Informe Laboratorio 4 IA

Implementación

La regularización será implementada en el código en los apartados que sean necesarios, desde la función de costo hasta el descenso por el gradiente de todas las funciones utilizadas en el código. Además se la usará también en la ecuación de la normalización como una medida de comprobante en su impacto con los ejercicios hechos

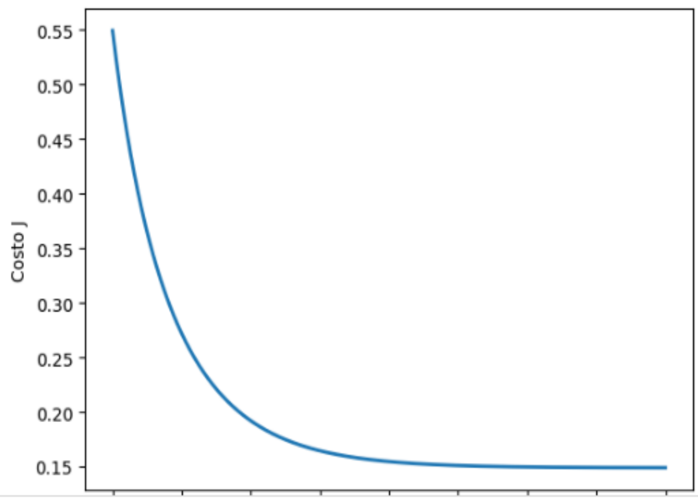
REGRESION LAB1

```
[ ] # regresión lineal multivariable
    def computeCostMulti(X, y, theta):
        # Inicializa algunos valores utiles
        m = y.shape[0] # numero de ejemplos de entrenamiento

        J = 0

        h = np.dot(X, theta)

        J = (1/(2 * m)) * np.sum(np.square(np.dot(X, theta) - y))

        return J
```

```
[ ] # implementación del algoritmo de descenso de gradiente
    def gradientDescentMulti(X, y, theta, alpha, num_iters):

        # Inicializa algunos valores
        m = y.shape[0] # numero de ejemplos de entrenamiento

        # realiza una copia de theta, el cual será acutalizada por el descenso por el gradiente
        theta = theta.copy()

        J_history = []

        for i in range(num_iters):
            theta = theta - (alpha / m) * (np.dot(X, theta) - y).dot(X)
            J_history.append(computeCostMulti(X, y, theta))

        return theta, J_history
```

```
[ ] # Elegir algun valor para alpha (probar varias alternativas)
    alpha = 0.001 # si es grande se resta más a los thetas, si es menor alfa va cambiando poco a poco
    #se necesitan muchas iteraciones, si es grande el alfa salta mucho y no llega facilmente
    # mejores alfas = 0.1, 0.01, 0.001, 0.0001, 0.0005
    num_iters = 4000

    # inicializa theta y ejecuta el descenso por el gradiente
    theta = np.zeros(9)
    theta, J_history = gradientDescentMulti(X, y, theta, alpha, num_iters)

    # Grafica la convergencia del costo
    pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
    pyplot.xlabel('Numero de iteraciones')
    pyplot.ylabel('Costo J')

    # Muestra los resultados del descenso por el gradiente
    print('theta calculado por el descenso por el gradiente: {:s}'.format(str(theta)))

    # Estimar el nivel con distintas caracteristicas
    X_array = [1, 487, 1, 1, 28.10, 7.0, 0.48, 12, 4]
    X_array[1:9] = (X_array[1:13] - mu) / sigma
    level = np.dot(X_array, theta)   # Se debe cambiar esto

    print('El nivel predecido es (usando el descenso por el gradiente): {:.4f}'.format(level))
```

```
theta calculado por el descenso por el gradiente: [ 0.71537533  0.00234552 -0.07767502  0.01274515  0.31070382 -0.04608345
  0.23425388 -0.00609362 -0.0801392 ]
El valor predecido es (usando el descenso por el gradiente): 1.5452
```



