# Valhalla

August 19, 2024

## 1 Challenge 01: Conversor de Celsius a Valks

**Autor:** Rodolfo Jesús Cruz Rebollar

**Matrícula:** A01368326

**Grupo:** 101

[1]:
```
# Importar librerías para el análisis, manipulación y graficación de datos

import pandas as pd # Pandas para análisis y manipulación de datos

import numpy as np # numpy para realizar operaciones matemáticas y matriciales

import matplotlib.pyplot as plt # Matplotlib para realizar gráficos
```

[2]:
```
# Importar los datos a analizar, leyéndolos del archivo csv

celsius_valks = pd.read_csv("Valhalla23.csv")

# Mostrar primeros 5 registros del dataframe para asegurar que los datos se␣
 ↪hayan importado correctamente

celsius_valks.head()
```

[2]:

| | Celsius | Valks |
|---|---|---|
| 0 | 61.4720 | -139.740 |
| 1 | 70.5790 | -156.600 |
| 2 | -7.3013 | 73.269 |
| 3 | 71.3380 | -165.420 |
| 4 | 43.2360 | -75.835 |

[3]:
```
# Mostrar características generales de la base de datos

celsius_valks.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype
```

```
---  ------   --------------  -----
 0   Celsius  100 non-null    float64
 1   Valks    100 non-null    float64
dtypes: float64(2)
memory usage: 1.7 KB
```

Al observar las características generales de la base de datos, se puede notar que en total se tienen 100 registros de datos con 2 columnas, correspondientes a la temperatura en Celsius y Valks respectivamente, además, se aprecia que ambas columnas de información no presentan datos faltantes, ya que todos los datos en ambas son no nulos, motivo por el cual, al no detectar ninguna inconsistencia o error en la estructura de los datos, a continuación se procederá a separar el conjunto principal de datos en 2 subconjuntos, 1 para el entrenamiento del modelo de regresión lineal y otro para ponerlo a prueba.

## 1.1 Creación del conjunto de entrenamiento y de prueba

### 1.1.1 Conjunto de entrenamiento

[4]:
```
# Importar librería random que contiene funciones para elegir aleatoriamente␣
 ↪elementos de estructuras de datos o generar
# valores aleatorios

import random as rnd

# Generar el conjunto de datos de entrenamiento para el modelo (el 70% de los␣
 ↪datos originales se destinará para entrenamiento)

train_data = celsius_valks.iloc[rnd.choices(range(celsius_valks.shape[0]), k =␣
 ↪round(0.7 * celsius_valks.shape[0])), :]

# Mostrar los primeros registros del subconjunto de datos para entrenar el␣
 ↪modelo

train_data.head()
```

[4]:

| | Celsius | Valks |
|---|---|---|
| 64 | 69.0900 | -140.640 |
| 20 | 45.5740 | -81.557 |
| 77 | 5.1084 | 35.183 |
| 91 | -14.6050 | 91.536 |
| 22 | 64.9130 | -142.020 |

En términos generales, para generar el subconjunto de datos para entrenar el modelo, lo primero que se realizó fue calcular cuántos registros (filas) tiene el conjunto total de datos con ayuda de la función shape que devuelve una tupla con dos elementos, el primero es la cantidad de filas del dataframe y el segundo es la cantidad de columnas del mismo, por lo que en este caso particular, se toma solamente el primer elemento de la tupla devuelta por la función shape (tiene como índice 0) que coresponde a la cantidad de registros del dataframe original, para luego en base a la cantidad de filas, la función range() genera una secuencia de números enteros desde 0 hasta la cantidad de

filas menos 1, por lo que al tener 100 filas, range devolverá una secuencia de enteros desde 0 hasta 99 y dicha secuencia de enteros es recibida por la función choices() del módulo random, la cual escoge aleatoriamente una cierta cantidad de elementos de una lista, especificada en su parámetro k, que en este caso, dicha cantidad k se calcula obteniendo el 70% de la cantidad de filas del dataframe original y luego redondeando dicho resultado al entero más cercano, esto con el objetivo de evitar tener errores de formato numérico al momento de ejecutar el renglón de código en caso de que al calcular el porcentaje de datos para entrenamiento, se obtenga una cantidad con decimales. Además de lo anterior, después de calcular el valor k y definir la secuencia de enteros aleatorios para la función choices(), se procede a que la función choices() elige aleatoriamente k valores enteros de la secuencia definida previamente y envía dicha lista de enteros aleatorios al método iloc para decirle que extraiga las filas especificadas en la lista que recibe dicho método , mismas que se encuentran identificadas por su índice numérico (son los enteros aleatorios obtenidos antes) y a su vez también extraiga todas las columnas que conforman a dichas filas. Es importante recalcar que el método iloc identifica filas y columnas mediante su índice numérico en lugar de sus nombres.

