To analyze the accuracy of model, we need a way to compare y and a, and ideally this functional comparison can be optimized with respect to (w, b) in such a way to minimize the error. To this end, we note that

$$\mathbb{P}(y|x) = a^y (1 - a)^{1-y},$$

or rather

$$\mathbb{P}(y = 1|x) = a, \qquad \mathbb{P}(y = 0|x) = 1 - a,$$

so $\mathbb{P}(y|x)$ represents the corrected probability. Now since we want

$$a \approx 1 \quad \text{when } y = 1,$$

and

$$a \approx 0 \quad \text{when } y = 0,$$

and $0 \leq a \leq 1$, any error using differences won't be refined enough to analyze when tuning the model. Moreover, since introducing the sigmoid function, our usual mean-squared-error function won't be convex. This leads us to apply the log function, which when restricted to $(0, 1)$ is a bijective mapping of $(0, 1) \to (-\infty, 0)$. This leads us to define our log-loss function

$$\begin{aligned}
\mathbb{L}(a, y) &= -\log(\mathbb{P}(y|x)) \\
&= -\log\left(a^y (1 - a)^{1-y}\right) \\
&= -\left[y \log(a) + (1 - y) \log(1 - a)\right],
\end{aligned}$$

and finally, since we wish to analyze how our model performs on the entire training set, we need to average our log-loss functions to obtain our cost function $\mathbb{J}$ defined by

$$\begin{aligned}
\mathbb{J}(w,b) &= \frac{1}{n}\sum_{j=1}^{n}\mathbb{L}(a_j,y_j) \\
&= -\frac{1}{n}\sum_{j=1}^{n}\left[y_j\log(a_j)+(1-y_j)\log(1-a_j)\right] \\
&= -\frac{1}{n}\sum_{j=1}^{n}\left[y_j\log(\sigma(w^Tx_j+b))+(1-y_j)\log(1-\sigma(w^Tx_j+b))\right].
\end{aligned}$$

1.1 The Gradient

To compute the gradient of our cost function $\mathbb{J}$, we first write $\mathbb{J}$ as a sum of compositions as follows: We have the log-loss function considered as a map $\mathbb{L} : (0,1)\times\mathbb{R}\to\mathbb{R}$,

$$\mathbb{L}(a,y) = -\left[y\log(a)+(1-y)\log(1-a)\right],$$

we have the sigmoid function $\sigma : \mathbb{R}\to(0,1)$ with $\sigma(z)=a$ and $\sigma'(z)=a(1-a)$, and we have the collection of affine-functionals $\phi_x : \mathbb{R}^m\times\mathbb{R}\to\mathbb{R}$ given by

$$\phi_x(w,b) = w^Tx+b,$$

for which we fix an arbitrary $x\in\mathbb{R}^m$ and write $\phi=\phi_x$, and set $z=\phi(w,b)$. Finally, we introduce the auxiliary function $\mathcal{L} : \mathbb{R}^m\times\mathbb{R}\to\mathbb{R}$ given by

$$\mathcal{L}(w,b) = \mathbb{L}(\sigma(\phi(w,b)),y).$$

Then by the chain rule, we have that

$$\begin{aligned}
d\mathcal{L} &= d_a\mathbb{L}(a,y)\circ d\sigma(z)\circ d_w\phi(w,b) \\
&= \left[-\frac{y}{a}+\frac{1-y}{1-a}\right]\cdot a(1-a)\cdot\begin{bmatrix}x^T & 1\end{bmatrix} \\
&= \left[-y(1-a)+a(1-y)\right]\cdot\begin{bmatrix}x^T & 1\end{bmatrix} \\
&= (a-y)\begin{bmatrix}x^T & 1\end{bmatrix}
\end{aligned}$$

Composition turns into matrix multiplication in the tangent space.

Moreover, since in Euclidean space, we have that $\nabla f = (df)^T$, and hence that

$$\nabla \mathcal{L}(w, b) = (a - y) \begin{bmatrix} x \\ 1 \end{bmatrix},$$

or rather

$$\partial_w \mathbb{L}(a, y) = (a - y)x, \qquad \partial_b \mathbb{L}(a, y) = a - y.$$

Finally, since our cost function $\mathbb{J}$ is the sum-log-loss, we have by linearity that

$$\begin{aligned}
\partial_w \mathbb{J}(w, b) &= \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j) x_j \\
&= \frac{1}{n} ((a - y) \cdot x^T)^T \\
&= \frac{1}{n} x \cdot (a - y)^T
\end{aligned}$$

and

$$\partial_b \mathbb{J}(w, b) = \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j).$$

1.1.1 Vectorization in Python

Here we include the general code to train a model using logistic regression without regularization and without tuning on a cross-validation set.

```
import copy

import numpy as np

def sigmoid(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    sigma : array_like
    """

    sigma = (1 / (1 + np.exp(-z)))
    return sigma