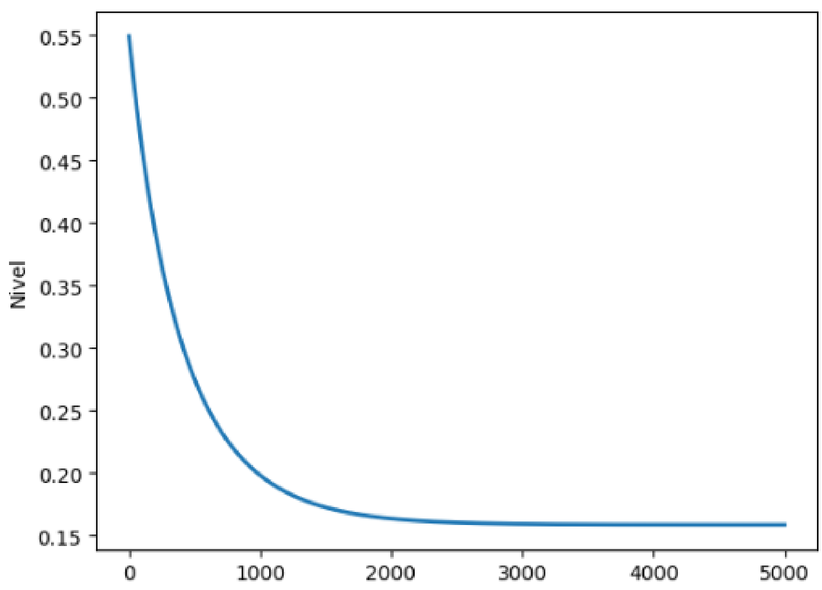
REGRESION LAB 1 CON REGULARIZACION

```
[24] # Costo y Decenso por el gradiente con regularización
     def computeCostMulti(X, y, theta, lambda_):
         # Inicializa algunos valores utiles
         m = y.shape[0] # numero de ejemplos de entrenamiento

         J = 0

         h = np.dot(X, theta)

         J = (1/(2 * m)) * (np.sum(np.square(h - y)) + lambda_ * np.sum(np.square(theta)))

         return J
```

```
[25] def gradientDescentMulti(X, y, theta, alpha, num_iters, lambda_):

         # Inicializa algunos valores
         m = y.shape[0] # numero de ejemplos de entrenamiento

         # realiza una copia de theta, el cual será acutalizada por el descenso por el gradiente
         theta = theta.copy()

         J_history = []

         for i in range(num_iters):
             theta = theta - (alpha / m) * ((np.dot(X, theta) - y).dot(X) + lambda_ * theta)
             J_history.append(computeCostMulti(X, y, theta, lambda_))

         return theta, J_history
```

```
[26] # Elegir algun valor para alpha (probar varias alternativas)
alpha = 0.001 # si es grande se resta más a los thetas, si es menor alfa va cambiando poco a poco
#se necesitan muchas iteraciones, si es grande el alfa salta mucho y no llega facilmente
# mejores alfas = 0.1, 0.01, 0.001, 0.0001, 0.0005
num_iters = 5000
# inicializa theta y ejecuta el descenso por el gradiente
theta = np.zeros(9)
lambda_ = 1000
theta, J_history = gradientDescentMulti(X, y, theta, alpha, num_iters, lambda_)

# Grafica la convergencia del costo
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Numero de iteraciones')
pyplot.ylabel('Nivel')

# Muestra los resultados del descenso por el gradiente
print('theta calculado por el descenso por el gradiente: {:s}'.format(str(theta)))

# Estimar el nivel con distintas caracteristicas
X_array = [1, 480, 1, 1, 27.10, 7.0, 0.48, 12, 4]
X_array[1:9] = (X_array[1:13] - mu) / sigma
level = np.dot(X_array, theta)   # Se debe cambiar esto

print('El nivel predecido es (usando el descenso por el gradiente): {:.4f}'.format(level))
```

```
theta calculado por el descenso por el gradiente: [ 0.70476734  0.00240342 -0.07621287  0.01183422  0.30664466 -0.044419
  0.23120999 -0.00593125 -0.07939521]
El nivel predecido es (usando el descenso por el gradiente): 1.4899
```



