MOBILE PRICE CLASSIFICATION



**Proyecto final de Aprendizaje Automático y Minería de Datos**

Presentado por

**Sergio Gavilán Fernández**  [sgavil01@ucm.es](mailto:sgavil01@ucm.es)

**Alejandro Villar Rubio** [alvill04@ucm.es](mailto:alvill04@ucm.es)

Facultad de Informática, Universidad Complutense de Madrid

Grado en Desarrollo de Videojuegos

Madrid, 2019/2020

Contenido

[Resumen 3](#_Toc30091980)

[Objetivos 3](#_Toc30091981)

[Trabajo previo 3](#_Toc30091982)

[Herramientas usadas 4](#_Toc30091983)

[Regresión Logística 4](#_Toc30091984)

[Implementación 4](#_Toc30091985)

[Resultados 4](#_Toc30091986)

[Redes Neuronales 4](#_Toc30091987)

[Implementación 4](#_Toc30091988)

[Resultados 5](#_Toc30091989)

[Support Vector Machines 5](#_Toc30091990)

[Implementación 5](#_Toc30091991)

[Resultados 5](#_Toc30091992)

[Conclusiones 5](#_Toc30091993)

[Bibliografía 5](#_Toc30091994)

[Apéndices 7](#_Toc30091995)

[Apéndice A. Código utilizado 7](#_Toc30091996)

[Apéndice B. Gráficas 13](#_Toc30091997)

# Resumen

La elección de un teléfono móvil puede ser en algunas ocasiones un quebradero de cabeza para algunas personas, ya sea por su edad o simplemente porque no conocen la industria lo suficiente. Además, a este desconocimiento hay que añadir que cada día hay más dispositivos en el mercado por lo que la dificultad de su compra aumenta. Esto puede provocar la llamada **paradoja de la elección** (“La paradoja de la elección: por qué nos cuesta decidir,” n.d.), nuestra tendencia a estar menos satisfechos con nuestras adquisiciones mientras más alternativas existan.

En este proyecto se planteará una herramienta con la cuál cualquier persona podrá crear un teléfono móvil a su gusto y comprobar en qué rango de precio se encuentra. Con esto, el usuario podrá orientarse para su posterior compra.

# Objetivos

El objetivo principal de este proyecto es, como se ha nombrado anteriormente, proporcionar una herramienta con la que poder comprobar el rango de precio de un determinado teléfono móvil.

Para ello se aplicarán diversas técnicas de clasificación sobre la base de datos que se dispone y realizar una selección para usar la que proporcione una mayor precisión.

Las técnicas que se usarán son:

* Regresión logística.
* Redes neuronales.
* Support Vector Machines (SVM)

# Trabajo previo

Se dispone de una base de datos adquirida de la plataforma *Kaggle* (“Mobile Price Classification | Kaggle,” n.d.)*.* Esta proporciona dos archivos, **test.csv** y **train.csv**, con un total de 21 características que conforman un teléfono móvil.

|  |  |  |
| --- | --- | --- |
| Capacidad batería | Bluetooth | Velocidad del procesador |
| Dual Sim | Resolución cámara frontal | 4G |
| Memoria interna | Profundidad del teléfono móvil (cm) | Peso |
| Número de núcleos del procesador | Resolución de la cámara principal | Resolución de la pantalla, altura |
| Resolución de la pantalla, anchura | RAM | Altura |
| Anchura | Autonomía de la batería | 3G |
| Pantalla táctil | Wifi | Rango de precio / ID |

Hay una pequeña diferencia entre estos dos archivos, ambos tienen el mismo número de características, pero *train* tiene una columna con “price\_range” y *test* una con “id”. Posiblemente este último no tiene la información del precio porque en alguna ocasión se ha usado este conjunto de datos para alguna competición de **machine learning**. Esto hace que solo se pueda usar el archivo *train* porque es el que contiene lo que será la futura “Y”.

El conjunto de datos *train* está formado por 21 características y 2000 ejemplos. Esto hace que se pueda dividir en 3 tipos de conjuntos y aun así tener un gran número de ejemplos en cada uno. Las diferentes divisiones son: **train** (60%, 1200 ejemplos)**, validation** (20%, 400 ejemplos)y **test** (20%, 400 ejemplos). Esta división se ha realizado a través de código (ver [**Apéndice A**](#_Código_usado_para)).

De las 21 características mencionadas anteriormente, se cogerán 20 de ellas para estudiar la restante, el **rango de precio**. Esta característica puede adquirir un valor del 0 al 3, ambos incluidos, donde 0 corresponderá a la gama de teléfonos más baja y 3 a la gama más alta.

# Herramientas usadas

Para realizar este trabajo se han utilizado diversas librerías que se han estudiado durante el curso:

* **Numpy**. Es un paquete que permite manejar de forma eficiente contenedores multidimensionales de datos genéricos. (“NumPy — NumPy,” n.d.)
* **Matplotlib**. Permite trazar gráficas 2D. (“Matplotlib: Python plotting — Matplotlib 3.1.2 documentation,” n.d.)
* **SciPy**. Ecosistema basado en Python de código abierto para matemáticos, científicos e ingenieros. (“SciPy.org — SciPy.org,” n.d.)

# Regresión Logística

## Implementación

## Resultados

# Redes Neuronales

## Implementación

En primer lugar, se han preparado los siguientes datos y funciones para que los cálculos sean correctos:

* Con la función *num\_to\_vector* se ha creado una nueva matriz Y donde cada vector está formado por “0” exceptuando el valor marcado en la Y del conjunto de datos, que se pone a “1”.
* La función ­*random\_weight(L\_in, L\_out)* crea dos matrices de pesos de manera aleatoria de tamaño (L\_out, L\_in +1).
* El tamaño de las capas, donde la primera es 20, número de características; la segunda se ha decidido que sea 8 y la tercera es 4, valor máximo + 1 que puede alcanzar la Y.

En segundo lugar, se ha implementado las funciones coste y gradiente de la red neuronal. Luego se ha creado la función *backprop* que utiliza la nueva matriz Y, las matrices de pesos dados aleatoriamente y la propagación hacia delante para calcular el coste regularizado y el gradiente.

Además, como en *regresión logística* se ha creado una función para averiguar el mejor valor para *lambda* del algoritmo que se está aplicando en ese momento. Estos cálculos necesitan los dataset *train* y *validation.*

A parte de estas funcionalidades que se han nombrado se encuentra la función *get\_optimize\_theta* al igual que en regresión, la cuál devuelve los pesos óptimos según los valores proporcionados.

Una vez que se han implementado todo lo que se ha comentado (ver [**Apéndice A**](#_Código_usado_para_2)), los pasos a seguir son los siguientes:

1. Se usa la función *num\_vector* para crear las matrices de “Y” e “Y\_val”.
2. Se asignan valores aleatorios a dos matrices de pesos, para ello está *random\_weight.*
3. Se concatenan los pesos en un solo vector.
4. Se averigua el valor de *lambda* que proporciona la mejor precisión para respectivo algoritmo.
5. Se averiguan los pesos óptimos según la información que hay hasta el momento.
6. Por último, se averigua el coste correspondiente usando “X\_test” y los pesos óptimos. Además, se calcula la precisión de los cálculos mediante “Y\_test”.

Queda añadir que se han usado un total de 5 algoritmos: BGFS, CG, L-BFGS-B, SLSQP y TNC. Se han excluido los demás algoritmos debido al tiempo que ocupa hacer tantos cada vez que se quiere mostrar una gráfica.

## Resultados

Se han extraído tres tipos de gráficas:

* El porcentaje de error que tienen los valores de las distintas *lambdas* de cada algoritmo (ver [**Apéndice B**](#_Redes_Neuronales._Error)). Lo más llamativo es que a medida que aumenta el valor de *lambda* más error de clasificación se obtiene.
* La precisión que tienen los algoritmos con cada una de las *lambdas* (ver [**Apéndice B**](#_Redes_Neuronales._Precisión)). Se puede observar que no hay mucha diferencia de precisión entre los distintos valores, las mayores fluctuaciones las sufre el algoritmo TNC.
* Precisión de los algoritmos con el mejor valor de *lambda* respectivamente, modificando el tamaño de la capa oculta (ver [**Apéndice B**](#_Redes_Neuronales._Precisión_1)). Las precisiones son muy similares, se podría descartar estas diferencias y escoger cualquiera, pero se puede observar que hay dos algoritmos que han resultado ser los mejores, BFGS y L-BFGS-B, ambos con un tamaño de capa 8.

Nota: Es posible que los resultados no coincidan entre las diversas gráficas debido a que son distintas ejecuciones.

# Support Vector Machines

## Implementación

## Resultados

# Conclusiones

# Bibliografía

La paradoja de la elección: por qué nos cuesta decidir. (n.d.). Retrieved December 24, 2019, from https://hipertextual.com/2015/07/paradoja-eleccion

Matplotlib: Python plotting — Matplotlib 3.1.2 documentation. (n.d.). Retrieved January 10, 2020, from https://matplotlib.org/

Mobile Price Classification | Kaggle. (n.d.). Retrieved December 24, 2019, from https://www.kaggle.com/iabhishekofficial/mobile-price-classification

NumPy — NumPy. (n.d.). Retrieved January 10, 2020, from https://numpy.org/

SciPy.org — SciPy.org. (n.d.). Retrieved January 10, 2020, from https://www.scipy.org/

# Apéndices

## Apéndice A. Código utilizado

##### Código usado para dividir el dataset principal

from pandas.io.parsers import read\_csv

import csv

import numpy as np

def carga\_csv(file\_name):

    valores = read\_csv(file\_name, header=None).values

    # suponemos que siempre trabajaremos con float

    return valores[1:]

def main():

    datos = carga\_csv("train.csv")

    # Eliminamos la primera fila con los atributos del dataset

    datosLen = len(datos)  # 2001 casos de entrenamiento

    # Cogemos el 60% de los datos set de entrenamiento

    n\_training = int(datosLen\*0.6)

    training\_set = datos[:n\_training]

    # Por otro lado el 20% para el set de validacion

    n\_validation = int(datosLen\*0.2)

    validation\_set = datos[n\_training:n\_training+n\_validation]

    # Por ultimo el 20% restante para el conjunto de prueba

    n\_test = datosLen - n\_training - n\_validation

    test\_set = datos[-n\_test:]

    with open('ProcessedDataSet/train.csv', mode='w', newline='') as processedTraining:

        processedTrainingWriter = csv.writer(

            processedTraining, delimiter=',')

        processedTrainingWriter.writerows(training\_set)

    with open('ProcessedDataSet/validation.csv', mode='w', newline='') as processedValidation:

        processedValidationWriter = csv.writer(

            processedValidation, delimiter=',')

        processedValidationWriter.writerows(validation\_set)

    with open('ProcessedDataSet/test.csv', mode='w', newline='') as processedTest:

        processedTestWriter = csv.writer(processedTest, delimiter=',')

        processedTestWriter.writerows(test\_set)

main()

##### Código usado para Regresión Logística

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import minimize

import math

########################################################################

###############             FUNCIONES BASICAS          #################

########################################################################

""" Sigmoide """

def sigmoid(z):

    sigmoid = 1 / (1 + np.exp(-z))

    return sigmoid

""" Calcula el coste de un determinado conjunto de ejemplos """

def f\_cost(Theta, X, Y, reg):

    m = X.shape[0]

    h\_theta = sigmoid(np.dot(X, Theta))

    # Calculo del coste sin el termino de regularizacion

    term1 = np.dot(-Y.T, np.log(h\_theta))

    term2 = np.dot((1 - Y).T, np.log(1 - h\_theta))

    # Calculo del termino de regularizacion

    reg\_term = (reg / (2 \* m)) \* np.sum(np.square(Theta[1:]))

    # Calculo del coste

    cost = (np.sum(term1 - term2) / m) + reg\_term

    return cost

""" Calcula el gradiente de un determinado conjunto de ejemplos """

def f\_gradient(Theta, X, Y, reg):

    m = X.shape[0]

    h\_theta = sigmoid(np.dot(X, Theta))

    # Calculo del gradiente sin el termino de regularizacion

    reg\_term = (reg / m) \* (Theta[1:])

    # Calculo del termino de regularizacion

    gradient = (1 / m) \* np.dot(X.T, (h\_theta - Y))

    # Calculo del gradiente

    gradient[1:] = gradient[1:] + reg\_term

    return gradient

""" Devuelve el coste y el gradiente """

def f\_opt(Theta, X, Y, reg):

    return f\_cost(Theta, X, Y, reg), f\_gradient(Theta, X, Y, reg)

########################################################################

#############   FUNCIONES USADAS PARA EL ENTRENAMIENTO   ###############

########################################################################

""" Calcula el Theta optimo """

def get\_optimize\_theta(X, Y, reg, comp\_method, use\_jac):

    initial\_theta = np.zeros((X.shape[1], 1))

    if use\_jac:

        optTheta = minimize(fun=f\_cost, x0=initial\_theta,

                            args=(X, Y, reg), method=comp\_method, jac=f\_gradient)

    else:

        optTheta = minimize(fun=f\_cost, x0=initial\_theta,

                            args=(X, Y, reg), method=comp\_method)

    return optTheta.x

""" Selecciona el mejor termino de regularizacion de una tupla de posibles valores """

def lambda\_term\_selection(X, Y, X\_val, Y\_val, comp\_method, use\_jac):

    lambda\_vec = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3])

    error\_train = np.zeros((len(lambda\_vec), 1))

    error\_val = np.zeros((len(lambda\_vec), 1))

    for i in range(len(lambda\_vec)):

        reg = lambda\_vec[i]

        newY = np.array((Y == i) \* 1)

        newY = newY[:None]

        newY\_val = np.array((Y\_val == i) \* 1)

        newY\_val = newY\_val[:None]

