Implement Neural Network (or Logistic Regression) From Scratch

Predicting if a person would buy life insurnace based on his age using logistic regression

Above is a binary logistic regression problem as there are only two possible outcomes (i.e. if person buys insurance or he/she doesn't).

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
```

```
df = pd.read_csv("insurance_data.csv")
df.head()
```


	age	affordibility	bought_insurance
0	22	1	0
1	25	0	0
2	47	1	1

Split train and test

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df[["age","afford
```

Preprocessing: Scale the data so that both age and affordibility are in same scaling range

```
X_train_scaled = X_train.copy()
X_train_scaled['age'] = X_train_scaled['age'] / 100

X_test_scaled = X_test.copy()
X_test_scaled['age'] = X_test_scaled['age'] / 100
```

Model Building: First build a model in keras/tensorflow and see what weights and bias values it comes up with. We will than try to reproduce same weights and bias in our plain python implementation of gradient descent. Below is the architecture of our simple neural network

```
model = keras.Sequential([
    keras.layers.Dense(1, input_shape=(2,), activation='sigmoid', ker
])

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(X_train_scaled, y_train, epochs=5000)
```

```
Epoch 1/500
1/1 [==============================] - 0s 482ms/step - loss: 0.7113 - accuracy: 0.5000
Epoch 2/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7110 - accuracy: 0.5000
Epoch 3/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7106 - accuracy: 0.5000
Epoch 4/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7102 - accuracy: 0.5000
Epoch 5/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7098 - accuracy: 0.5000
Epoch 6/500
```



```
1/1 [==============================] - 0s 9ms/step - loss: 0.7094 - accuracy: 0.5000
Epoch 7/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7091 - accuracy: 0.5000
Epoch 8/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7087 - accuracy: 0.5000
Epoch 9/500
1/1 [==============================] - 0s 12ms/step - loss: 0.7083 - accuracy: 0.5000
Epoch 10/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7079 - accuracy: 0.5000
Epoch 11/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7076 - accuracy: 0.5000
Epoch 12/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7072 - accuracy: 0.5000
Epoch 13/500
1/1 [==============================] - 0s 12ms/step - loss: 0.7068 - accuracy: 0.5000
Epoch 14/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7065 - accuracy: 0.5000
Epoch 15/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7061 - accuracy: 0.5000
Epoch 16/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7057 - accuracy: 0.5000
Epoch 17/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7054 - accuracy: 0.5000
Epoch 18/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7050 - accuracy: 0.5000
Epoch 19/500
1/1 [==============================] - 0s 9ms/step - loss: 0.7046 - accuracy: 0.5000
Epoch 20/500
1/1 [==============================] - 0s 12ms/step - loss: 0.7043 - accuracy: 0.5000
Epoch 21/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7039 - accuracy: 0.5000
Epoch 22/500
```

```
1/1 [==============================] - 0s 10ms/step - loss: 0.7035 - accuracy: 0.5000
Epoch 23/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7032 - accuracy: 0.5000
Epoch 24/500
1/1 [==============================] - 0s 12ms/step - loss: 0.7028 - accuracy: 0.5000
Epoch 25/500
1/1 [==============================] - 0s 18ms/step - loss: 0.7025 - accuracy: 0.5000
Epoch 26/500
1/1 [==============================] - 0s 16ms/step - loss: 0.7021 - accuracy: 0.5000
Epoch 27/500
1/1 [==============================] - 0s 11ms/step - loss: 0.7017 - accuracy: 0.5000
Epoch 28/500
1/1 [==============================] - 0s 10ms/step - loss: 0.7014 - accuracy: 0.5000
Epoch 29/500
```

Evaluate the model on test

```
model.evaluate(X_test_scaled,y_test)
```

```
1/1 [==============================] - 0s 153ms/step - loss: 0.6210 - accuracy: 0.6667
[0.621041476726532, 0.6666666865348816]
```

```
model.predict(X_test_scaled)
```

```
1/1 [==============================] - 0s 83ms/step
array([[0.66075516],
       [0.6084721 ],
       [0.45149705],
       [0.6268543 ],
```



```
       [0.66423655],
       [0.68474233]], dtype=float32)
```

Now get the value of weights and bias from the model

```
coef, intercept = model.get_weights()
coef, intercept
```

```
(array([[0.7785071 ],
        [0.69779867]], dtype=float32),
 array([-0.39703575], dtype=float32))
```

This means w1=5.060867, w2=1.4086502, bias =-2.9137027

NOW MAKE THE FUNCTUION'S

```
def sigmoid(x):
        import math
        return 1 / (1 + math.exp(-x))
sigmoid(18)

X_test
```

	age	affordibility
2	47	1
10	18	1
21	26	0
11	28	1
14	49	1
9	61	1

Instead of model.predict, write our own prediction function that uses w1,w2 and bias


```
def prediction_function(age, affordibility):
    weighted_sum = coef[0]*age + coef[1]*affordibility + intercept
    return sigmoid(weighted_sum)

prediction_function(.47, 1)
```

```
0.6607551573836065
```

```
prediction_function(.18, 1)
```

```
0.6084720839511557
```