REGRESION LAB2

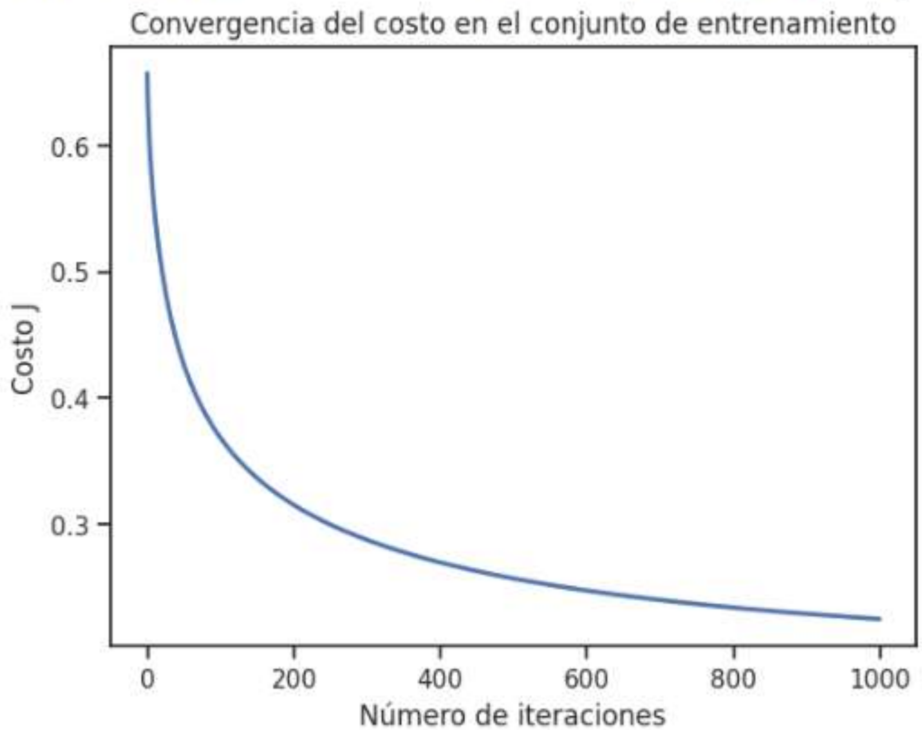
```
[ ] def calcularCosto(theta, X, y):
        # Inicializar algunos valores utiles
        m = y.size  # numero de ejemplos de entrenamiento

        J = 0
        h = calcularSigmoide(X.dot(theta.T))
        J = (1 / m) * np.sum(-y.dot(np.log(h)) - (1 - y).dot(np.log(1 - h)))

        return J
```

