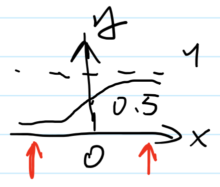
Week2 - Logistic Regression with Gradient Descent

Donnerstag, 16. Juli 2020 19:32

Logistic Regression:

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



$x \to -\infty \quad y = 0$
$x \to +\infty \quad y = 1$ } Binary Classification problem

$\hat{y} = \sigma(w^T x + b)$ with $w^T \in \mathbb{R}^n$, $b \in \mathbb{R}$

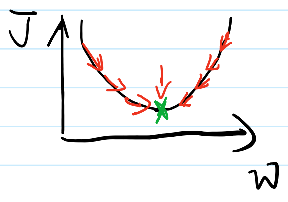
$\hat{y}$ predicted value

y ground true

loss / error $\to$ $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$

cost function $\to$ $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$

Gradient descent:



Repeat {

$w := w - \alpha \underbrace{\frac{dJ(w)}{dw}}_{\text{gradient}}$

}

$\alpha \to$ learning rate

when learning rate too large, fast but may never converge;
when learning rate too small, will converge however slow.

Vectorization:

Vectorization can greatly increase the process and simplify the algorithms.

$X = \left[\underbrace{x^{(1)}}_{n} \; \underbrace{x^{(2)}}_{n} \; x^{(3)} \; \cdots \; x^{(m)} \right] \quad x \in \mathbb{R}^{n \times m}$

$z = [z^{(1)} \; z^{(2)} \cdots z^{(m)}] = \underbrace{w^T x}_{} + \overbrace{[b \; b \; b \cdots b]}^{1\times m}$

$z = \text{np.dot}(w^T, x) + b \to$ broadcasting to $1\times m$

$A = [a^{(1)} \; a^{(2)} \; a^{(3)} \cdots a^{(m)}] = \sigma(z) \quad z \to (1\times m)$

$Y = [y^{(1)} \; y^{(2)} \cdots y^{(m)}]$

$dz = A - Y = [a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \cdots a^{(m)} - y^{(m)}] \to 1\times m$

$$\begin{cases} dw = \frac{1}{m} X dz^T & X \to (n \times m) \quad dz^T \to (m \times 1) \Rightarrow (n \times 1) \\ db = \frac{1}{m} np.\text{sum}(dz) & (1 \times 1) \end{cases}$$

Take out from programming assignment:

-- Get familiar with Numpy --

sigmoid function : $x \to$ numpy array

$s = \text{sigmoid}(x)$

$= 1/(1+ np.exp(x))$

sigmoid gradient : $\sigma'(x) = \sigma(x)(1-\sigma(x))$

An image is represented by a 3D array of shape (length, height, depth = 3); Yet, when you read an image as the input of an algorithm you convert it to a vector of shape (length*height*3, 1), so to speak a column vector, 1D vector.

```
v = image.reshape((image.shape[2]*image.shape[0]*image.shape[1], 1))
```

with image -- a numpy array of shape (length, height, depth)

Normalizing input data helps algorithms to converge faster: dividing each row vector of x by its norm.

```
x_norm = np.linalg.norm(x, ord = 2, axis = 1, keepdims = True)
x = x / x_norm
```

Matrix Operation:

`dot product: np.dot(x1,x2) result is scale`

`outer product:  np.outer(x1,x2)`

`Elementwise Multiplication: np.multiply(x1,x2)  or x1 * x2`

`Matrix Multiplication: x1 @ x2`

--- Logistic Regression with a Neural Network mindset--

The first step to a classifier is commonly data preprocessing.