```

```python
def cost_function(x, y, w, b):
    """
    Parameters
    -----------
    x : array_like
        x.shape = (m, n) with m-features and n-examples
    y : array_like
        y.shape = (1, n)
    w : array_like
        w.shape = (m, 1)
    b : float

    Returns
    -------
    J : float
        The value of the cost function evaluated at (w, b)
    dw : array_like
        dw.shape = w.shape = (m, 1)
        The gradient of J with respect to w
    db : float
        The partial derivative of J with respect to b
    """

    # Auxiliary assignments
    m, n = x.shape
    z = w.T @ x + b
    assert z.size == n
    a = sigmoid(z).reshape(1, n)
    dz = a - y

    # Compute cost J
    J = (-1 / n) * (np.log(a) @ y.T + np.log(1 - a) @ (1 - y).T)

    # Compute dw and db
    dw = (x @ dz.T) / m
    assert dw.shape == w.shape
    db = np.sum(dz) / m

    return J, dw, db

def grad_descent(x, y, w, b, alpha=0.001, num_iters=2000, print_cost=False):
    """
    Parameters
    ----------
    x, y, w, b : See cost_function above for specifics.
        w and b are chosen to initialize the descent (likely all components 0)
    alpha : float
```

```
        The learning rate of gradient descent
    num_iters : int
        The number of times we wish to perform gradient descent

    Returns
    -------
    costs : List[float]
        For each iteration we record the cost-values associated to (w, b)
    params : Dict[w : array_like, b : float]
        w : array_like
            Optimized weight parameter w after iterating through grad descent
        b : float
            Optimized bias parameter b after iterating through grad descent
    grads : Dict[dw : array_like, db : float]
        dw : array_like
            The optimized gradient with repsect to w
        db : float
            The optimized derivative with respect to b
    """

    costs = []
    w = copy.deepcopy(w)
    b = copy.deepcopy(b)
    for i in range(num_iters):
        J, dw, db = cost_function(x, y, w, b)
        w = w - alpha * dw
        b = b - alpha * db

        if i % 100 == 0:
            costs.append(J)
            if print_cost:
                idx = int(i / 100) - 1
                print(f'Cost␣after␣iteration␣{i}:␣{costs[idx]}')

    params = {'w' : w, 'b' : b}
    grads = {'dw' : dw, 'db' : db}

    return costs, params, grads

def predict(w, b, x):
    """
    Parameters
    ----------
    w : array_like
        w.shape = (m, 1)
    b : float
    x : array_like
```

```
        x.shape = (m, n)

    Returns
    -------
    y_predict : array_like
        y_pred.shape = (1, n)
        An array containing the prediction of our model applied to training
        data x, i.e., y_pred = 1 or y_pred = 0.
    """

    m, n = x.shape
    # Get probability array
    a = sigmoid(w.T @ x + b)
    # Get boolean array with False given by a < 0.5
    pseudo_predict = ~(a < 0.5)
    # Convert to binary to get predictions
    y_predict = pseudo_predict.astype(int)

    return y_predict

def model(x_train, y_train, x_test, y_test, alpha=0.001, num_iters=2000, accuracy=Tr
    """
    Parameters:
    -----------
    x_train, y_train, x_test, y_test : array_like
        x_train.shape = (m, n_train)
        y_train.shape = (1, n_train)
        x_test.shape = (m, n_test)
        y_test.shape = (1, n_test)
    alpha : float
        The learning rate for gradient descent
    num_iters : int
        The number of times we wish to perform gradient descent
    accuracy : Boolean
        Use True to print the accuracy of the model

    Returns:
    d : Dict
        d['costs'] : array_like
            The costs evaluated every 100 iterations
        d['y_train_preds'] : array_like
            Predicted values on the training set
        d['y_test_preds'] : array_like
            Predicted values on the test set
        d['w'] : array_like
            Optimized parameter w
        d['b'] : float
```

```python
            Optimized parameter b
        d['learning_rate'] : float
            The learning rate alpha
        d['num_iters'] : int
            The number of iterations with which gradient descent was performed

    """

    m = x_train.shape[0]
    # initialize parameters
    w = np.zeros((m, 1))
    b = 0.0
    # optimize parameters
    costs, params, grads = grad_descent(x_train, y_train, w, b, alpha, num_iters)
    w = params['w']
    b = params['b']
    # record predictions
    y_train_preds = predict(w, b, x_train)
    y_test_preds = predict(w, b, x_test)
    # group results into dictionary for return
    d = {'costs' : costs,
         'y_train_preds' : y_train_preds,
         'y_test_preds' : y_test_preds,
         'w' : w,
         'b' : b,
         'learning_rate' : alpha,
         'num_iters' : num_iters}

    if accuracy:
        train_acc = 100 - np.mean(np.abs(y_train_preds - y_train)) * 100
        test_acc = 100 - np.mean(np.abs(y_test_preds - y_test)) * 100
        print(f'Training Accuracy: {train_acc}%')
        print(f'Test Accuracy: {test_acc}%')


    return d
```