Funcion de Descenso por el Gradiente

```
[ ] def descensoGradiente(theta, X, y, alpha, num_iters):
        # Inicializa algunos valores
        m = y.shape[0] # numero de ejemplos de entrenamiento

        # realiza una copia de theta, el cual será acutalizada por el descenso por el gradiente
        theta = theta.copy()
        J_history = []

        for i in range(num_iters):
            h = calcularSigmoide(X.dot(theta.T))
            theta = theta - (alpha / m) * (h - y).dot(X)

            J_history.append(calcularCosto(theta, X, y))
        return theta, J_history
```

```
[ ] # Configurar hiperparámetros y realizar descenso por el gradiente en el conjunto de entrenamiento
    alpha = 0.25
    num_iters = 1000       #con 500 iteraciones ya tenemos una buena convergencia
    theta = np.zeros(18)
    theta, J_history = descensoGradiente(theta, X_train, y_train, alpha, num_iters)

    # Muestra los resultados del descenso por el gradiente
    print('theta calculado por el descenso por el gradiente: \n',(theta))


    # Graficar la convergencia del costo en el conjunto de entrenamiento
    pyplot.plot(np.arange(len(J_history)), (J_history), lw=2)

    pyplot.xlabel('Número de iteraciones')
    pyplot.ylabel('Costo J')
    pyplot.title('Convergencia del costo en el conjunto de entrenamiento')
    pyplot.show()
```

```
theta calculado por el descenso por el gradiente:
 [ 0.         -0.00901837 -0.06263355 -0.1220825   0.48670214  0.21782861
  3.08058609 -1.76657005 -0.69356933 -0.00901683  0.         -0.01484519
  0.00757236  0.1147444  -6.53507669  0.11474461  0.21351198  0.01808487]
```



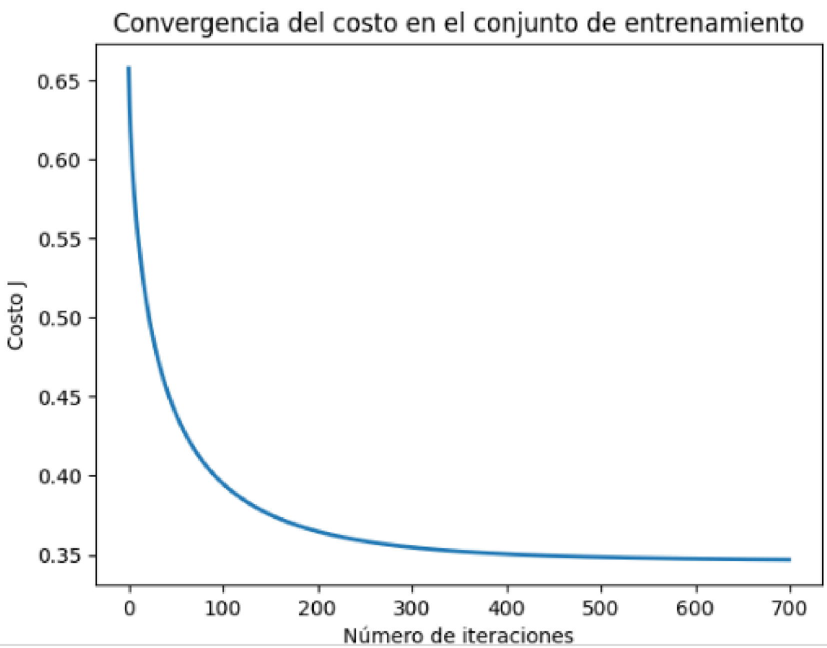
REGRESION LOGISTICA LAB 2 CON REGULARIZACION

```
[ ]  def calcularCosto(theta, X, y, lambda_):
         # Inicializar algunos valores utiles
         m = y.size  # numero de ejemplos de entrenamiento

         J = 0

         # temp = theta.copy()
         # temp[0] = 0

         #hacemos el uso de la funcion sigmoid
         h = calcularSigmoide(X.dot(theta.T))

         J = (1 / m) * np.sum(-y.dot(np.log(h)) - (1 - y).dot(np.log(1 - h)))

         # Calculamos el término de regularización (sin incluir el primer término de theta)
         regularization_term = (lambda_ / (2 * m)) * np.sum(np.square(theta[1:]))

         # Sumamos el término de regularización al costo total
         J += regularization_term

         return J
```

```
[ ]  def descensoGradiente(theta, X, y, alpha, lambda_, num_iters):
         # Inicializa algunos valores
         m = y.shape[0] # numero de ejemplos de entrenamiento

         # realiza una copia de theta, el cual será acutalizada por el descenso por el gradient
         theta = theta.copy()
         J_history = []

         for i in range(num_iters):
             h = calcularSigmoide(X.dot(theta.T))

             # Calcula el gradiente descendente sin regularización
             gradient = (1 / m) * X.T.dot(h - y)

             # Calcula el término de regularización (excepto para el término de sesgo theta[0])
             regularization_term = (lambda_ / m) * theta[1:]

             # theta[0] -= alpha * (1 / m) * np.sum(h - y)
             theta[0] -= alpha * gradient[0]
             theta[1:] -= alpha * (gradient[1:]+ regularization_term)

             # Calcula y guarda el costo en cada iteración
             J_history.append(calcularCosto(theta, X, y, lambda_))

         return theta, J_history
```

```
[ ] # Configurar hiperparámetros y realizar descenso por el gradiente en el conjunto de entrenamiento
    alpha = 0.25
    num_iters = 500       #con 500 iteraciones ya tenemos una buena convergencia
    lambda_= 700
    theta = np.zeros(18)
    theta, J_history = descensoGradiente(theta, X_train, y_train, alpha, num_iters, lambda_)

    # Muestra los resultados del descenso por el gradiente
    print('theta calculado por el descenso por el gradiente: \n',(theta))


    # Graficar la convergencia del costo en el conjunto de entrenamiento
    pyplot.plot(np.arange(len(J_history)), (J_history), lw=2)

    pyplot.xlabel('Número de iteraciones')
    pyplot.ylabel('Costo J')
    pyplot.title('Convergencia del costo en el conjunto de entrenamiento')
    pyplot.show()
```

```
theta calculado por el descenso por el gradiente:
 [ 0.          0.00489763 -0.03932334 -0.08395674  0.25224649  0.10748243
  1.54906659 -0.91894597 -0.3501945   0.00489881  0.         -0.01204493
  0.00742137  0.0239869  -3.9391566   0.02398639  0.17112238  0.0283991 ]
```