        Theta = get\_optimize\_theta(X, newY, reg, comp\_method, use\_jac)

        error\_train[i] = f\_opt(Theta, X, newY, reg)[0]

        error\_val[i] = f\_opt(Theta, X\_val, newY\_val, reg)[0]

    #draw\_lambda\_values(lambda\_vec, error\_train, error\_val, method=comp\_method)

    best\_lambda = 0

    min\_error = float("inf")

    for i in range(len(lambda\_vec)):

        if not math.isnan(error\_val[i]) and error\_val[i] < min\_error:

            min\_error = error\_val[i]

            best\_lambda = lambda\_vec[i]

    return best\_lambda

""" Entrena los clasificadores de cada clase """

def oneVsAll(X, Y, num\_of\_price\_range, reg, comp\_method, use\_jac):

    # Numero de propiedades de los ejemplos

    n = X.shape[1]

    matResult = np.zeros((num\_of\_price\_range, n))  # (4, 21)

    for i in range(num\_of\_price\_range):

        # Se obtiene una nueva "y" donde se indica si el ejemplo

        # j-esimo pertence a dicha clase o no.

        newY = np.array((Y == i) \* 1)

        newY = newY[:None]

        matResult[i] = get\_optimize\_theta(

            X, newY, reg, comp\_method, use\_jac).ravel()

    return matResult

""" Calcula la precision """

def testClassificator(Theta, X, Y):

    aciertos = 0

    for m in range(X.shape[0]):  # Para cada ejemplo de entrenamiento

        bestClassificator = -1

        index = 0

        for j in range(Theta.shape[0]):  # Ponemos a prueba cada clasificador

            result = sigmoid(np.dot(Theta[j], X[m]))

            if(result > bestClassificator):

                bestClassificator = result

                index = j

        if(index == Y[m]):

            aciertos += 1

    precission = round((aciertos / X.shape[0]) \* 100, 1)

    return precission

def draw\_lambda\_values(lambda\_values, error\_train, error\_val, method):

    plt.figure(figsize=(8, 5))

    plt.plot(lambda\_values, error\_val, 'or--', label='Validation Set Error')

    plt.plot(lambda\_values, error\_train, 'bo--', label='Training Set Error')

    plt.xlabel('$\lambda$ value', fontsize=16)

    plt.ylabel('Classification Error [%]', fontsize=14)

    plt.title(f'Finding Best $\lambda$ value for method {method}', fontsize=18)

    plt.xscale('log')

    plt.legend()

    plt.show()

def logistic\_regression(X, Y, X\_val, Y\_val, X\_test, Y\_test, method, jac):

    #best\_lambda = lambda\_term\_selection(X, Y, X\_val, Y\_val, method, jac)

    optTheta = oneVsAll(X, Y, 4, 1, method, jac)

    print(f'método {method} terminado con éxito!')

    return testClassificator(optTheta, X\_test, Y\_test)

from matplotlib.ticker import FuncFormatter

import matplotlib.pyplot as plt

import numpy as np

from pandas.io.parsers import read\_csv

import reg\_logistica

'''

    Theta: (n + 1, 1)

    X: (m, n + 1)

    Y: (m, 1)

'''

def draw\_precission(precission):

    plt.figure(figsize=(14, 6))

    plt.title('Regularized Logistic Regression with $\lambda$ = 1')

    plt.xlabel('Algorithm method')

    plt.ylabel('Precission')

    plt.ylim(0, 100)

    x = np.arange(len(precission))

    rects = plt.bar(x, precission, color='red')

    plt.xticks(x, ('CG', 'BFGS', 'L-BFGS-B', 'TNC', 'SLSQP'))

    for rect in rects:

        height = rect.get\_height()

        plt.annotate('{}%'.format(height),

                     xy=(rect.get\_x() + rect.get\_width() / 2, height / 2),

                     xytext=(0, 3),  # 3 points vertical offset

                     textcoords="offset points",

                     ha='center', va='bottom', color=(1.0, 1.0, 1.0, 1.0),

                    fontsize=20, weight='bold')

    plt.show()

# Carga un fichero ".csv" y devuelve los datos

def load\_data(file\_name):

    values = read\_csv(file\_name, header=None).values

    return values.astype(float)

def normalize\_matrix(X):

    mu = np.mean(X, axis=0)

    X\_norm = X - mu

    sigma = np.std(X\_norm, axis=0)

    X\_norm = X\_norm / sigma

    return X\_norm

def get\_data\_matrix(data):

    X = np.delete(data, data.shape[1] - 1, axis=1)  # (1200, 20)

    X = normalize\_matrix(X)

    X = np.insert(X, 0, 1, axis=1)  # (1200, 21)

    Y = data[:, data.shape[1] - 1]  # (1200,)

    return X, Y

def main():

    train\_data = load\_data("../ProcessedDataSet/train.csv")

    validation\_data = load\_data("../ProcessedDataSet/validation.csv")

    test\_data = load\_data("../ProcessedDataSet/test.csv")

    X, Y = get\_data\_matrix(train\_data)

    X\_val, Y\_val = get\_data\_matrix(validation\_data)

    X\_test, Y\_test = get\_data\_matrix(test\_data)

    cg\_precission = reg\_logistica.logistic\_regression(

        X, Y, X\_val, Y\_val, X\_test, Y\_test, 'CG', True)

    bfgs\_precission = reg\_logistica.logistic\_regression(

        X, Y, X\_val, Y\_val, X\_test, Y\_test, 'BFGS', True)

    l\_bfgs\_b\_precission = reg\_logistica.logistic\_regression(

        X, Y, X\_val, Y\_val, X\_test, Y\_test, 'L-BFGS-B', True)

    tnc\_precission = reg\_logistica.logistic\_regression(

        X, Y, X\_val, Y\_val, X\_test, Y\_test, 'TNC', True)

    slsqp\_precission = reg\_logistica.logistic\_regression(

        X, Y, X\_val, Y\_val, X\_test, Y\_test, 'SLSQP', True)

    precission = [cg\_precission, bfgs\_precission, l\_bfgs\_b\_precission, tnc\_precission, slsqp\_precission]

    draw\_precission(precission)

main()

##### Código usado para Redes Neuronales

import numpy as np

import scipy.optimize as opt

import matplotlib.pyplot as plt

import math

# Función sigmoide

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

# Cálculo de la derivada de la función sigmoide

def der\_sigmoid(z):

    return (sigmoid(z) \* (1.0 - sigmoid(z)))

# Cáculo del coste no regularizado

def coste\_no\_reg(m, h, y):

    J = 0

    for i in range(m):

        J += np.sum(-y[i] \* np.log(h[i]) \

             - (1 - y[i]) \* np.log(1 - h[i]))

    return (J / m)

# Cálculo del coste regularizado

def f\_cost(m, h, Y, reg, theta1, theta2):

    return (coste\_no\_reg(m, h, Y) +

        ((reg / (2 \* m)) \*

        (np.sum(np.square(theta1[:, 1:])) +

        np.sum(np.square(theta2[:, 1:])))))

# Inicializa una matriz de pesos aleatorios

def random\_weight(L\_in, L\_out):

    ini = 0.12

    theta = np.random.uniform(low=-ini, high=ini, size=(L\_out, L\_in))

    theta = np.hstack((np.ones((theta.shape[0], 1)), theta))

    return theta

def num\_to\_vector(n, output\_layer):

    lenN = len(n)

    n = n.ravel()

    n\_onehot = np.zeros((lenN, output\_layer))

    for i in range(lenN):

        n\_onehot[i][int(n[i])] = 1

    return n\_onehot

""" Selecciona el mejor termino de regularizacion de una tupla de posibles valores """

def lambda\_term\_selection(nn\_params, input\_layer, hidden\_layer , output\_layer, X, Y\_onehot, \

    X\_val, Y\_val\_onehot, comp\_method, use\_jac):

    lambda\_vec = np.array([0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10])

    error\_train = np.zeros((len(lambda\_vec), 1))

    error\_val = np.zeros((len(lambda\_vec), 1))

    for i in range(len(lambda\_vec)):

        reg = lambda\_vec[i]

        Theta = get\_optimize\_theta(nn\_params, input\_layer, hidden\_layer, \

        output\_layer, X, Y\_onehot, reg, comp\_method, use\_jac)

        # Despliegue de params\_rn para sacar las Thetas

        theta1 = np.reshape(Theta.x[:hidden\_layer \* (input\_layer + 1)],

                (hidden\_layer, (input\_layer + 1)))

        theta2 = np.reshape(Theta.x[hidden\_layer \* (input\_layer + 1): ],

            (output\_layer, (hidden\_layer + 1)))

        a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)

        error\_train[i] = f\_cost(X.shape[0], h, Y\_onehot, reg, theta1, theta2)

        a1, z2, a2, z3, h\_val = forward\_propagate(X\_val, theta1, theta2)

        error\_val[i] = f\_cost(X\_val.shape[0], h\_val, Y\_val\_onehot, reg, theta1, theta2)

    best\_lambda = 0

    min\_error = float("inf")

    for i in range(len(lambda\_vec)):

        if not math.isnan(error\_val[i]) and error\_val[i] < min\_error:

            min\_error = error\_val[i]

            best\_lambda = lambda\_vec[i]

    return best\_lambda

def get\_optimize\_theta(nn\_params, input\_layer, hidden\_layer , output\_layer, X, Y\_onehot, reg, comp\_method, use\_jac):

    initial\_theta = np.zeros((X.shape[1], 1))

    # Obtención de los pesos óptimos entrenando una red con los pesos aleatorios

    if use\_jac:

        optTheta = opt.minimize(

            fun=backprop,

            x0=nn\_params,

            args=(input\_layer, hidden\_layer, output\_layer, X, Y\_onehot, reg),

            method=comp\_method,

            jac=True,

            options={'maxiter': 70})

    else:

        optTheta = opt.minimize(

            fun=backprop,

            x0=nn\_params,

            args=(input\_layer, hidden\_layer, output\_layer, X, Y\_onehot, reg),

            method=comp\_method,

            options={'maxiter': 70})

    return optTheta

# Devuelve "Y" a partir de una X y no unos pesos determinados

def forward\_propagate(X, theta1, theta2):

    m = X.shape[0]

    a1 = np.hstack([np.ones([m, 1]), X])    # (5000, 401)

    z2 = np.dot(a1, theta1.T)   # (5000, 25)

    a2 = np.hstack([np.ones([m, 1]), sigmoid(z2)])  # (5000, 26)

    z3 = np.dot(a2, theta2.T)   # (5000, 10)

    h = sigmoid(z3) # (5000, 10)

    return a1, z2, a2, z3, h

# Devuelve el coste y el gradiente de una red neuronal de dos capas

def backprop(params\_rn, input\_layer, hidden\_layer, output\_layer, X, y, reg):

    m = X.shape[0]

    # Despliegue de params\_rn para sacar las Thetas

    theta1 = np.reshape(params\_rn[:hidden\_layer \* (input\_layer + 1)],

            (hidden\_layer, (input\_layer + 1)))

    theta2 = np.reshape(params\_rn[hidden\_layer \* (input\_layer + 1): ],

        (output\_layer, (hidden\_layer + 1)))

    a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)

    coste = f\_cost(m, h, y, reg, theta1, theta2) # Coste regularizado

    # Inicialización de dos matrices "delta" a 0 con el tamaño de los thethas respectivos

    delta1 = np.zeros\_like(theta1)

    delta2 = np.zeros\_like(theta2)

    # Por cada ejemplo

    for t in range(m):

        a1t = a1[t, :] # (1, 401)

        a2t = a2[t, :] # (1, 26)

        ht = h[t, :] # (1, 10)

        yt = y[t]

        d3t = ht - yt

        d2t = np.dot(theta2.T, d3t) \* (a2t \* (1 - a2t)) # (1, 26)

        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])

        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    delta1 = delta1 / m

    delta2 = delta2 / m

    # Gradiente perteneciente a cada delta

    delta1[:, 1:] = delta1[:, 1:] + (reg \* theta1[:, 1:]) / m

    delta2[:, 1:] = delta2[:, 1:] + (reg \* theta2[:, 1:]) / m

    # Concatenación de los gradientes

    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

    return coste, grad

# Cálculo de la precisión

def testClassificator(h, Y):

    aciertos = 0

    for i in range (h.shape[0]):

        max = np.argmax(h[i])

        if max == Y[i]:

            aciertos += 1

    precision = round((aciertos / h.shape[0]) \* 100, 1)

    return precision

def training\_neural\_network(X, Y, X\_val, Y\_val, X\_test, Y\_test, input\_layer, hidden\_layer, output\_layer, \

    comp\_method, use\_jac):

    # Transforma Y en un vector

    Y\_onehot = num\_to\_vector(Y, output\_layer)

    Y\_val\_onehot = num\_to\_vector(Y\_val, output\_layer)

    # Inicialización de dos matrices de pesos de manera aleatoria

    Theta1 = random\_weight(input\_layer, hidden\_layer)

    Theta2 = random\_weight(hidden\_layer, output\_layer)

    # Crea una lista de Thetas

    Thetas = [Theta1, Theta2]