### 1.1.2 Conjunto de prueba

[5]:
```
# El conjunto de prueba estará conformado por todos los datos que no fueron␣
 ↪seleccionados de forma aleatoria al momento de
# generar el conjunto de entrenamiento

# Obtener los índices numéricos de las filas que no fueron seleccionadas al␣
 ↪generar el conjunto de entrenamiento

# la función lambda en conjunto con la función filter, verifican si cada índice␣
 ↪de fila del dataframe completo
# se encuentra entre los índices de fila del subconjunto de entrenamiento, de␣
 ↪no ser así, se selecciona dicho índice para
# incluir la fila correspondiente al mismo, al conjunto de prueba, esto␣
 ↪garantiza que el subconjunto de prueba tenga datos que
# el modelo aún no haya visto y finalmente se incluyen los índices filtrados en␣
 ↪una nueva lista

filas_no_elegidas = list(filter(lambda x: x not in train_data.index.values,␣
 ↪celsius_valks.index.values))

# Extraer las filas que no estén en el subconjunto de entrenamiento con todas␣
 ↪sus columnas

test_data = celsius_valks.iloc[filas_no_elegidas, :]

# Mostrar los primeros registros del conjunto de prueba para asegurar que los␣
 ↪datos estén bien estructurados

test_data.head()
```

[5]:

| | Celsius | Valks |
|---|---|---|
| 0 | 61.472 | -139.740 |
| 1 | 70.579 | -156.600 |
| 5 | -10.246 | 83.437 |
| 7 | 34.688 | -55.108 |
| 8 | 75.751 | -182.820 |

**Nota:** la cantidad de registros extraídos para el conjunto de prueba, corresponde al 30% de la cantidad de registros de la base de datos original (30 registros de 100).

### 1.2 Regresión lineal con gradiente descendente

Para implementar un modelo de regresión lineal con gradiente descendente para resolver la problemática planteada, en primera instancia es necesario tomar en consideración aquellas ecuaciones en las que está basado el funcionamiento del modelo en sí mismo, entre las que se encuentran: la función de costo denotada como J, las funciones correspondientes a las derivadas parciales respecto a $\theta_0$ y $\theta_1$ de dicha función de costo, además de aquellas otras ecuaciones correspondientes al cálculo tanto del parámetro $\theta_0$ como del parámetro $\theta_1$, mismos que irán actualizándose a medida que el algoritmo necesite una mayor cantidad de iteraciones para encontrar el punto de convergencia, mismo que se define como el momento en el cual la función de costo J alcanza su mínimo valor posible para los datos en cuestión.

Debido a lo anterior, a continuación se mencionan las ecuaciones que se utilizarán para la implementación efectiva del modelo de regresión lineal:

**Función de costo J:**

$J_\theta = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)^2$

Donde $h_\theta(x_i) = \theta_0 + \theta_1 x_i$

**Derivadas parciales de J:**

$\frac{\partial J}{\partial \theta_0} = \frac{1}{n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)$

$\frac{\partial J}{\partial \theta_1} = \frac{1}{n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)x_i$

**Cálculo de $\theta_0$ y $\theta_1$ mediante gradiente descendente:**

$\theta_0 = \theta_0 - \alpha\frac{1}{n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)$

$\theta_1 = \theta_1 - \alpha\frac{1}{n}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)x_i$

Para definir la función del método de gradiente descendente a continuación, los parámetros $\theta_0$ y $\theta_1$ tendrán ambos un valor inicial de 1, esto debido principalmente a que al asignar inicialmente valores unitarios a dichos parámetros del modelo, se comienza el algoritmo con un modelo básico en el sentido de que al comienzo de todo el proceso iterativo, al no saber todavía con certeza cuáles serán los valores adecuados para los parámetros del modelo, el hecho de asignar un valor unitario a éstos garantiza que al iniciar el proceso, tanto la ordenada al origen como el coeficiente de la variable x tengan una cierta contribución, peso o influencia en las primeras predicciones derivadas del algoritmo, esto mientras se calculan nuevos valores para los parámetros del modelo a medida que avanzan las iteraciones del algoritmo, por lo que después de la primera iteración del algoritmo, los valores unitarios iniciales se reemplazarán por los nuevos valores calculados en dicha iteración

y así sucesivamente, por lo cual, se seleccionaron valores iniciales de 1 para los $\theta$ solamente para comenzar el algoritmo con un "modelo por default".