CLASIFICACION LAB 3

```
[ ] def calcularCosto(theta, X, y, lambda_):
        # Inicializa algunos valores utiles
        m = y.size

        # convierte las etiquetas a valores enteros si son boleanos
        if y.dtype == bool:
            y = y.astype(int)

        J = 0
        grad = np.zeros(theta.shape)

        h = calcularSigmoide(X.dot(theta.T))

        temp = theta
        temp[0] = 0

        J = (1 / m) * np.sum(-y.dot(np.log(h)) - (1 - y).dot(np.log(1 - h))) + (lambda_ / (2 * m)) * np.sum(np.square(temp))

        grad = (1 / m) * (h - y).dot(X)
        # Se aplica regularizacion en la siguiente linea
        grad = grad + (lambda_ / m) * temp

        return J, grad
        # j = num_real   el costo
        # grad = vector  el gradiente
```

```
[ ] # Realiza las predicciones utilizando la función predictOneVsAll
    predic = predictOneVsAll(all_theta, X_train)

    # Calcula la precisión del modelo en el conjunto de prueba
    precision_test = np.mean(predic == y_train) * 100
    print('Precision del conjunto de entrenamiento: {:.2f}%'.format(precision_test))

    # Calcula los ejemplos donde el modelo acertó y donde se equivocó
    aciertos = np.sum(predic == y_train)
    errores = np.sum(predic != y_train)

    print(f'Ejemplos acertados: {aciertos}')
    print(f'Ejemplos erróneos: {errores}')
```

```
Precision del conjunto de entrenamiento: 9.06%
Ejemplos acertados: 15970
Ejemplos erróneos: 160286
```