    # Concatenación de las matrices de pesos en un solo vector

    unrolled\_Thetas = [Thetas[i].ravel() for i,\_ in enumerate(Thetas)]

    nn\_params = np.concatenate(unrolled\_Thetas)

    reg = lambda\_term\_selection(nn\_params, input\_layer, hidden\_layer, output\_layer, \

        X, Y\_onehot, X\_val, Y\_val\_onehot, comp\_method, use\_jac)

    optTheta = get\_optimize\_theta(nn\_params, input\_layer, hidden\_layer, \

        output\_layer, X, Y\_onehot, reg, comp\_method, use\_jac)

    # Desglose de los pesos óptimos en dos matrices

    newTheta1 = np.reshape(optTheta.x[:hidden\_layer \* (input\_layer + 1)],

        (hidden\_layer, (input\_layer + 1)))

    newTheta2 = np.reshape(optTheta.x[hidden\_layer \* (input\_layer + 1): ],

        (output\_layer, (hidden\_layer + 1)))

    # H, resultado de la red al usar los pesos óptimos

    a1, z2, a2, z3, h = forward\_propagate(X\_test, newTheta1, newTheta2)

    # Cálculo de la precisión

    return testClassificator(h, Y\_test)

import numpy as np

from pandas.io.parsers import read\_csv

import matplotlib.pyplot as plt

import scipy.optimize as opt

import neural\_network

'''

3 capas:

    + 20 en la primera capa (la primera siempre fijada +1)

    + 8 en la capa oculta

    + 4 en la de salida

Theta1 de dimension (8 x 21)

Theta2 de dimension (4 x 9)

'''

def draw\_precission(precission):

    plt.figure(figsize=(14, 6))

    plt.title('Neural Network Precission with best $\lambda$ for each method')

    plt.xlabel('Algorithm method')

    plt.ylabel('Precission')

    plt.ylim(0, 100)

    x = np.arange(len(precission))

    rects = plt.bar(x, precission)

    plt.xticks(x, ('CG', 'BFGS', 'L-BFGS-B', 'TNC', 'SLSQP'))

    for rect in rects:

        height = rect.get\_height()

        plt.annotate('{}%'.format(height),

                    xy=(rect.get\_x() + rect.get\_width() / 2,

                    height / 2),

                    xytext=(0, 0),  # 3 points vertical offset

                    textcoords="offset points",

                    ha='center', va='bottom', color=(0.0, 0.0, 0.0, 1.0),

                    fontsize=20, weight='bold')

    plt.show()

# Carga un fichero ".csv" y devuelve los datos

def load\_data(file\_name):

    values = read\_csv(file\_name, header=None).values

    return values.astype(float)

def normalize\_matrix(X):

    mu = np.mean(X, axis=0)

    X\_norm = X - mu

    sigma = np.std(X\_norm, axis=0)

    X\_norm = X\_norm / sigma

    return X\_norm

def get\_data\_matrix(data):

    X = np.delete(data, data.shape[1] - 1, axis=1) # (1200, 20)

    X = normalize\_matrix(X)

    X = np.insert(X, 0, 1, axis=1) # (1200, 21)

    Y = data[:, data.shape[1] - 1] # (1200,)

    return X, Y

def main():

    train\_data = load\_data("../ProcessedDataSet/train.csv")

    validation\_data = load\_data("../ProcessedDataSet/validation.csv")

    test\_data = load\_data("../ProcessedDataSet/test.csv")

    X, Y = get\_data\_matrix(train\_data)

    X\_val, Y\_val = get\_data\_matrix(validation\_data)

    X\_test, Y\_test = get\_data\_matrix(test\_data)

    input\_layer = X.shape[1]

    hidden\_layer = 32

    output\_layer = 4

    cg\_precission = neural\_network.training\_neural\_network(X, Y, X\_val, Y\_val, X\_test, Y\_test, input\_layer, \

        hidden\_layer, output\_layer, 'CG', True)

    bfgs\_precission = neural\_network.training\_neural\_network(X, Y, X\_val, Y\_val, X\_test, Y\_test, input\_layer, \

        hidden\_layer, output\_layer, 'BFGS', True)

    l\_bfgs\_b\_precission = neural\_network.training\_neural\_network(X, Y, X\_val, Y\_val, X\_test, Y\_test, input\_layer, \

        hidden\_layer, output\_layer, 'L-BFGS-B', True)

    tnc\_precission = neural\_network.training\_neural\_network(X, Y, X\_val, Y\_val, X\_test, Y\_test, input\_layer, \

        hidden\_layer, output\_layer, 'TNC', True)

    slsqp\_precission = neural\_network.training\_neural\_network(X, Y, X\_val, Y\_val, X\_test, Y\_test, input\_layer, \

        hidden\_layer, output\_layer, 'SLSQP', True)

    precission = [cg\_precission, bfgs\_precission, l\_bfgs\_b\_precission, tnc\_precission, slsqp\_precission]

    draw\_precission(precission)

main()

## Apéndice B. Gráficas

##### Regresión Logística. Error de clasificación según el valor de *lambda.*

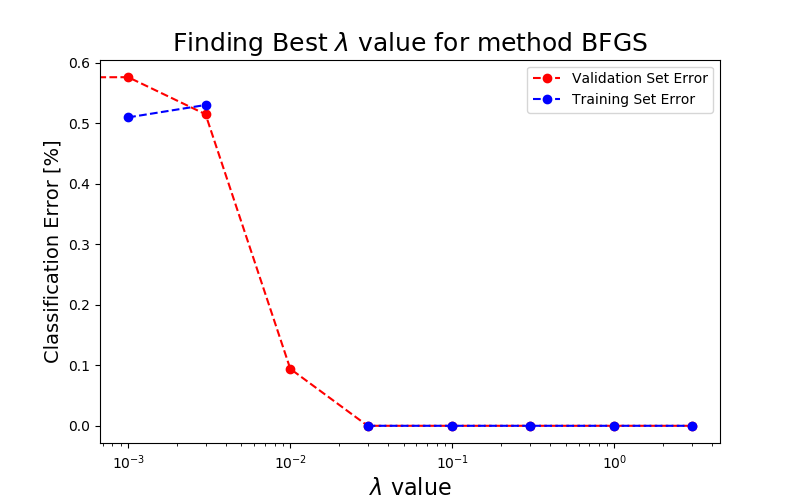


Ilustración 1. Error de clasificación para los distintos valores de lambda usando el algoritmo BFGS

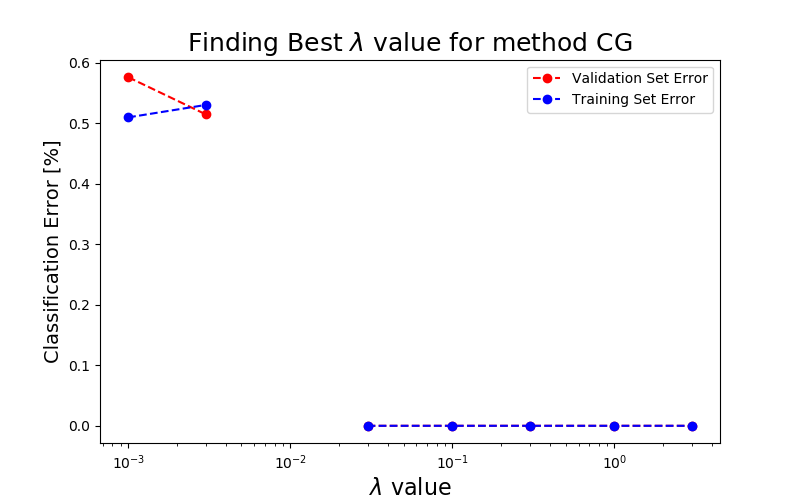


Ilustración 2. Error de clasificación para los distintos valores de lambda usando el algoritmo CG

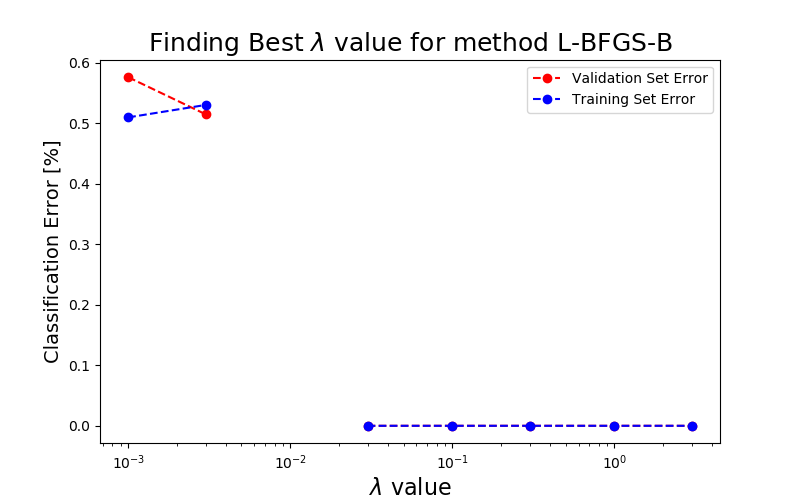


Ilustración 3. Error de clasificación para los distintos valores de lambda usando el algoritmo L-BFGS-B

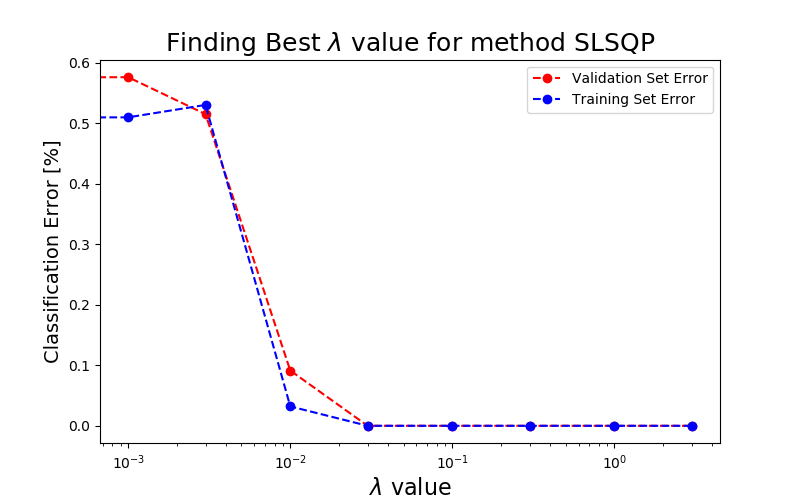


Ilustración 4. Error de clasificación para los distintos valores de lambda usando el algoritmo SLSQP

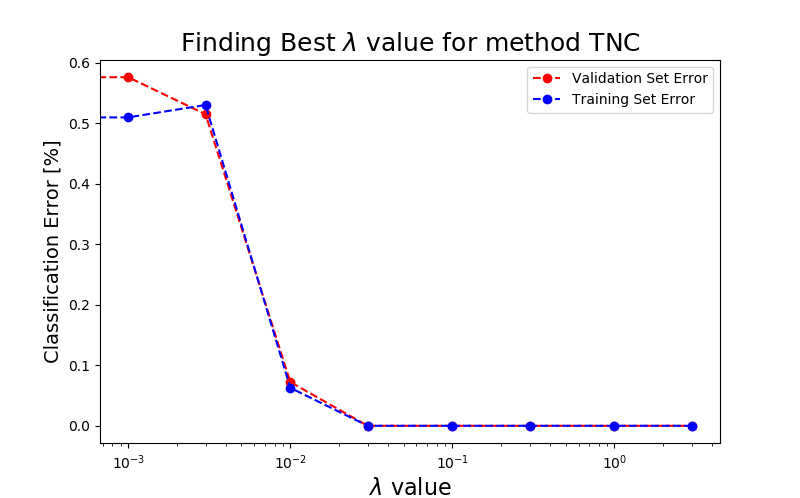


Ilustración 5. Error de clasificación para los distintos valores de lambda usando el algoritmo TNC