Además de lo anterior, para implementar el modelo de regresión lineal con gradiente descendente se necesita definir un parámetro adicional llamado $\alpha$ que será la tasa de aprendizaje del modelo (rapidez con la que aprende el modelo a identificar patrones en los datos) y otro parámetro conocido como $\epsilon$ que se refiere al límite máximo de error permitido para el modelo, es decir, el criterio que se utilizará para definir si la función de costo J converge, por lo que en caso de que llegue un momento en el cual el valor de ambas derivadas de la función J sean inferiores al valor de $\epsilon$, se alcanzará la convergencia del algoritmo y en ese punto terminará todo el proceso iterativo, es decir, terminará el algoritmo en sí. Por otro lado, utilizaremos una tasa de aprendizaje $\alpha = 0.003$ debido a que es un valor que no es excesivamente pequeño pero tampoco muy grande, esto en el sentido de que garantizará que el algoritmo aprenda a un ritmo moderado, es decir, que no sea excesivamente rápido o lento, dado que en caso de que dicho valor $\alpha$ sea muy pequeño, se corre el riesgo de que el algoritmo al aprender a un ritmo muy lento, necesite demasiadas iteraciones, lo cual implicaría que el algoritmo demore mucho tiempo en proporcionar una respuesta a tal punto de que dicha respuesta al no llegar a tiempo, retrase los procesos por ejemplo de alguna empresa, o industria y perjudicando su capacidad para operar de forma eficente, por lo que en resumen un valor de $\alpha$ muy pequeño puede provocar que el algoritmo se vuelva prácticamente ineficiente sobretodo en situaciones en las que se requiere una respuesta a la brevedad posible. Sin embargo, si se elige un valor de $\alpha$ grande, se corre el riesgo de que durante el proceso iterativo del algoritmo, se pase por alto el punto mínimo en el que la función J alcanza un valor inferior al $\epsilon$, provocando que al final de todo el algoritmo, el modelo de regresión lineal resultante no sea el que mejor se ajuste a los datos en cuestión, por lo que en pocas palabras, un $\alpha$ grande reduciría la capacidad del algoritmo para encontrar el mejor modelo posible para los datos además de que el modelo derivado del mismo no arrojaría predicciones de temperatura suficientemente confiables, lo que podría ocasionar que en vez de resolver la crisis del calentamiento global mencionada en el challenge 1, el planeta Tierra se podría calentar o enfriar en exceso y por consiguiente afectar negativamente a la vida en la Tierra.

Adicionalmente, también cabe mencionar que en cuanto al parámetro de error máximo $\epsilon$, éste mismo tendrá un valor de 0.01 para el problema en cuestión, ya que entre más pequeño sea el valor de éste parámetro, el algoritmo buscará obtener un modelo en su gran mayoría confiable en cuanto a las predicciones que arroje, sin embargo, $\epsilon$ tampoco puede ser 0, debido a que por la propia naturaleza de los datos involucrados, tampoco resulta posible encontrar algún modelo que prediga correctamente la temperatura en Valks para absolutamente todos los datos en grados celsius, por lo que 0 no es un valor adecuado para el parámetro $\epsilon$. No obstante si en cambio el valor de $\epsilon$ es grande, tampoco es recomendable dado que estaríamos permitiendo que el modelo resultante del algoritmo tenga mayor margen de error, lo cual puede causar que las predicciones derivadas del mismo no tengan un grado de confiabilidad suficiente para convertir grados celsius a valks, lo que de nuevo puede llevar a una toma de decisiones errónea y nuevamente eso agravaría la crisis del cambio climático que se desea resolver en la problemática abordada, por lo que 0.01 resulta ser una elección que no es un valor excesivamente pequeño pero tampoco muy grande, por lo que resulta adecuado como valor para el parámetro de error máximo admisible $\epsilon$.