1. Check the dimensions and shapes of the datasets, including training dataset, test dataset, etc.#
2. The dataset should normally has a shape of ((num_of_pixel * num_of_pixel * 3, 1))
3. In case of images, reshape the datasets into vector of size (height * width * 3 , 1) may be helpful
   ```
   -> X_flatten = X.reshape(X.shape[0], -1).T
   Or X_flatten = X.reshapt(X.shape[0] * X.shape[1] * 3,  1)
   ```
4. To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255. One common preprocessing step is to center and standardize the dataset, meaning:
   a. Substracting the mean of the whole numpy array from each example, and then divide each example by the

standard deviation of the whole numpy array.

b. **For image input, substracting each pixel value by 255.**

After preprocessing of the datasets, the logistic regression itself is implemented in:

$$z^{(i)} = w^T x^{(i)} + b$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -\underline{y^{(i)}\log(a^{(i)})} - (1 - \underline{y^{(i)})\log(1} - a^{(i)}) \xrightarrow{} \text{elementwise}$$

$$J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

General structure of a NN algorithms:

- Define the model structure (such as number of input features, hidden layers, etc.)
- Initialize the model's parameters
- Loop(vertorized):
  - Calculate current loss (forward propagation)
  - Calculate current gradient (backward propagation)
  - Update parameters (gradient descent)

The previous steps can be implemented in different functions and then called in one model()

Steps:

1. initialize parameters (w, b)
   `-> w = np.zeros((dim,1)); b = 0`
2. Forward propagation:
   a. sigmoid() or other transfer function
   b. activation A = sigmoid(w*X + b)
   c. cost function L
3. Back propagation:
   a. dw = dL/dw
   b. db = dL/db
4. Optimizing (w, b):
   a. for i in range(number of iterations)
      i. w = w - learning_rate * dw
      ii. b = b - learning_rate * db

```
def propagation(w, b, X, Y):
    '''
    Implement cost function and gradients

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1,
number of examples)
    '''
    m = X.shape[1];  # number of examples

    # Forward Prog
    A = sigmoid(np.dot(w.T, X) + b)  # (1*m)
    cost =  -1/m * np.sum(Y * np.log(A) + (1-Y) * np.log(1-A))

    # Back Prog
    dw = 1/m * np.dot(X, (A-Y).T)   # (n*1)
    db = 1/m * np.sum(A-Y)  # (1*1)
    grads = {"dw" : dw, "db" : db}

    Return grads, cost
```

$$\xrightarrow{} dw = \frac{1}{m} X dz^T$$

```
def optimize(w, b, X, Y, num_iterations, learning_rate):
    '''
    Gradient descent

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with respect to
    the cost function
    costs -- list of all the costs computed during the optimization, this will be used
    to plot the learning curve.

    '''
    Cost = []
    for i in range(num_iterations):

            # Cost and gradient calculation
            grads, cost = propagate(w, b, X, Y)

            # derivatives from grads
            dw = grads["dw"]
            db = grads["db"]

            # update
            w = w - learning_rate * dw
            b = b - learning_rate * db

            # Record the costs every 100 iterations and print on screen
            if i % 100 == 0:
                costs.append(cost)
                print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w, "b": b}
    grads = {"dw": dw,  "db": db}
    return params, grads, costs
```

5. Predict using optimized (w, b)
   a. if A = sigmoid(w*X + b) > 0.5:
      i. Y_predict = 1
   b. else:
      i. y_predic = 0
6. integrate all previous steps into a learning model

```
def predict(w, b, X):
    '''
    Predict using learned (w,b)
    '''
     m = X.shape[1]
     Y_prediction = np.zeros((1,m))

     m = X.shape[1]
     Y_prediction = np.zeros((1,m))
     w = w.reshape(X.shape[0], 1)

    for i in range(A.shape[1]):

            # Convert probabilities A[0,i] to actual predictions
    p[0,i]
            if A[:, i] > 0.5:
                Y_prediction[:, i] = 1
            else:
                Y_prediction[:, i] = 0

    Return grads, cost
```

```
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5):
    # initialize parameters with zeros
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, 
learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - 
Y_train)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - 
Y_test)) * 100))

    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test, 
         "Y_prediction_train" : Y_prediction_train, 
         "w" : w, 
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

7. By plot d["cost"] against iterations we get the learning curve
   a. Different learning rates give different costs and thus different predictions results.
   b. If the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge.
   c. In deep learning recommended:
      - Choose the learning rate that better minimizes the cost function.
      - If your model overfits, use other techniques to reduce overfitting.

Note: This is the implementation of a logistic regression problem with gradient decent approach. The Forward / Back propagation involves Neural network thinking and will be looked into detail in the further assignments.