**Now we start implementing gradient descent in plain python. Again the goal is to come up with same w1, w2 and bias that keras model calculated. We want to show how keras/tensorflow would have computed these values internally using gradient descent

First write couple of helper routines such as sigmoid and log_loss**

```
def sigmoid_numpy(X):
   return 1/(1+np.exp(-X))

sigmoid_numpy(np.array([12,0,1]))
```

```
array([0.99999386, 0.5       , 0.73105858])
```

```
def log_loss(y_true, y_predicted):
    epsilon = 1e-15
    y_predicted_new = [max(i,epsilon) for i in y_predicted]
    y_predicted_new = [min(i,1-epsilon) for i in y_predicted_new]
    y_predicted_new = np.array(y_predicted_new)
    return -np.mean(y_true*np.log(y_predicted_new)+(1-y_true)*np.log(1
```

All right now comes the time to implement our own custom neural network class !! yay !!!*


```
class myNN:
    def __init__(self):
        self.w1 = 1
        self.w2 = 1
        self.bias = 0

    def fit(self, X, y, epochs, loss_thresold):
        self.w1, self.w2, self.bias = self.gradient_descent(X['age'],
        print(f"Final weights and bias: w1: {self.w1}, w2: {self.w2},

    def predict(self, X_test):
        weighted_sum = self.w1*X_test['age'] + self.w2*X_test['afford
        return sigmoid_numpy(weighted_sum)

    def gradient_descent(self, age,affordability, y_true, epochs, los
        w1 = w2 = 1
        bias = 0
        rate = 0.5
        n = len(age)
        for i in range(epochs):
            weighted_sum = w1 * age + w2 * affordability + bias
            y_predicted = sigmoid_numpy(weighted_sum)
```

```
            loss = log_loss(y_true, y_predicted)

            w1d = (1/n)*np.dot(np.transpose(age),(y_predicted-y_true))
            w2d = (1/n)*np.dot(np.transpose(affordability),(y_predict

            bias_d = np.mean(y_predicted-y_true)
            w1 = w1 - rate * w1d
            w2 = w2 - rate * w2d
            bias = bias - rate * bias_d

            if i%50==0:
                print (f'Epoch:{i}, w1:{w1}, w2:{w2}, bias:{bias}, lo

            if loss<=loss_thresold:
                print (f'Epoch:{i}, w1:{w1}, w2:{w2}, bias:{bias}, lo
                break

```

Now make a Object of the class myNN


```
customModel = myNN()
customModel.fit(X_train_scaled, y_train, epochs=8000, loss_thresold=0.
```

```
Epoch:0, w1:0.974907633470177, w2:0.948348125394529, bias:-0.11341867736368583, loss:0.7113403233723
Epoch:50, w1:1.503319554173139, w2:1.108384790367645, bias:-1.2319047301235464, loss:0.5675865113475
Epoch:100, w1:2.200713131760032, w2:1.2941584023238903, bias:-1.6607009122062801, loss:0.53906804177
Epoch:150, w1:2.8495727769689085, w2:1.3696895491572745, bias:-1.986105845859897, loss:0.51764621642
Epoch:200, w1:3.443016970881803, w2:1.4042218624465033, bias:-2.2571369883752723, loss:0.50050112696
Epoch:250, w1:3.982450494649576, w2:1.4239127329321233, bias:-2.494377365971801, loss:0.486540895376
Epoch:300, w1:4.472179522095915, w2:1.438787986553552, bias:-2.707387811922373, loss:0.4750814640632
Epoch:350, w1:4.917245868007634, w2:1.4525660781176122, bias:-2.901176333556766, loss:0.465614753069
Epoch:366, w1:5.051047623653049, w2:1.4569794548473887, bias:-2.9596534546250037, loss:0.46293944095
Final weights and bias: w1: 5.051047623653049, w2: 1.4569794548473887, bias: -2.9596534546250037
```

```
coef, intercept
```

```
(array([[0.7785071 ],
        [0.69779867]], dtype=float32),
 array([-0.39703575], dtype=float32))
```

This shows that in the end we were able to come up with same value of w1,w2 and bias using a plain python implementation of gradient descent function

```
X_test_scaled
```

	age	affordibility
2	0.47	1
10	0.18	1
21	0.26	0
11	0.28	1
14	0.49	1
9	0.61	1

(1) Predict using custom model

```
1 customModel.predict(X_test_scaled)
```

```
2     0.705020
10    0.355836
21    0.161599
11    0.477919
14    0.725586
9     0.828987
dtype: float64
```

(2) Predict using tensorflow model


```
1 model.predict(X_test_scaled)
```

```
1/1 [==============================] - 0s 35ms/step
array([[0.66075516],
       [0.6084721 ],
       [0.45149705],
       [0.6268543 ],
       [0.66423655],
       [0.68474233]], dtype=float32)
```

Above you can compare predictions from our own custom model and tensoflow model. You will notice that predictions are almost same