[15]:
```
# Crear una serie de pandas que contenga ambos valores de theta a lo largo del␣
↪algoritmo

# Incialmente ambos hiperparámetros theta tendrán un valor de 1
```

```
theta = pd.Series([1, 1], index = ["theta0", "theta1"])

# Definir la tasa de aprendizaje alpha

alpha = 0.003

# Definir la máxima proporción permitida de error épsilon

epsilon = 0.01
```

### 1.2.1 Función de hipótesis h

[16]:
```
# Establecer una función anónima lambda para evaluar los datos de la variable independiente (Celsius) en la función de hipótesis

h = lambda x: theta["theta0"] + theta["theta1"] * x
```

### 1.2.2 Función de costo J

[17]:
```
# Definir la función de costo que recibe 2 parámetros de entrada: número de registros del dataframe y datos de entrenamiento

def J_function(n, data):

    # Calcular el valor de la función J para los datos: primero los datos de temperatura en grados Celsius se multiplican por
    # el valor del parámetro theta 1 y después a dichos resultados se les suma el valor de theta 0, para después a dichos
    # resultdos restarles los datos de temperatura en Valks, elevar los valores resultantes al cuadrado, sumarlos y finalmente
    # multiplicar el resultado de la suma por 1/2n

    J  = (1 / (2 * n)) * (((h(data["Celsius"]) - data["Valks"]) ** 2).sum())

    # Devolver el valor de la función J

    return J
```

### 1.2.3 Derivadas parciales de J

[18]:
```
# Función para calcular derivada parcial de J respecto a theta0 que recibe como parámetros de entrada: número de datos de entre-
# namiento (n) y los datos de entrenamiento (data)

def J_partial_theta0(n, data):
```

```
    # Calcular la derivada de J respecto a theta0, primero calculando los␣
 ↪residuos del modelo
    # (Valks predichos con función h menos Valks reales), luego sumando los␣
 ↪residuos obtenidos por muestra y
    # por último, multiplicando dicha suma por 1/n

    derivada_theta_0 = (1 / n) * ((h(data["Celsius"]) - data["Valks"]).sum())

    # Devolver el valor de la derivada de J respecto a theta0

    return derivada_theta_0
```

[19]:
```
# Función que calcula el valor de la derivada de J respecto a theta1,␣
 ↪recibiendo como parámetros de entrada: cantidad de datos
# de entrenamiento (n) y los datos de entrenamiento en sí (data)

def J_partial_theta1(n, data):

    # Calcular valor de la derivada de J respecto a theta1, primero calculando␣
 ↪para cada registro de datos, la diferencia entre
    # los Valks predichos con función h menos los Valks reales, después␣
 ↪multiplicando las diferencias por los datos de celsius
    # luego sumando los productos obtenidos y finalmente multiplicando el␣
 ↪resultado de la suma por 1/n

    derivada_theta_1 = (1 / n) * (((h(data["Celsius"]) - data["Valks"]) *␣
 ↪data["Celsius"]).sum())

    # Devolver el valor de la derivada de J respecto a theta1

    return derivada_theta_1
```