##### Regresión Logística. Precisión de los algoritmos con cada una de las *lambdas*.

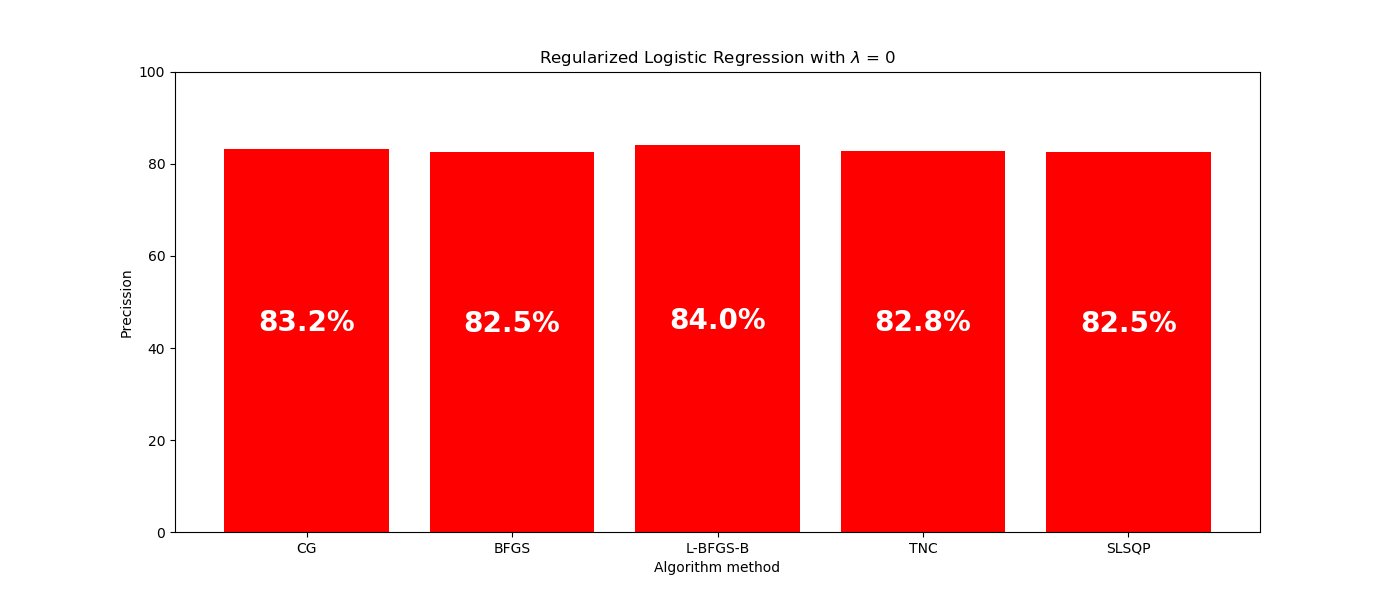


Ilustración 6. Precisión de la regresión logística regularizada para lambda 0

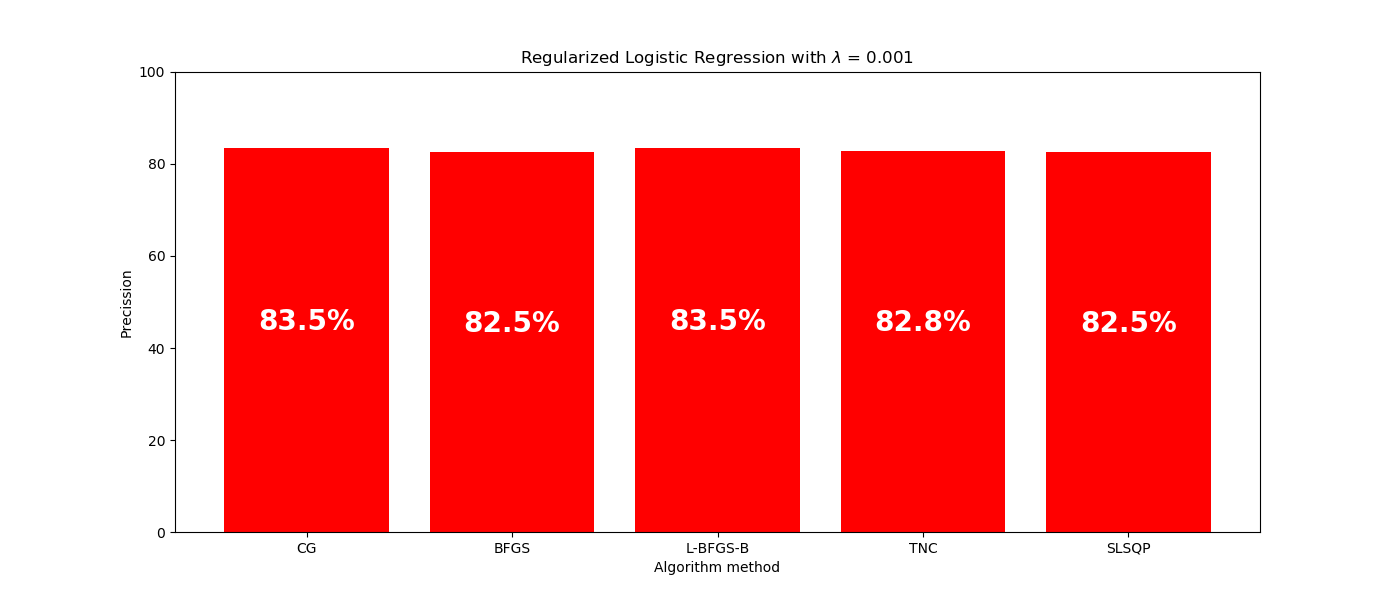


Ilustración 7. Precisión de la regresión logística regularizada para lambda 0.001

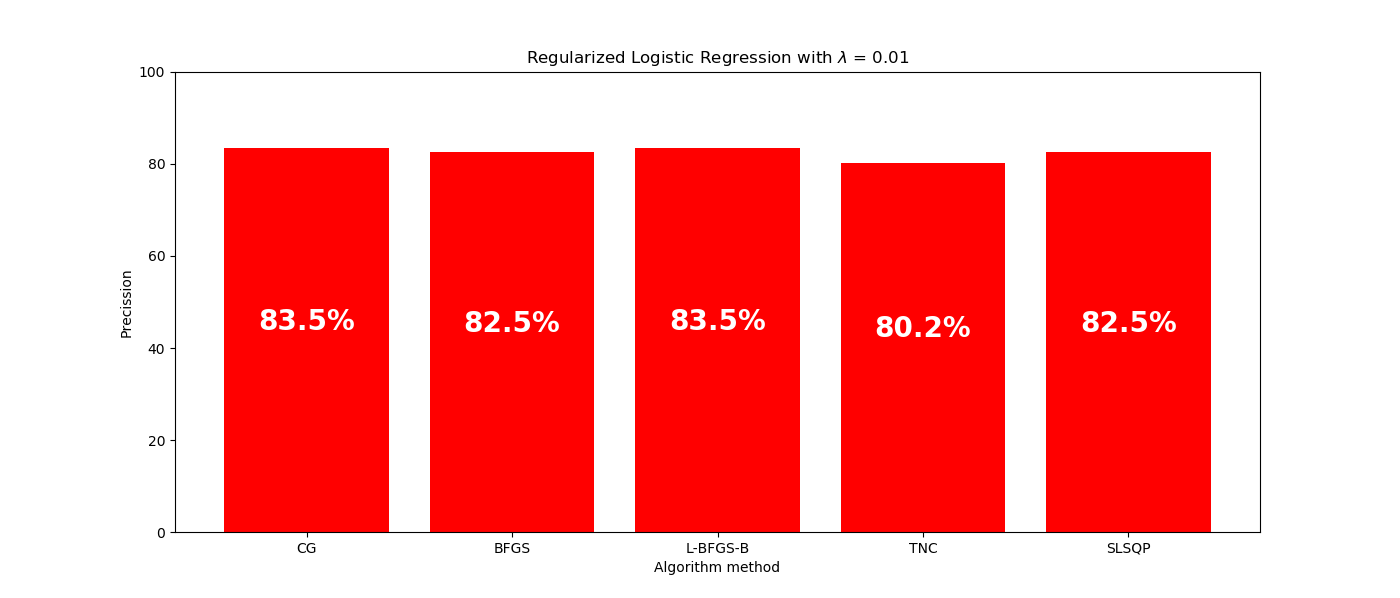


Ilustración 8. Precisión de la regresión logística regularizada para lambda 0.01

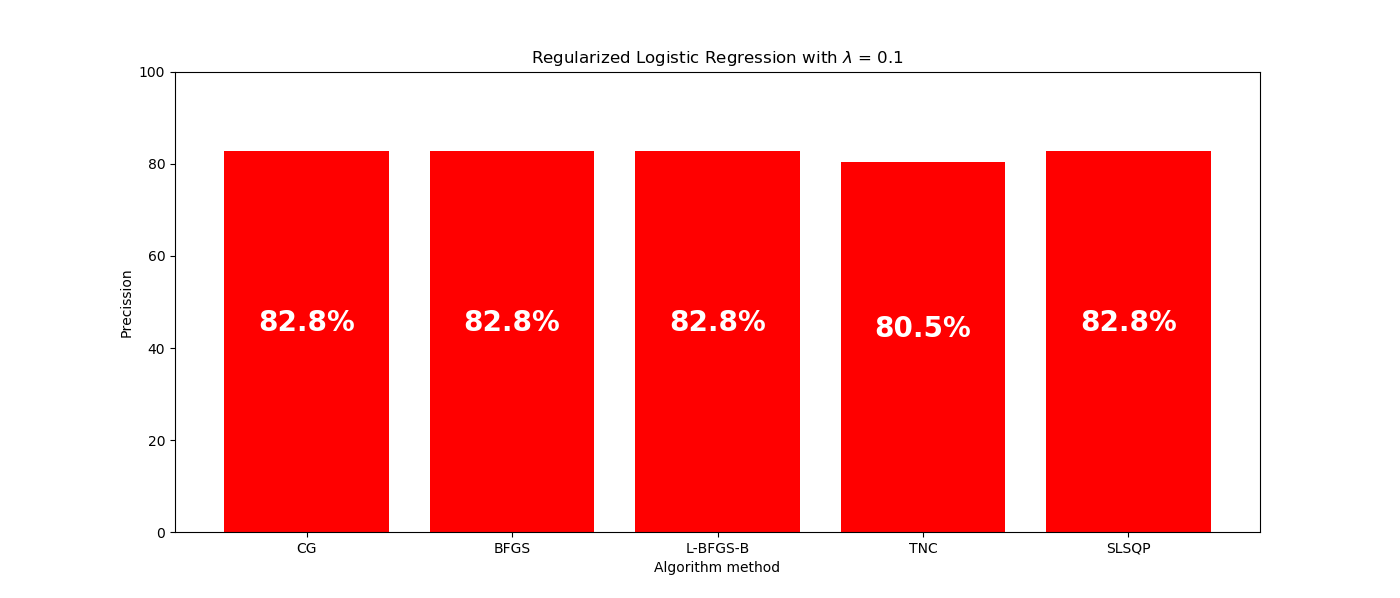


Ilustración 9. Precisión de la regresión logística regularizada para lambda 0.1

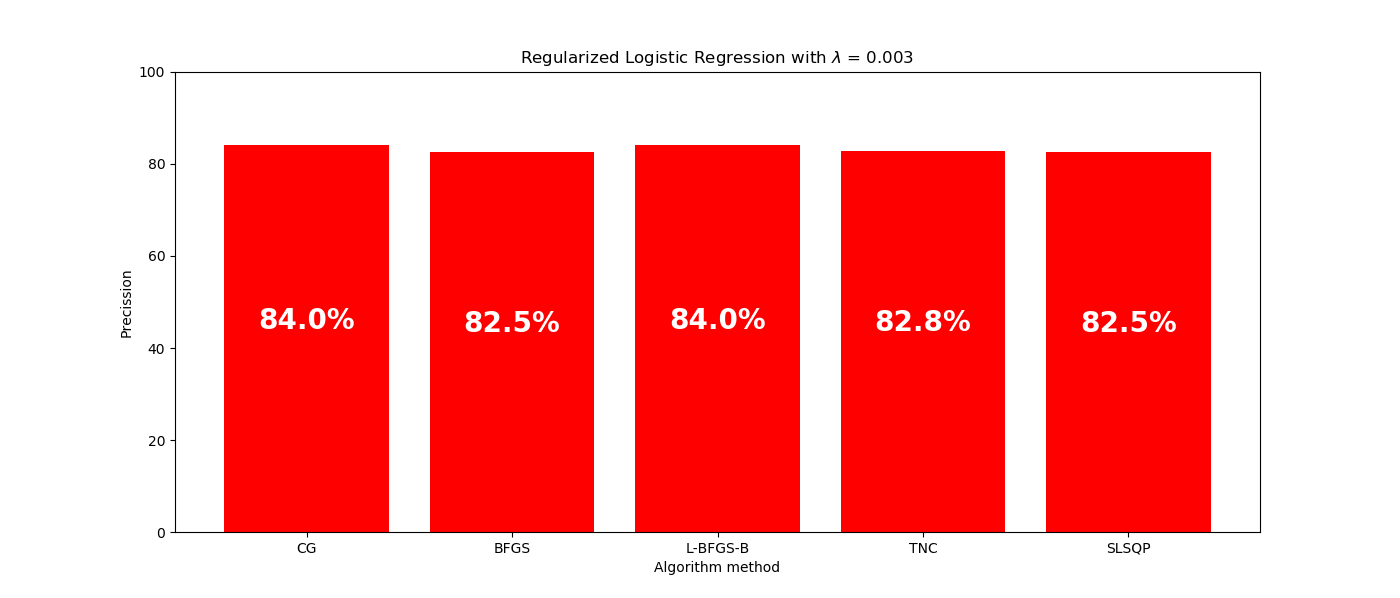


Ilustración 10. Precisión de la regresión logística regularizada para lambda 0.003