Por ultimo se hace las predicciones con el otro 20% de los datos

```
[ ] # Realiza las predicciones utilizando la función predictOneVsAll
    predic = predictOneVsAll(all_theta, X_test)

    # Calcula la precisión del modelo en el conjunto de prueba
    precision_test = np.mean(predic == y_test) * 100
    print('Precision del conjunto de prueba: {:.2f}%'.format(precision_test))

    # Calcula los ejemplos donde el modelo acertó y donde se equivocó
    aciertos = np.sum(predic == y_test)
    errores = np.sum(predic != y_test)

    print(f'Ejemplos acertados: {aciertos}')
    print(f'Ejemplos erróneos: {errores}')
```

```
Precision del conjunto de prueba: 8.84%
Ejemplos acertados: 3895
Ejemplos erróneos: 40169
```

CLASIFICACION LAB 3 CON REGULARIZACION

```
[14] def calcularCosto(theta, X, y, lambda_):
        # Inicializa algunos valores utiles
        m = y.size
        # convierte las etiquetas a valores enteros si son boleanos
        if y.dtype == bool:
            y = y.astype(int)

        J = 0
        grad = np.zeros(theta.shape)

        h = calcularSigmoide(X.dot(theta.T))

        temp = theta
        temp[0] = 0

        #Con regularizacion
        J = (1 / m) * np.sum(-y.dot(np.log(h)) - (1 - y).dot(np.log(1 - h))) + (lambda_ / (2 * m)) * np.sum(np.square(temp))

        #Sin regularizacion
        #J = (1 / m) * np.sum(-y.dot(np.log(h)) - (1 - y).dot(np.log(1 - h)))

        grad = (1 / m) * (h - y).dot(X)
        grad = grad + (lambda_ / m) * temp


        return J, grad
        # j = num_real   el costo
        # grad = vector  el gradiente
```

```
[29] # Realiza las predicciones utilizando la función predictOneVsAll
     predic = predictOneVsAll(all_theta, X_train)

     # Calcula la precisión del modelo en el conjunto de prueba
     precision_test = np.mean(predic == y_train) * 100
     print('Precision del conjunto de entrenamiento: {:.2f}%'.format(precision_test))

     # Calcula los ejemplos donde el modelo acertó y donde se equivocó
     aciertos = np.sum(predic == y_train)
     errores = np.sum(predic != y_train)

     print(f'Ejemplos acertados: {aciertos}')
     print(f'Ejemplos erróneos: {errores}')
```

```
Precision del conjunto de entrenamiento: 26.51%
Ejemplos acertados: 46724
Ejemplos erróneos: 129532
```

Por ultimo se hace las predicciones con el otro 20% de los datos

```
[30] # Realiza las predicciones utilizando la función predictOneVsAll
     predic = predictOneVsAll(all_theta, X_test)

     # Calcula la precisión del modelo en el conjunto de prueba
     precision_test = np.mean(predic == y_test) * 100
     print('Precision del conjunto de prueba: {:.2f}%'.format(precision_test))

     # Calcula los ejemplos donde el modelo acertó y donde se equivocó
     aciertos = np.sum(predic == y_test)
     errores = np.sum(predic != y_test)

     print(f'Ejemplos acertados: {aciertos}')
     print(f'Ejemplos erróneos: {errores}')
```

```
Precision del conjunto de prueba: 26.35%
Ejemplos acertados: 11611
Ejemplos erróneos: 32453
```

CONCLUSIONES

Es importante, ya que demuestra que la regularización es una técnica eficiente y escalable, que puede ser aplicada sin comprometer el rendimiento general y de hecho puede mejorar en casos donde se tengan varias características.

Al aplicarse la regularización se observa que si hubo un aumento en las predicciones de los Modelos “regresión, regresión logística y Clasificación”

Sin embargo, el aumento en la precisión no es muy relevante debido a que en el modelo de clasificación solo aumento en decimales, también se ve que en la regresión normal tubo un aumento de precisión igual, solo se puede destacar que el modelo solo aumento en la regresión logística pero aun así está muy por debajo del 50%. Se debería revisar nuevamente el modelo y las aplicaciones de las formulas. Además de hacer una limpieza de datos adecuada al dataset.