### 1.2.4 Función de gradiente descendente

[20]:
```
# Función para calcular los parámetros adecuados para el modelo de regresión␣
 ↪lineal mediante el método del gradiente descendente

"""La función recibe 4 argumentos de entrada: datos para entrenamiento, tasa de␣
 ↪aprendizaje alpha, límite máximo de error
   permitido para definir la convegencia del algoritmo y el número máximo de␣
 ↪iteraciones para generar el modelo (100)."""

# El parámetro i = 500 indica que se entrenará el modelo con 500 iteraciones␣
 ↪(más del mínimo solicitado de 100 iteraciones)
```

```
def gradiente_descendente(data, alpha, epsilon, i = 500):

    # Longitud de conjunto de datos para entrenamiento

    n = len(data)

    # Definir variable booleana convergencia que indicará si se alcanza o no la␣
 ↪convergencia del algoritmo

    convergencia = False # se inicializa en false dado que aún no se encuentra␣
 ↪ningún modelo para los datos

    # Inicializar en 1 la variable k que contará el número de iteraciones del␣
 ↪ciclo while

    k = 1

    # Ejecutar el algoritmo mientras que no se alcance el máximo de iteraciones␣
 ↪y no se encuentre convergencia
    # Esto con el propósito de que en caso de que el algoritmo converga,␣
 ↪entonces se detenga el algoritmo aunque no se
    # alcance el número máximo de iteraciones establecido. Además, el ciclo␣
 ↪también se detendrá en caso de que a pesar de
    # que el algoritmo no converga, se llegue a la cantidad máxima establecida␣
 ↪de iteraciones, lo cual será solamente un paro
    # de emergencia en el caso de que no suceda la convergencia y para que el␣
 ↪algoritmo no itere infinitamente, sino que arroje
    # una respuesta en el máximo de iteraciones establecido aunque no sea el␣
 ↪modelo que mejor se ajuste a los datos

    while (k <= i) & (convergencia == False):

        # Calcular el valor de J (función de costo)

        J = J_function(n, data)

        # Calcular el valor de la derivada de J respecto a theta0

        deriv_theta_0 = J_partial_theta0(n, data)

        # Calcular el valor de la derivada de J respecto a theta1

        deriv_theta_1 = J_partial_theta1(n, data)

        # Calcular el valor actualizado del hiperparámetro theta0
```

```
        theta["theta0"] -= alpha * deriv_theta_0

        # Calcular el valor actualizado del hiperparámetro theta1

        theta["theta1"] -= alpha * deriv_theta_1

        # Verificar si el valor de las derivadas es inferior al límite máximo␣
↪epsilon

        if (deriv_theta_0 < epsilon) & (deriv_theta_1 < epsilon):

            # De ser así, se habrá encontrado la convergencia del algoritmo␣
↪para los datos

            convergencia = True

        # Actualizar el valor del número de iteración k

        k += 1

    # Desplegar el modelo de regresión lineal calculado para los datos

    print(f'El modelo para los datos es: Valks = {theta["theta0"]} +␣
↪{theta["theta1"]} * Celsius')

    # Mostrar el valor óptimo de la función de costo J

    print(f'Valor J para el conjunto de datos: J = {J}')
```

[21]:
```
# Crear y entrenar el modelo de regresión lineal con gradiente descendente

# El modelo se crea en base a los datos de entrenamiento, por lo que se envía␣
↪el dataset de entrenamiento como argumento a la
# función de gradiente_descendente() además del valor de alpha y de epsilon

# Nota: el resto de los argumentos ya no es necesario enviarlos porque ya␣
↪tienen asignado un valor por defecto

# Además, el modelo se entrenará con 500 iteraciones como se establece␣
↪anteriormente

gradiente_descendente(train_data, alpha, epsilon)
```

```
El modelo para los datos es: Valks = 2.4247347523859704 + 78.20939979482725 *
Celsius
Valor J para el conjunto de datos: J = 253678.71585944213
```

### 1.3 Prueba del modelo implementado

[22]:
```
# Calcular el modelo de regresión lineal que se ajuste a los datos del␣
 ↪subconjunto de prueba

gradiente_descendente(test_data, alpha, epsilon)
```

```
El modelo para los datos es: Valks = 32.945394258120174 + 1847.6431462425403 *
Celsius
Valor J para el conjunto de datos: J = 143718546.57103246
```

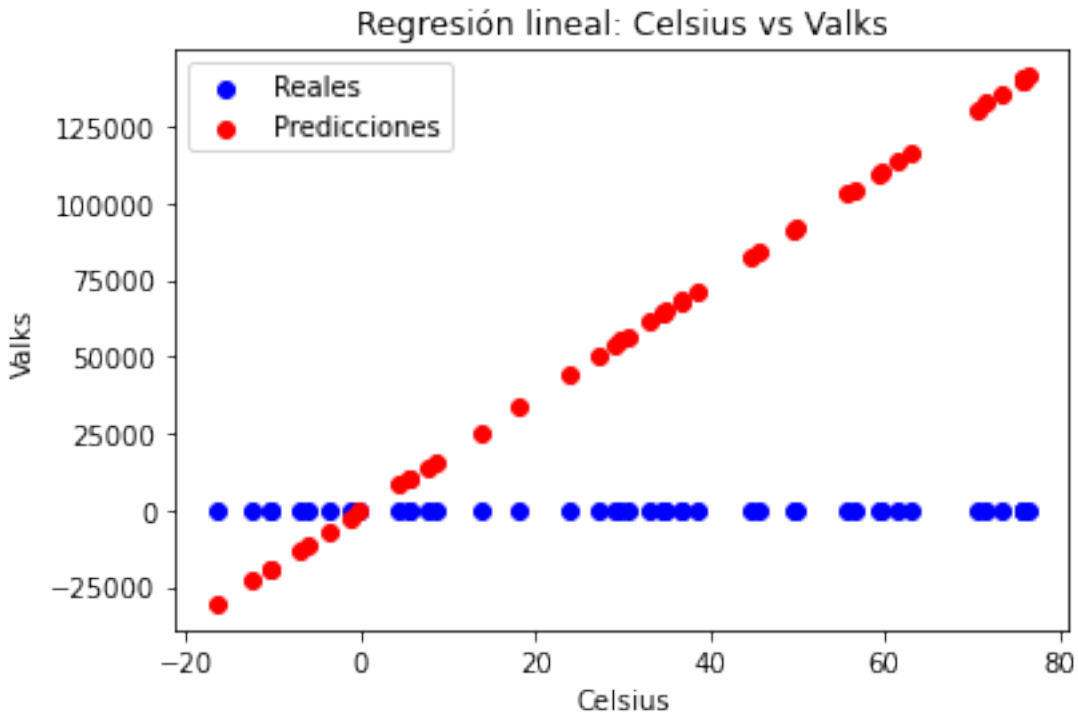
[29]:
```
# Calcular las predicciones para los Valks a partir de los celsius en base al␣
 ↪modelo arrojado para los datos de prueba

"""Nuevamente usar la función de hipótesis h para realizar las predicciones␣
 ↪(los coeficientes calculados por el algoritmo se
   encuentran en la lista original de valores theta"""

pred_valks = h(test_data["Celsius"])

# Mostrar las primeras 10 predicciones en Valks para los datos de prueba

pred_valks.head(10)
```

