# Regularization to Reduce Overfitting

You will:

- Understand what is overfit and how to reduce it.
- Understand the concept of regularization and how it is used to reduce overfit.
- Extend the previous linear and logistic cost function with a regularization term.
- Extend the previous linear and logistic gradients with a regularization term added.

## Overfit

**To understand overfitting let's say:**

Suppose we have a training set with $m$ examples and $n$ features, we want to predict the output in weither linear or logistic prediction.

1. If our predictions from our model ***does not*** fit the training set ***well***:
    - Here we will say our model is **Underfitting** the data or has **high bias**.
2. If our predictios from our model ***fits*** the training set ***just well***.
    - Here we will say our model is **Generalized**.
3. If our predictions from our model ***fits*** the training set ***extremely well***.
    - Here we will say our model is ***Overfit*** the data or has ***high variance***

**How to address Overfitting**

1. Collect more training data.
    - If you can get more dataset, you can add them to the training set.
2. Reduce number of features, $n$.
    - Perform **Feature selection**.
        - If you have many features $n$ but fewer number of examples $m$, the good solution to reduce **overfitting** is by reducing number of features $n$.
3. Perform **Regularization**.
    - This is very usefull technic for training models.

**The Idea Behind Regularization**

- Let say we have $n$ number of features, $n = 100$.
    - $w_0, w_1, w_2, \cdots w_{99},\ b$ - parameters

    If $w_0 \ \cdots \ w_{99},\ b$ will be **smaller**

    Then

    Our model will be equivalent to a **Simpler model**

    Therefore: our model will be **less** likely to **overfit**

- So, to get **smaller** $w_0, w_1, w_2, \cdots, w_{99},\ b$

  We will pinalize all $w_j$ by adding $\frac{\lambda}{2m} \sum_{n=0}^{n-1} w_j^2$

  Where

  $\lambda \ \text{ is regularization parameter}, \lambda > 0$

**Note:**

- Cost.
    - Cost functions differ significantly between Linear and Logistic Regression, but adding Regularization to the equations is the same.
- Gradient.
    - The gradient functions for Linear and Logistic Regression are very similar. They differ only in the implementation of $f_{w,b}$

Logistic Regularization

**Logistic Regressiom**

$$f_{w,b}(\mathbf{x}^{(i)}) = g(w_j \cdot \mathbf{x}_j^{(i)} + b)$$

$$J(w,b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{w,b}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\quad w_j = w_j - \alpha \frac{\partial J(w,b)}{\partial w_j} \quad \text{for j} = 0, \ldots, \text{n-1} \\ &\quad b = b - \alpha \frac{\partial J(w,b)}{\partial b} \\ &\} \end{aligned}$$

*Where:*

$$z = w_j \cdot \mathbf{x}_j^{(i)} + b$$

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

$$f_{w,b}(\mathbf{x}^{(i)}) = g(z)$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

**Regularized Logistic Regression**

$$f_{w,b}(\mathbf{x}^{(i)}) = g(w_j \cdot \mathbf{x}_j^{(i)} + b)$$

$$J(w,b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{w,b}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$$

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\quad w_j = w_j - \alpha \frac{\partial J(w,b)}{\partial w_j} \quad \text{for j} = 0, \ldots, \text{n-1} \\ &\quad b = b - \alpha \frac{\partial J(w,b)}{\partial b} \\ &\} \end{aligned}$$

*where*

$$z = w_j \cdot \mathbf{x}_j^{(i)} + b$$

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

$$f_{w,b}(\mathbf{x}^{(i)}) = g(z)$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \sum_{j=0}^{n-1} w^2$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

**Sigmoid function**

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

**Python Implementation**

```python
def sigmoid(z):
    """
    Copute sigmoid
    Arg:
        z (scalar): prediction

    Return:
        logistic prediction
    """
    return 1/(1 + np.exp(-z))
```

✓ 0.0s Python

**C Implementation**

```c
double sigmoid(double z){

    /*
    Compute sigmoid
    Arg:
        z (scalar):  prediction
    Return:
        logistic prediction
    */

    return 1/(1 + exp(-z));
}
```

**Regularized Cost Logistic Function**

$J(w,b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{w,b}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$

$f_{w,b}(\mathbf{x}^{(i)}) = g(z)$

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

**Python Implementation**

```python
def compute_cost_logistic_reg(X, y, w, b, lambda_):
    """
    Computes the cost over all examples
    Args:
        X (ndarray (m,n): Data, m examples with n features
        y (ndarray (m,)): target values
        w (ndarray (n,)): model parameters
        b (scalar)      : model parameter
        lambda_ (scalar): Controls amount of regularization
    Returns:
        total_cost (scalar):  cost
    """

    m,n=X.shape
    cost=0

    for i in range(m):
        f_wb = np.dot(X[i],w) + b
        f_wb_i = sigmoid(f_wb)
        cost += -y[i]*np.log(f_wb_i) - (1-y[i])*(np.log(1-f_wb_i))
    cost /=m

    reg_cost=0
    for i in range(n):
        reg_cost += w[i]**2
    reg_cost =(lambda_/(2*m)) * reg_cost

    total_cost = cost + reg_cost
    return total_cost
```