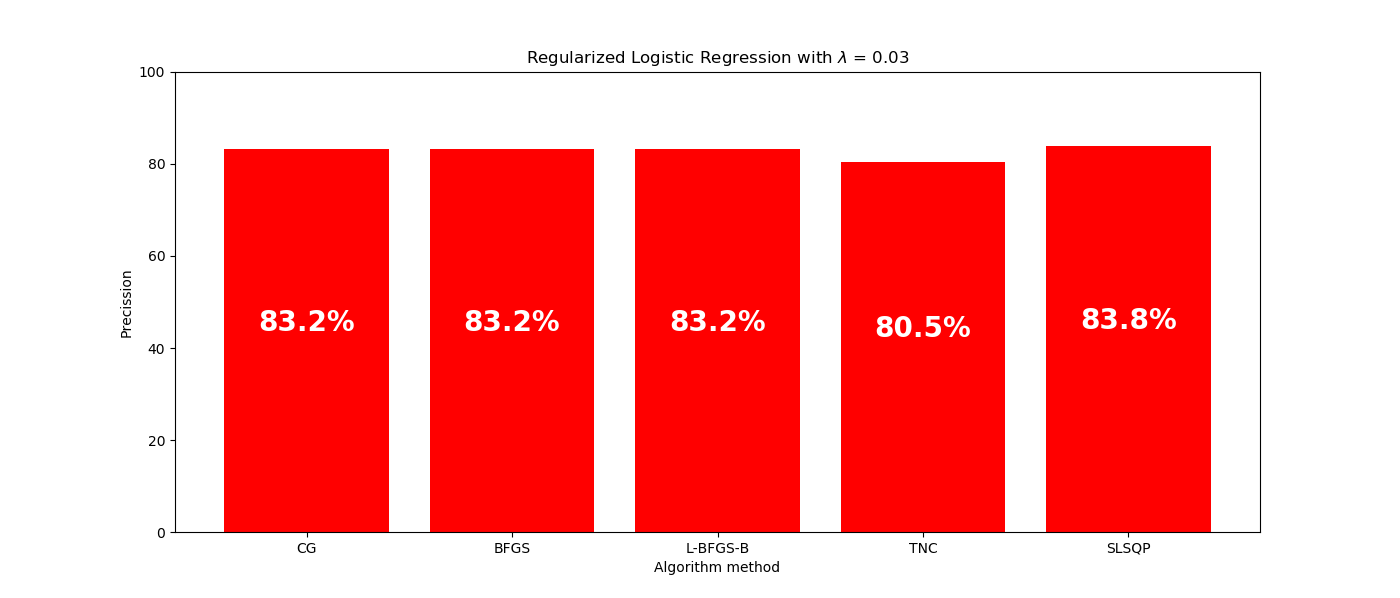


Ilustración 11. Precisión de la regresión logística regularizada para lambda 0.03

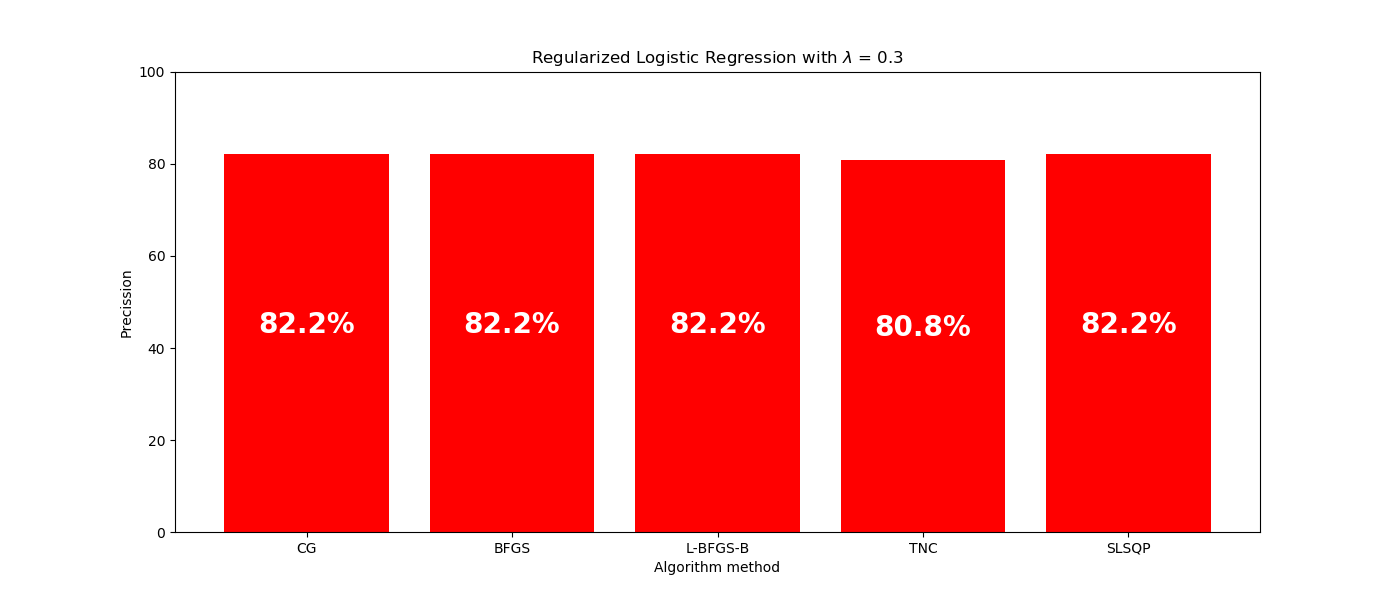


Ilustración 12. Precisión de la regresión logística regularizada para lambda 0.3

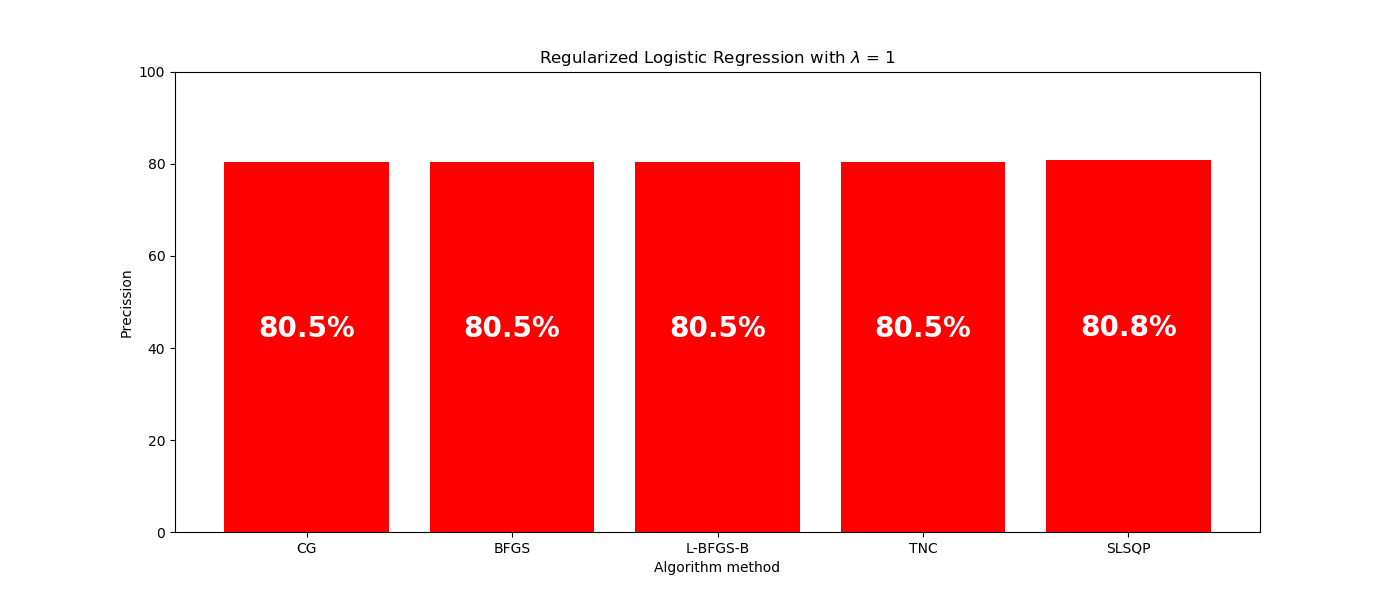


Ilustración 13. Precisión de la regresión logística regularizada para lambda 1

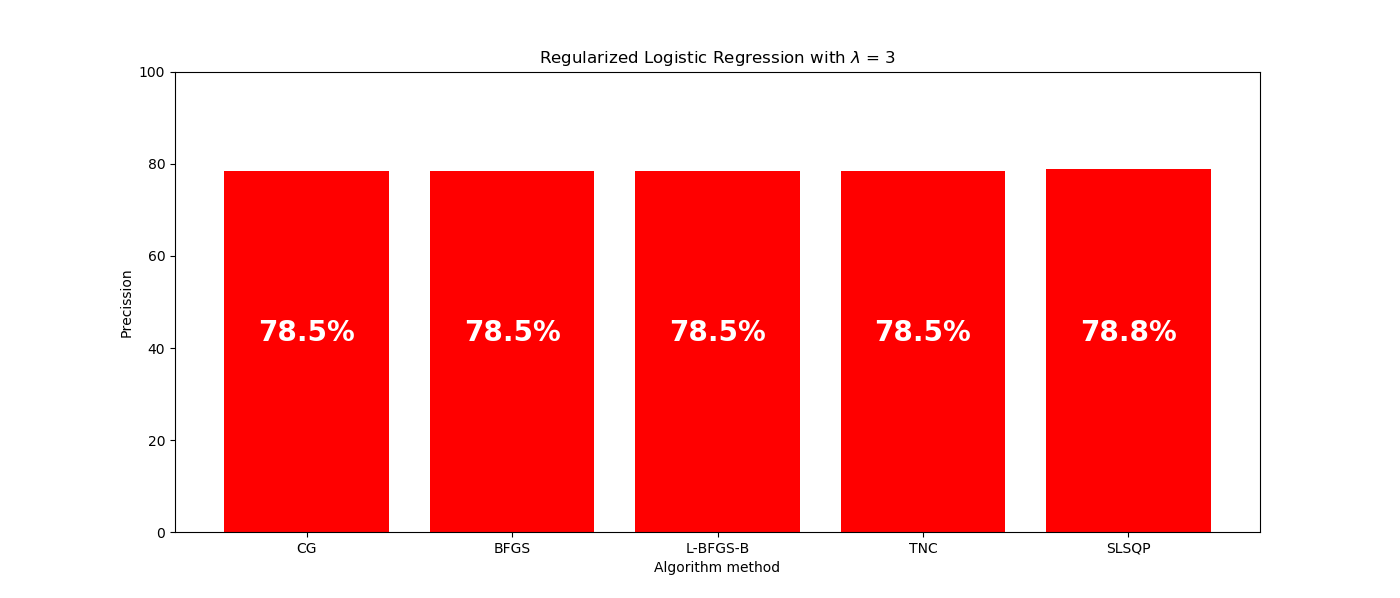


Ilustración 14. Precisión de la regresión logística regularizada para lambda 3

##### Regresión Logística. Precisión de los algoritmos con el mejor valor de *lambda* para cada uno de ellos.

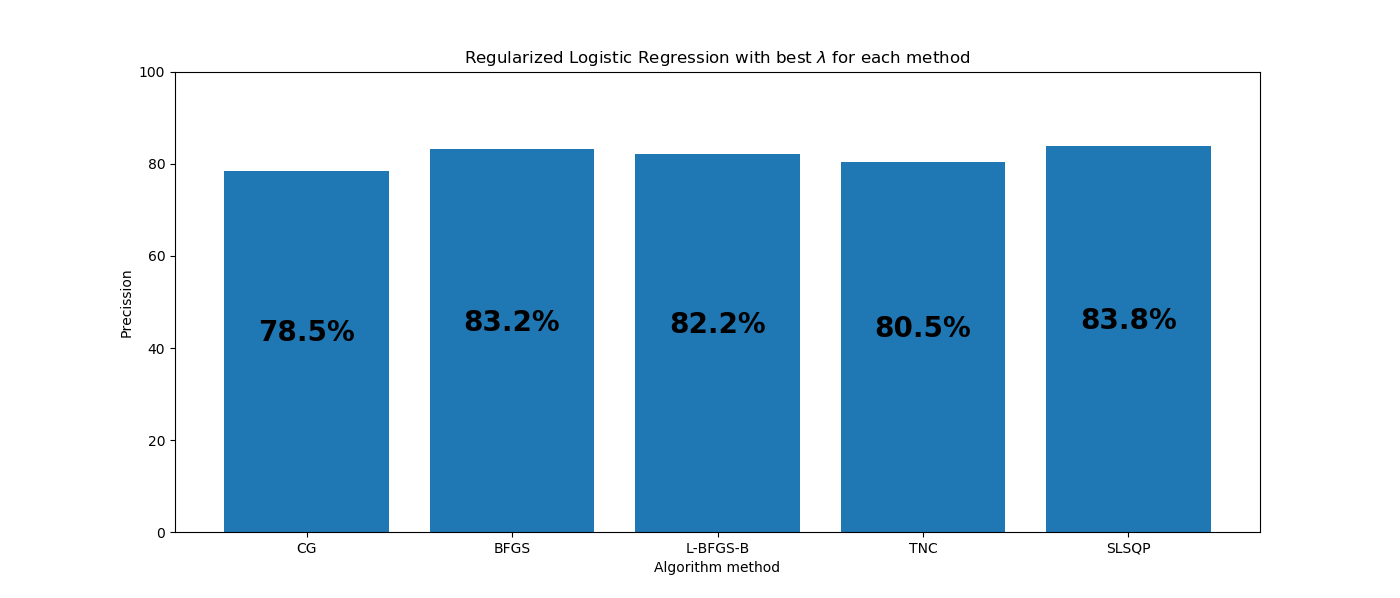


Ilustración 15. Precisión de la regresión logística regularizada con cada algoritmo