[29]:
```
0     113611.264880
1     130437.751013
5     -18898.006282
7      64123.990851
8     139993.761365
9     141357.322007
12    139930.941498
18    109452.220158
19    140359.594708
23    135648.104685
Name: Celsius, dtype: float64
```

#### 1.3.1 Graficar los Valks reales vs los predichos

[41]:
```
# Graficar los Valks reales del conjunto de prueba contra los Valks predichos␣
 ↪por el modelo

# El parámetro c hace referencia a que los datos de Valks reales se graficarán␣
 ↪en color azul y los predichos en color rojo

plt.scatter(test_data["Celsius"], test_data["Valks"], color = "blue", label =␣
 ↪"Reales") # Gráfico de los datos reales
```

```
plt.scatter(test_data["Celsius"], pred_valks, color = "red", label =␣
 ↪"Predicciones") # Gráfico de los datos predichos

plt.xlabel("Celsius") # Graficar grados celsius en el eje horizontal

plt.ylabel("Valks") # Graficar Valks en el eje vertical

plt.title("Regresión lineal: Celsius vs Valks") # Título del gráfico

plt.legend() # Agregar leyenda al gráfico

plt.show() # Mostrar gráfico de datos reales vs predicciones del modelo
```




En términos generales, en la gráfica anterior se observa que las predicciones realizadas por el modelo implementado se alejan significativamente de los datos reales en Valks del subconjunto de datos de prueba, lo cual se debe principalmente al hecho de que el coeficiente $\theta_1$ que se encuentra multiplicando a la variable predictora del modelo (temperatura en grados celsius), aún sigue siendo considerablemente grande, esto a pesar de que se realizó el entrenamiento del modelo con un total máximo de 500 iteraciones. Adicionalmente, esta diferencia notable entre los datos reales y los predichos también se atribuye a que los datos empleados para la generación del modelo también poseen una cierta variación considerable entre ellos en cuanto a su naturaleza, lo cual ocasiona que al momento de utilizarlos para aplicar el algoritmo, la variación existente entre los nuevos datos predichos también sea considerable, no obstante, dado que al momento de ejecutar el algoritmo, éste mismo no arroja ningún error durante su ejecución sobre el conjunto de datos en cuestión, además

de que también logra converger después de una cierta cantidad de iteraciones, se concluye que el algoritmo tiene en general un desempeño mayormente adecuado para predecir la equivalencia en Valks de los datos de temperatura en celsius, pertenecientes al dataset de prueba, por lo que a pesar de que los datos predichos se encuentren alejados de los datos verdaderos, el algoritmo presenta un adecuado desempeño en cuanto a su capacidad de funcionamiento para calcular modelos predictivos que intenten explicar la variabilidad presente en los datos reales, además de que el alejamiento entre datos reales y predichos no se atribuye a algún fallo en la implementación del modelo, sino más bien a la variabilidad de los datos producida a su vez por la propia naturaleza de los mismos (son datos grandes en cuanto a su valor numérico).