✓ 0.0s Python

## Regularized Gradient Logistic Function

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \sum_{j=0}^{n-1} w^2$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

$$f_{w,b}(\mathbf{x}^{(i)}) = g(z)$$

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

## Python Implementation

```python
def compute_gradient_logistic_reg(X, y, w, b, lambda_):
    """
    Computes the gradient for linear regression

    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
      lambda_ (scalar): Controls amount of regularization
    Returns
      dj_dw (ndarray Shape (n,)): The gradient of the cost w.r.t. w.
      dj_db (scalar)            : The gradient of the cost w.r.t. b.
    """

    m,n = X.shape
    dj_dw = np.zeros((n,))
    dj_db = 0.0

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)
        err_i = f_wb_i - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + (err_i*X[i][j])
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m
    dj_db = dj_db/m

    for j in range(n):
        dj_dw[j] = dj_dw[j] + (lambda_/m)*w[j]

    return dj_db, dj_dw
```

✓ 0.0s Python

## C Implementation

```c
struct reg_grads compute_gradient_logistic_reg(float X[][n], int y[], double w[][1], double b, float lambda_){
    /*
    Compute the gradient for linear regression
    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
      lambda_ (scalar): Controls amount of regularization

    Returns:
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameters w.
      dj_db (scalar):       The gradient of the cost w.r.t. the parameter b.
    */

    struct reg_grads grads;

    grads.dj_db=0;
    grads.dj_dw = (double *)calloc(n, sizeof(double));
    double f_wx;
    double f_wb;
    double *err = (double *)calloc(m, sizeof(double));

    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < 1; j++)
        {
            f_wx=0;
            f_wb=0;
            for (int k = 0; k < n; k++)
            {
                f_wx += X[i][k]*w[k][j];
            }
            f_wb = sigmoid(f_wx + b);

        }
        err[i] = (f_wb - y[i]);

        grads.dj_db += err[i];

    }

    grads.dj_db = grads.dj_db / m;

    for (int j = 0; j < n; j++)
    {
        for (int i = 0; i < m; i++)
        {
            grads.dj_dw[j] += (err[i]*X[i][j]);
        }

        grads.dj_dw[j] /=m;

        grads.dj_dw[j] += (lambda_/m)*w[j][0];

    }


    return grads;
}
```

**Python main function**

```python
import numpy as np

def main():
    X=np.array([[4.17022005e-01, 7.20324493e-01, 1.14374817e-04],
                [1.86260211e-01, 3.45560727e-01, 3.96767474e-01],
                [2.04452250e-01, 8.78117436e-01, 2.73875932e-02],
                [1.40386939e-01, 1.98101489e-01, 8.00744569e-01],
                [8.76389152e-01, 8.94606664e-01, 8.50442114e-02]])
    y = np.array([0,1,0,1,0])
    w = np.array([-0.40165317, -0.07889237,  0.45788953])
    b = 0.5
    lambda_ = 0.7
    dj_db, dj_dw =  compute_gradient_logistic_reg(X, y, w, b, lambda_)

    m,n = X.shape

    print("Regularized Parameters: \n")
    for i in range(n):
        print(f"dj_dw[{i}]:\t{dj_dw[i]}")

    print(f"dj_db: {dj_db}", )

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

✓ 0.0s Python

```
Regularized Parameters:

dj_dw[0]:	0.08590186822598606
dj_dw[1]:	0.2318715168852964
dj_dw[2]:	-0.0019798089387517287
dj_db: 0.20333487598147518
```

**C main function**

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#define m 5
#define n 3

struct reg_grads{
    double *dj_dw;
    double dj_db;
};


struct reg_grads compute_gradient_logistic_reg(float X[][n], int y[], double w[][1], double b, float lambda_);
double sigmoid(double z);

int main(){
    float X[][n]={{4.17022005e-01, 7.20324493e-01, 1.14374817e-04},
                  {1.86260211e-01, 3.45560727e-01, 3.96767474e-01},
                  {2.04452250e-01, 8.78117436e-01, 2.73875932e-02},
                  {1.40386939e-01, 1.98101489e-01, 8.00744569e-01},
                  {8.76389152e-01, 8.94606664e-01, 8.50442114e-02}};

    double w[][1] = {-0.40165317, -0.07889237,  0.45788953};

    int y[] = {0, 1, 0, 1, 0};

    double b = 0.5;
    float lambda_ = 0.7;

    struct  reg_grads grads;

    grads = compute_gradient_logistic_reg(X, y, w, b, lambda_);

    printf("Regularized gradient:\n\n");
    for (int i = 0; i < n; i++)
    {
        printf("dj_dw[%d]: %f\n", i, grads.dj_dw[i]);
    }
    printf("\n");
    printf("dj_db: %f\n", grads.dj_db);

    return 0;
}
```

```
venvsuzanodero@suzans-MacBook-Air overfitting % gcc overfitting.c
venvsuzanodero@suzans-MacBook-Air overfitting % ./a.out
Regularized gradient:

dj_dw[0]: 0.085902
dj_dw[1]: 0.231872
dj_dw[2]: -0.001980

dj_db: 0.203335
```