##### Redes Neuronales. Error de clasificación según el valor de *lambda.*

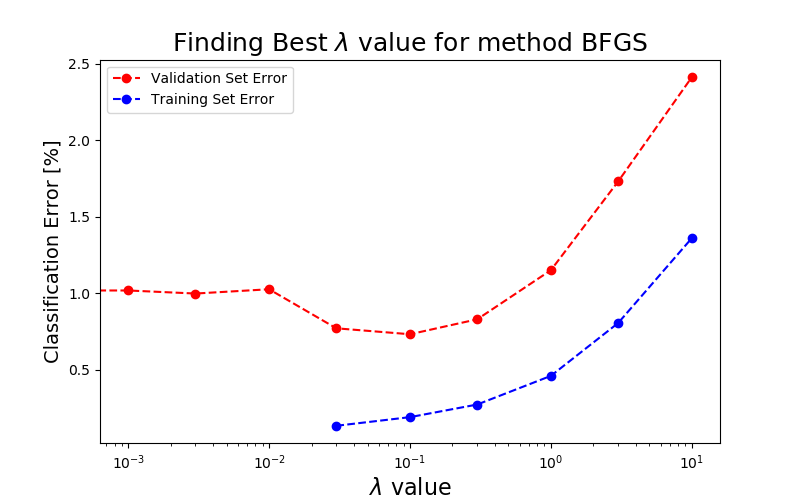


Ilustración 16. Error de clasificación para los distintos valores de lambda usando el algoritmo BFGS

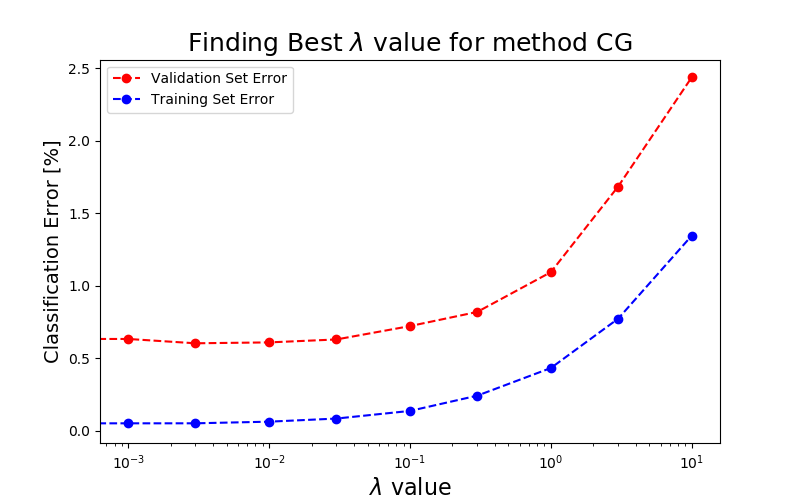


Ilustración 17. Error de clasificación para los distintos valores de lambda usando el algoritmo CG

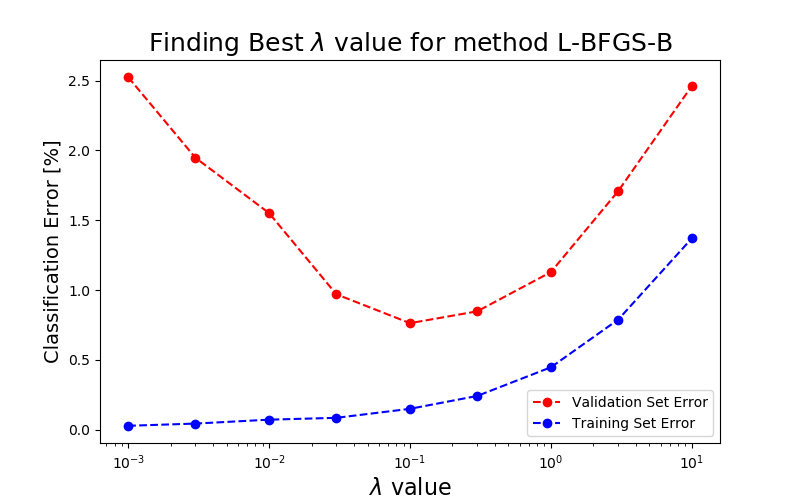


Ilustración 18. Error de clasificación para los distintos valores de lambda usando el algoritmo L-BFGS-B

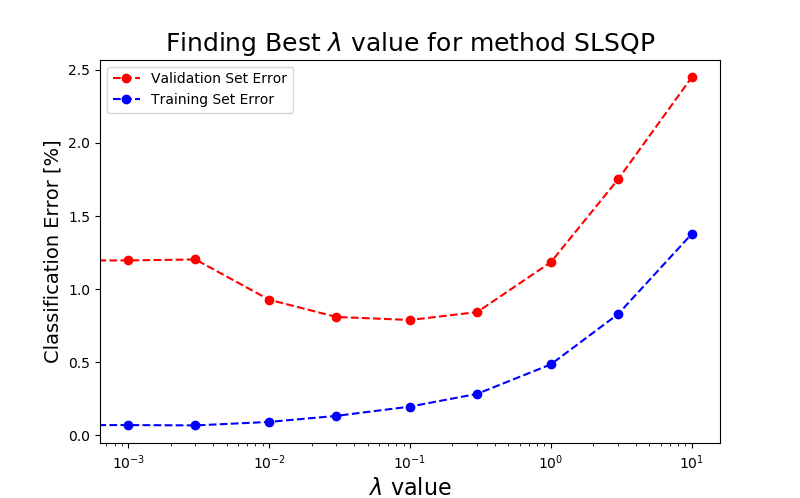


Ilustración 19. Error de clasificación para los distintos valores de lambda usando el algoritmo SLSQP

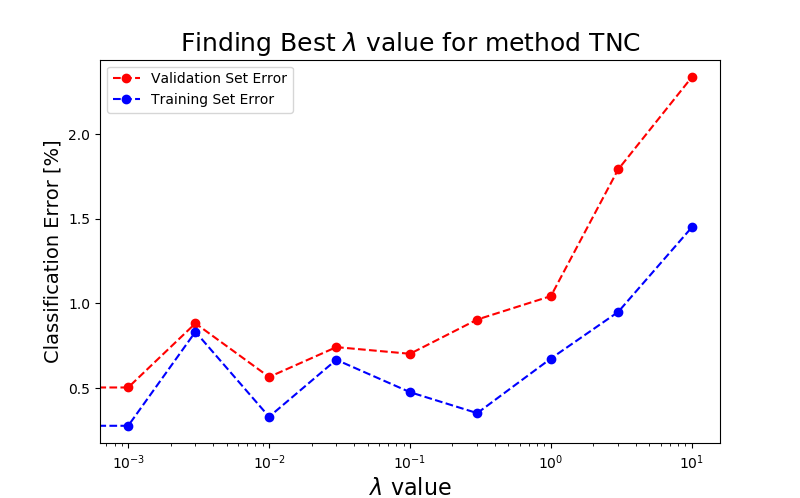


Ilustración 20. Error de clasificación para los distintos valores de lambda usando el algoritmo TNC

##### Redes Neuronales. Precisión de los algoritmos con cada una de las *lambdas*.

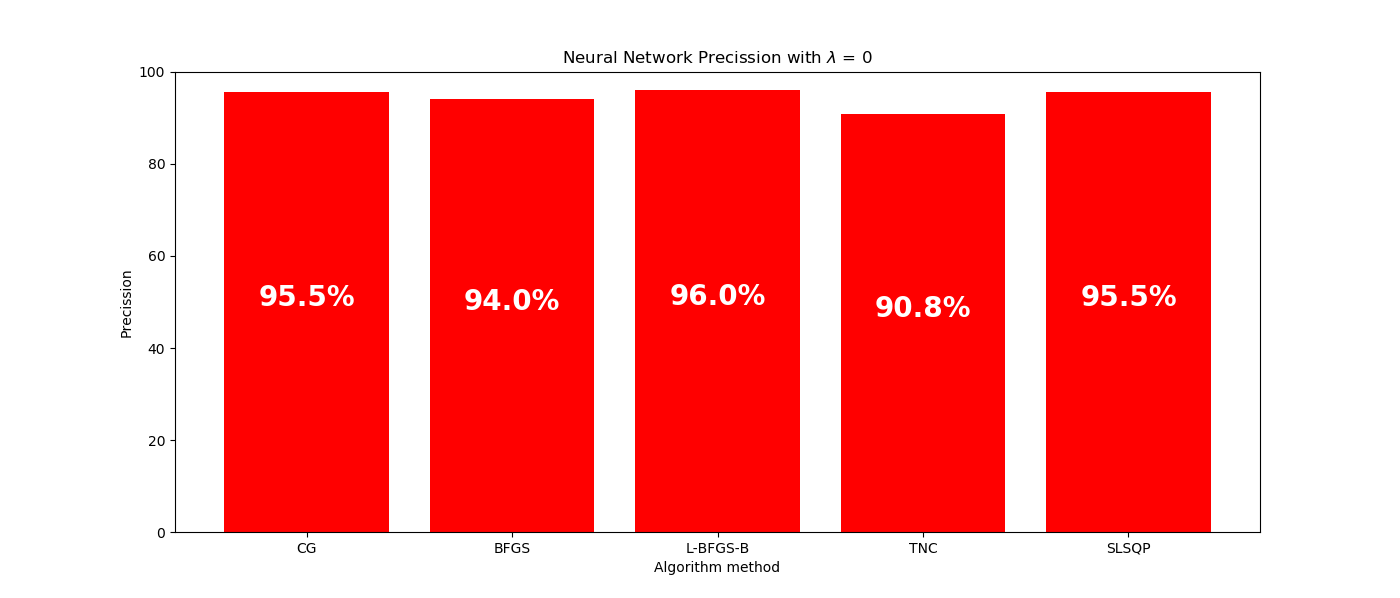


Ilustración 21. Precisión de la red neuronal para lambda 0

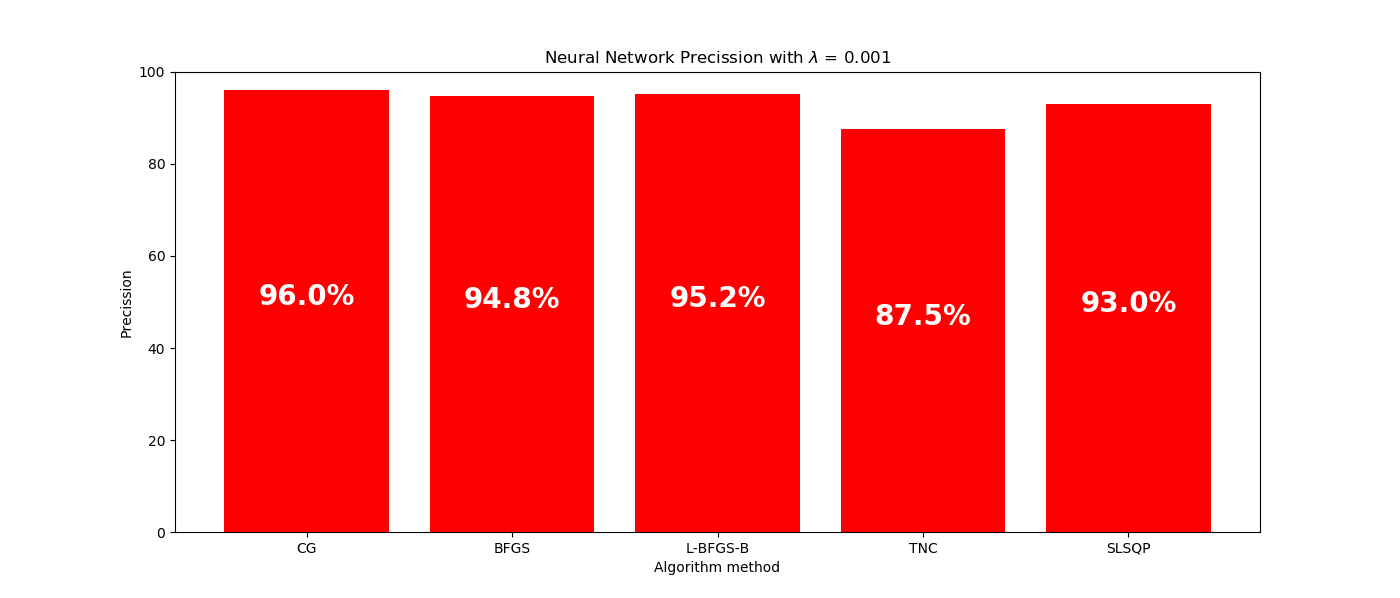


Ilustración 22. Precisión de la red neuronal para lambda 0.001

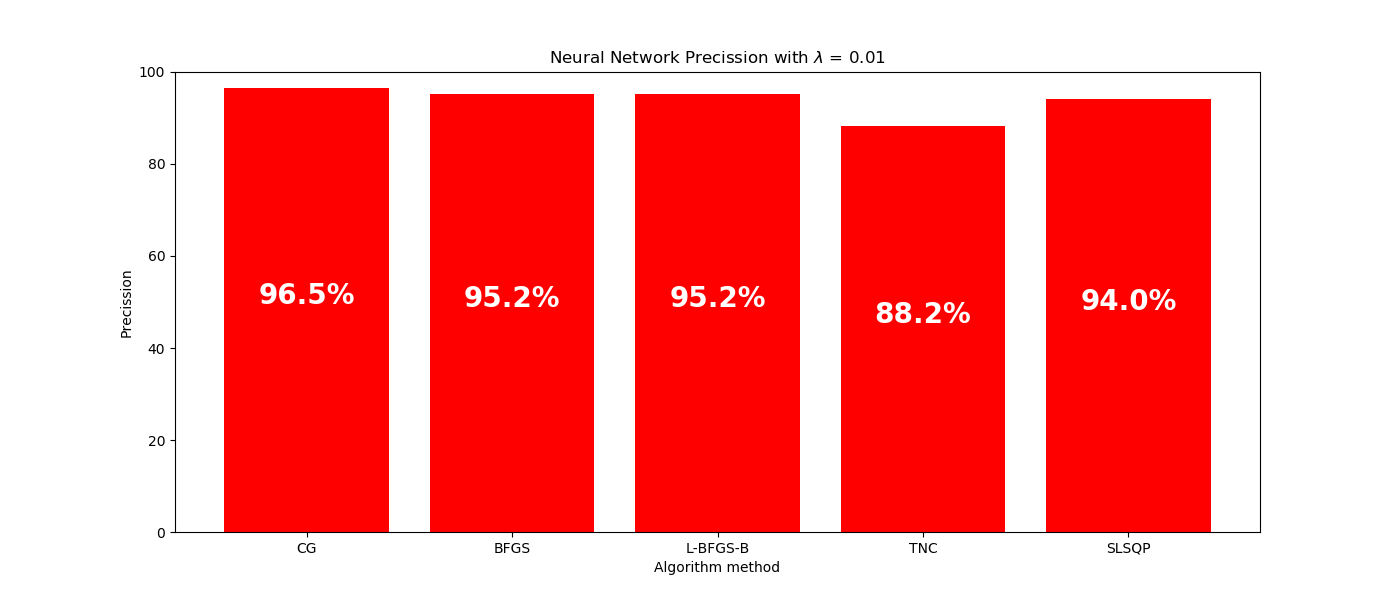


Ilustración 23. Precisión de la red neuronal para lambda 0.01

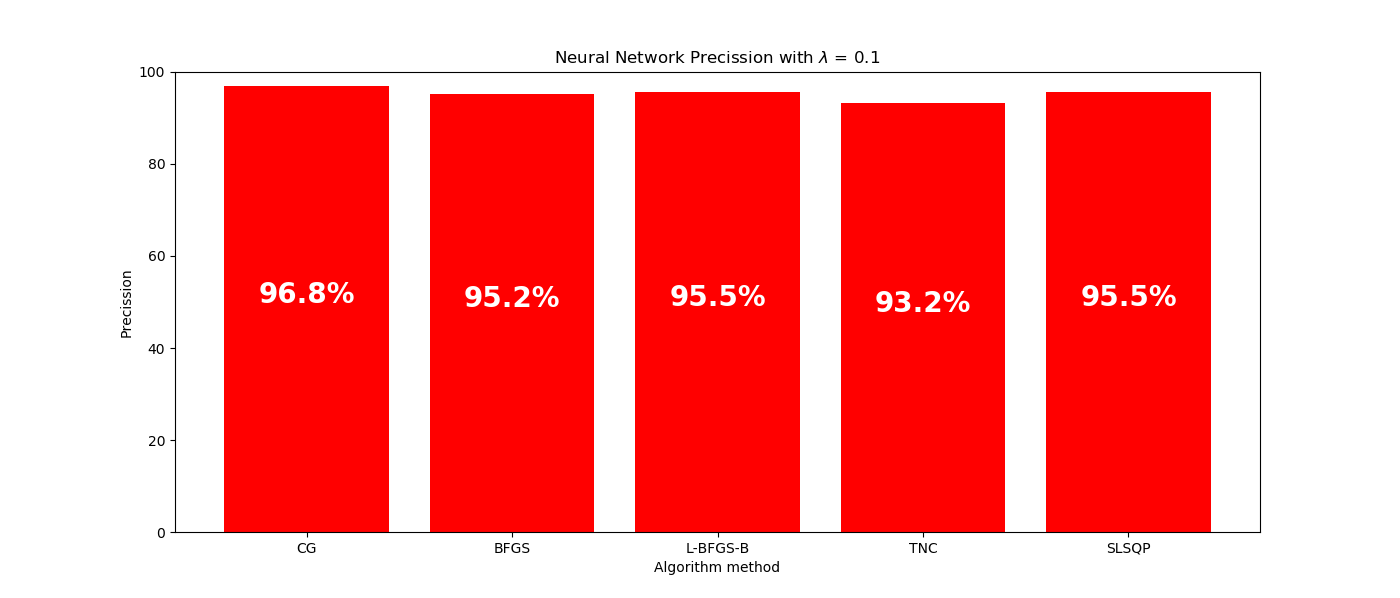


Ilustración 24. Precisión de la red neuronal para lambda 0.1

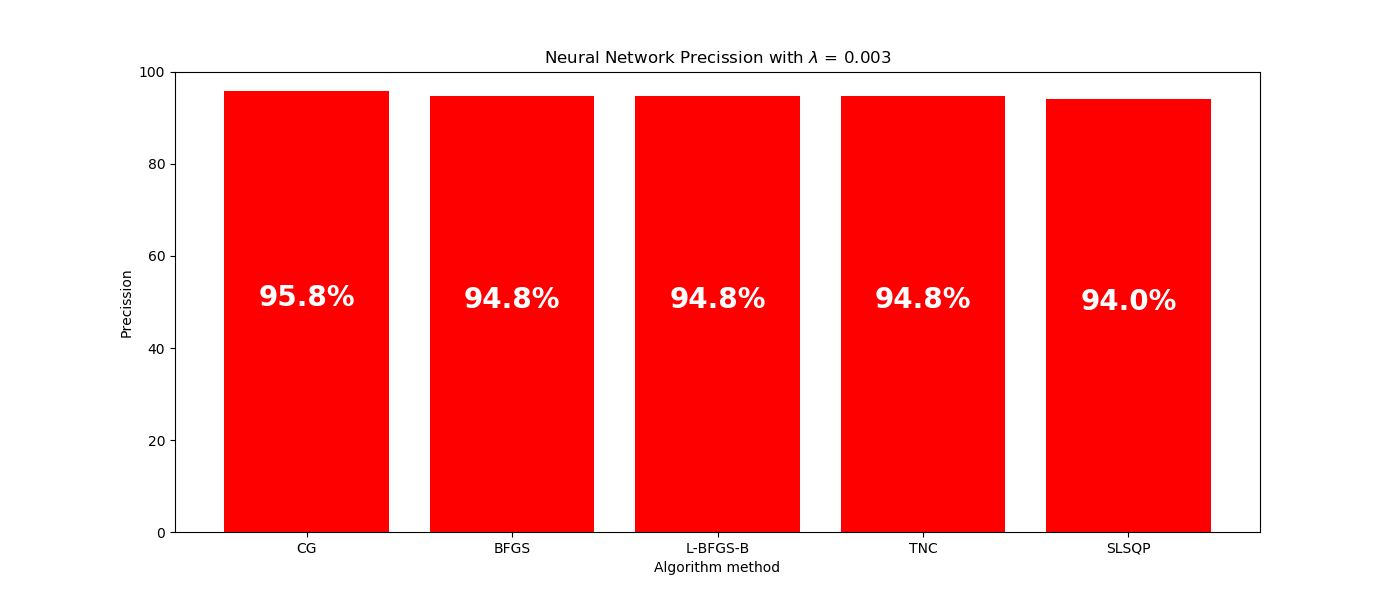


Ilustración 25. Precisión de la red neuronal para lambda 0.003

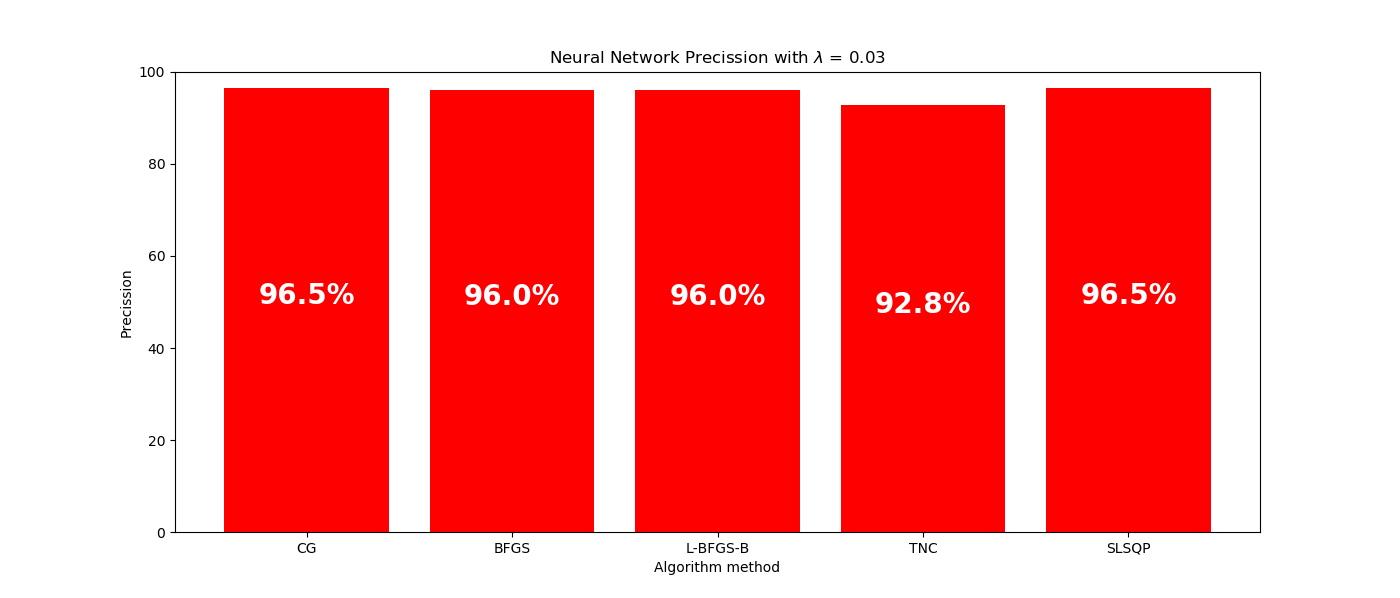


Ilustración 26. Precisión de la red neuronal para lambda 0.03

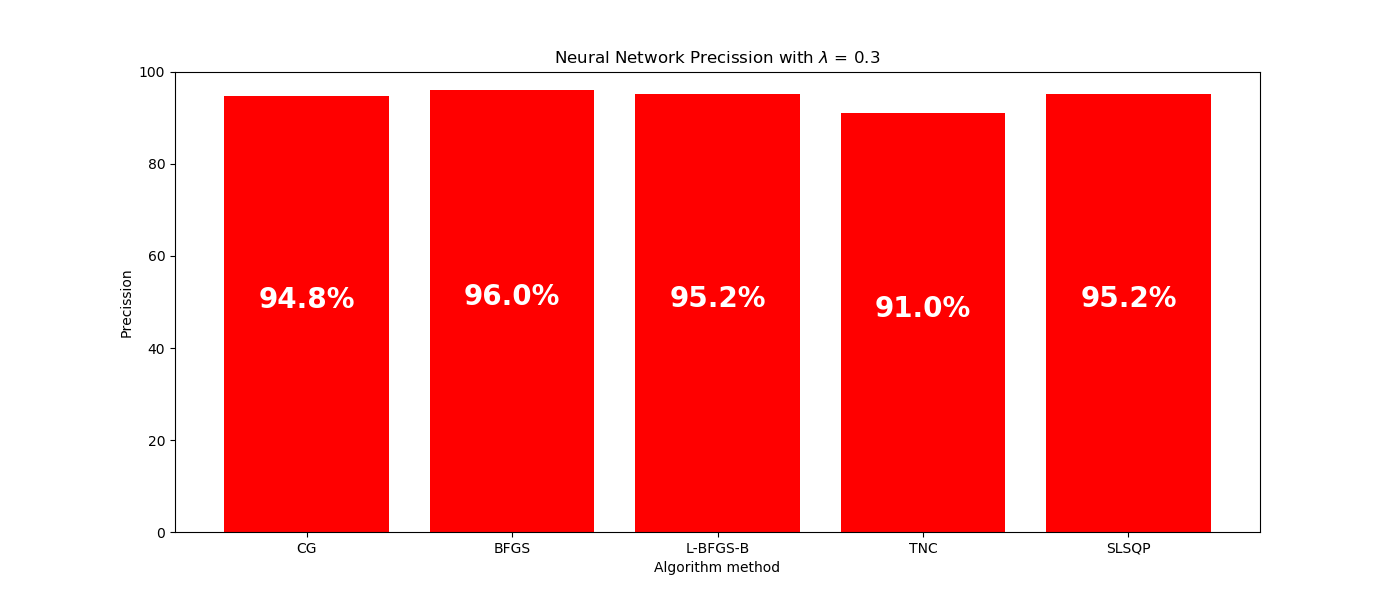


Ilustración 27. Precisión de la red neuronal para lambda 0.3

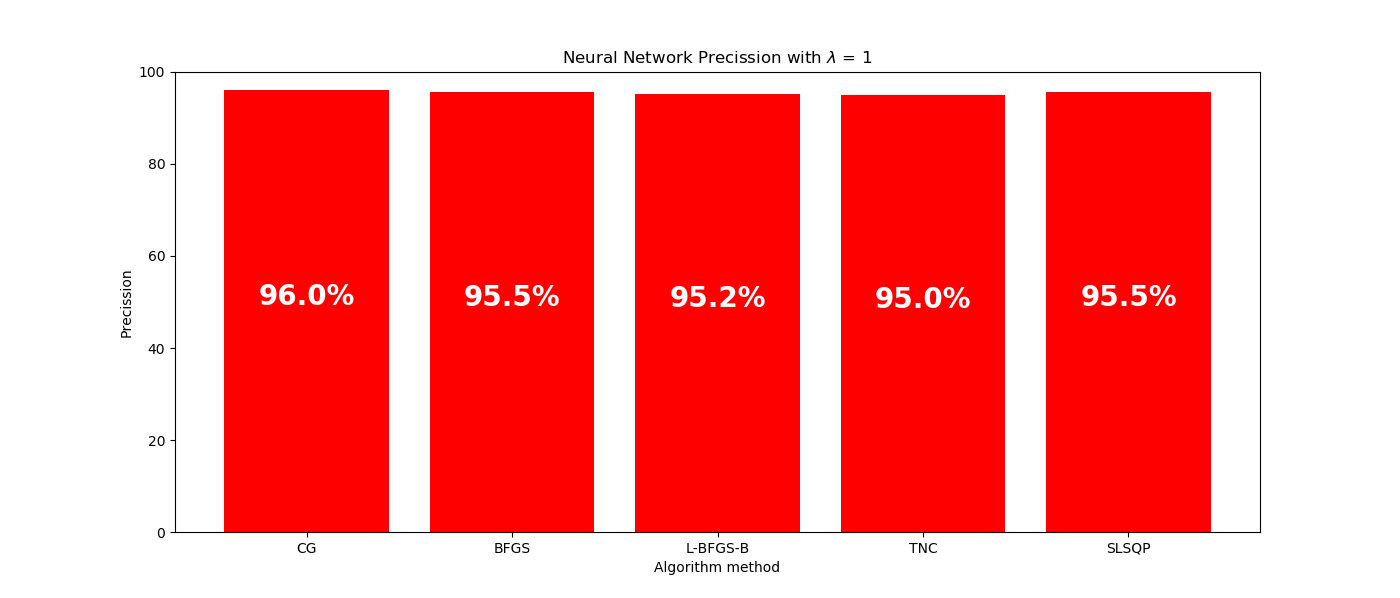


Ilustración 28. Precisión de la red neuronal para lambda 1

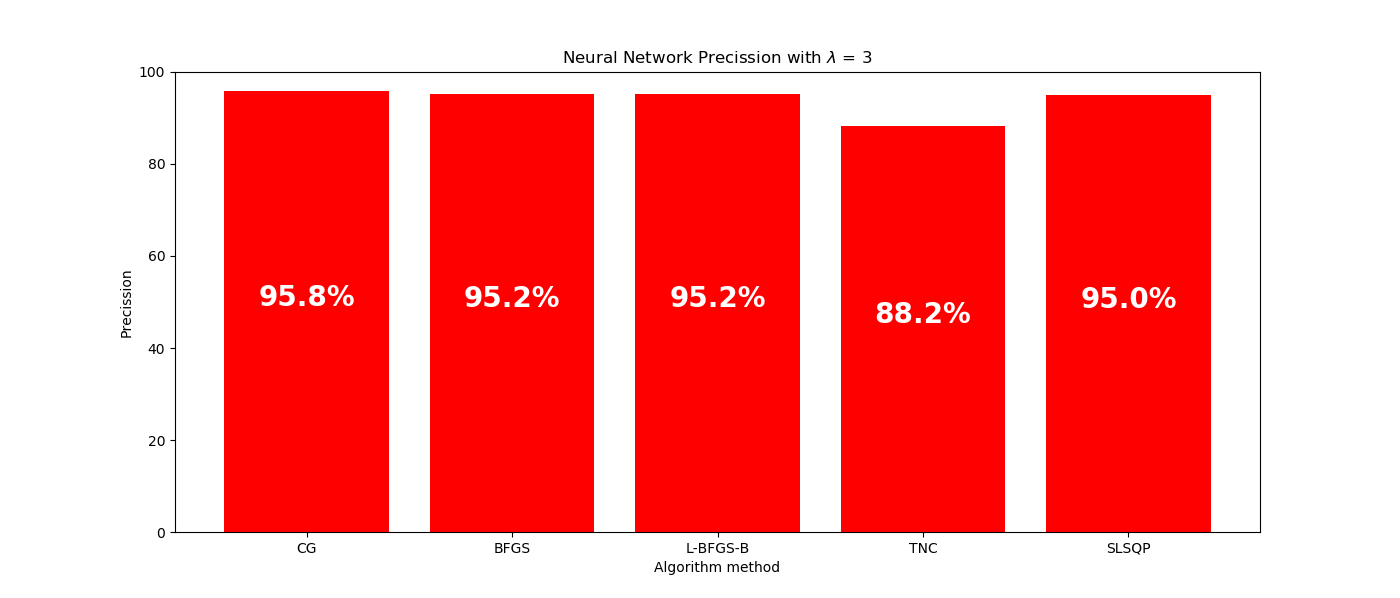


Ilustración 29. Precisión de la red neuronal para lambda 3

##### Redes Neuronales. Precisión de los algoritmos con el mejor valor de *lambda* para cada uno de ellos. Modificando el tamaño de la capa oculta.

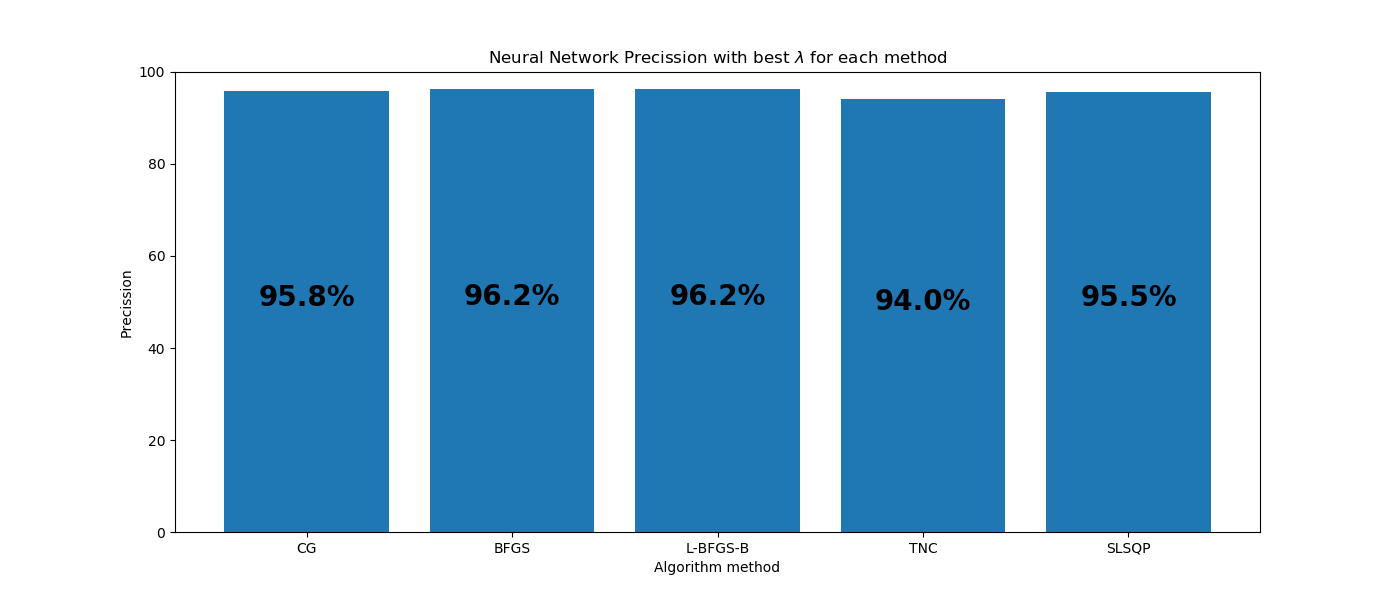


Ilustración 30. Precisión de los algoritmos con una capa oculta de tamaño 8

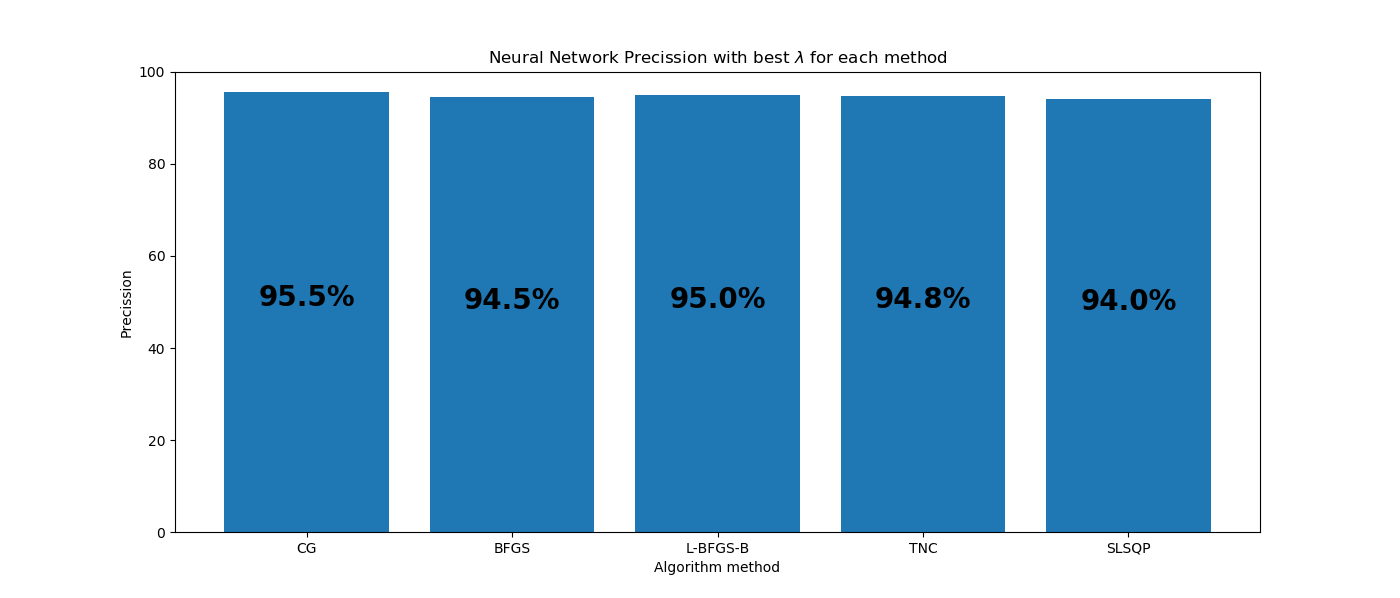


Ilustración 31. Precisión de los algoritmos con una capa oculta de tamaño 16

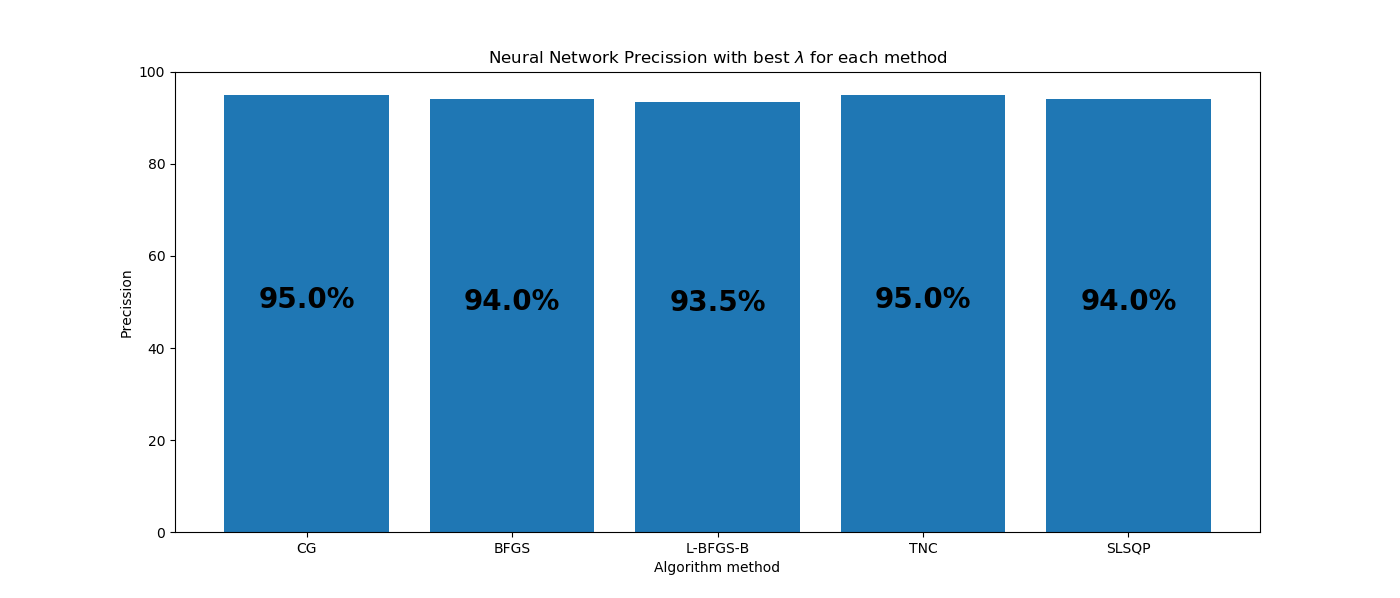


Ilustración 32. Precisión de los algoritmos con una capa oculta de tamaño 32