# <u>CE – 235: Artificial Intelligence and Data Science</u>

Course Project: Rainfall Prediction in Australia

Team members


| Name | Roll Number |
| --- | --- |
| **Aryan Amit Bagdia** | 22b2730 |
| **Nikhil Agrawal** | 22b4212 |
| **Naveen banoth** | 22b0707 |
| **Pramod Sai** | 22b0709 |
| **Bikram** | 22b0730 |
| **Shubham kumar meena** | 22b0712 |


# Introduction:

We are going to build a logistic regression model from scratch using Numpy,pandas,matplotlib and the cost function and the sigmoidal function , going to test it on data set and get the result .To build a logistic regression model from scratch, we need to define the cost function and the sigmoidal function. The cost function measures how well the model fits the training data. The sigmoidal function is the activation function that we will use to predict the probability of the target class.The cost function for logistic regression is called the binary cross-entropy loss function.

### Data Collection:

We took the dataset from kaggel . The dataset available from Kaggle serves as the primary data source for this project. (here is the link) This comprehensive dataset contains daily weather observations from numerous locations across Australia over a period of nearly 10 years. The dataset encompasses various meteorological variables, including temperature, humidity, rainfall, and wind speed, providing ample information for developing a robust rainfall prediction model.The target variable of interest in this study is RainTomorrow, which indicates whether or not it will rain at a given location on the following day. This binary outcome serves as the basis for classifying days into either rainy or non-rainy categories.

# Methodology:

We started with building a logistic Regression model.

```
07]
class LogisticRegression:
    def __init__(self, learning_rate, no_of_iterations):

        self.learning_rate = learning_rate

        self.no_of_iterations = no_of_iterations



    def fit(self, x, y):
        self.m, self.n = x.shape

        self.w = np.zeros(self.n)

        self.b = 0

        self.x = x

        self.y = y

        for i in range(self.no_of_iterations):

            self.update_weights()

    def update_weights(self):

        y_predicted = 1 / (1 + np.exp(-(self.x.dot(self.w)+ self.b)))

        dw = (1 / self.m) * np.dot(self.x.T, (y_predicted - self.y))

        db = (1 / self.m) * np.sum(y_predicted - self.y)

        self.w = self.w - self.learning_rate * dw

        self.b = self.b - self.learning_rate * db


    def predict(self, x):

        y_prediction = 1 / (1 + np.exp(-(x.dot(self.w) + self.b)))

        y_prediction = np.where(y_prediction > 0.5, 1, 0)
        return y_prediction
```

Here we created a class called Logistic regression with arguments learning_rate and no_of_iterations. And gave a constructor which initilizes it with hyperparameters. learning_rate and no_of_iterations are the hyperparameters given to the function by us . . And now I defined a function called fit() and in the function I defined some variables b,x,y,w. W is an array with all zeros and will update all the entries in those variables in further steps. To update weights we created a function called update_weights().In this function we used the results we derived.
Further, i defined a function called predict() which will take the input “X” and put it in the sigmod equation and gives probabilities to predict the output .

Cost function of Logistic Regression

Let there are $x_1, x_2 ---- x_n$ Predictor Variables

let $w_1, w_2 ---- w_n$ are weights of Predictor Variables & 'b' is a bias.

now

$$y = \sigma(w_1 x_1 + w_2 x_2 + ------ + w_n x_n + b)$$

now

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} \begin{bmatrix} x_1 & x_2 & ---- & x_n \end{bmatrix} + b$$

$$= w^T \cdot x + b$$

$$\Rightarrow \boxed{\hat{y} = \sigma(w^T \cdot x + b)} \Rightarrow \boxed{\hat{y} = \frac{1}{1+e^{-z}}}, \boxed{z = w^T x + b}$$

now

Cost Function of Logistic Regression $= -\frac{1}{m} \sum_{i=1}^{m} \left[ y_i \log \hat{y}_i + (1-y_i) \log (1-\hat{y}_i) \right]$

now

By using Gradient Descent we will take some random Parameters for w, b Initially & to optimise the cost Function, so that to minimize the cost Function.

Let $w_1, b_1$ be Initial Parameters $(w_1, b_1)$ $(w', b')$

To optimse & Move further, we use a Parameter called learning rate

Parameter called learning rate

→ To proceed further $\hat{w}, \hat{b}$ are choosen in such a way that it reduces lost function.

i.e

$$\boxed{w^1 = w_1 - (\text{Learning rate}) \times \frac{d[\text{lost Function}]}{\partial w}}$$

$$\boxed{b' = b_1 - (\text{Learning rate}) \times \frac{d[\text{lost Function}]}{\partial b}}$$

now for a general point $(w, x)$

Let 'L' be the Loss Function, $\boxed{z = w^T x + b}$

$$\Rightarrow \frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}}\left(\frac{d\hat{y}}{dw}\right) = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w}$$

$$\boxed{\frac{\partial L}{\partial \hat{y}} = \frac{-y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}}$$

now

$$\frac{\partial \hat{y}}{\partial z} = \left(1+e^{-z}\right)^{-1'} = \frac{-1}{(1+e^{-z})^2} \cdot e^{-z} = \hat{y}(1-\hat{y})$$

now w.r.t

$$\hat{y} = \frac{1}{1+e^{-z}} \Rightarrow \boxed{e^{-z} = \frac{1-\hat{y}}{\hat{y}}}$$

now

$$\frac{\partial L}{\partial w} = \frac{-y}{\hat{y}} + \left(\frac{1-y}{1-\hat{y}}\right) \times \hat{y}(1-\hat{y}) \times x$$

$$= \frac{\hat{y}-y}{\cancel{\hat{y}(1-\hat{y})}} \cdot \cancel{\hat{y}(1-\hat{y})} \times x$$

$$\boxed{\frac{\partial L}{\partial w} = (\hat{y}-y)x}$$

now

$$\frac{\partial L}{\partial w_1} = (\hat{y}_1 - y_1)x_1$$

$$\frac{\partial L}{\partial w_2} = (\hat{y}_2 - y_2)x_2$$

$$\vdots \qquad \vdots$$

$$\frac{\partial L}{\partial w_n} = (\hat{y}_n - y_n)x_n$$

Similarly

$$\left(\frac{\text{Cost Function}}{\partial w}\right) = \frac{\sum \text{Loss Function}}{m}$$

$$= \frac{\hat{y}-y}{m}$$

now

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial b}$$

$$= \frac{(\hat{y}-y) \times \hat{y}(1-\hat{y}) \times 1}{\hat{y}(1-\hat{y})}$$

$$= \hat{y} - y$$

now

$$\boxed{\frac{\partial[\text{Cost Function}]}{\partial b} = \frac{\hat{y}-y}{m}}$$

now we are going to use them in the code to Build Logistic Regression

```
data = pd.read_csv("/content/weatherAUS.csv")
data
```

| | Date | Location | MinTemp | MaxTemp | Rainfall | Evaporation | Sunshine | WindGustDir | WindGustSpeed | WindDir9am | ... | Humidity9am | Humidity3pm | Pressure9am | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2008-12-01 | Albury | 13.4 | 22.9 | 0.6 | NaN | NaN | W | 44.0 | W | ... | 71.0 | 22.0 | 1007.7 | |
| | 2008-12-02 | Albury | 7.4 | 25.1 | 0.0 | NaN | NaN | WNW | 44.0 | NNW | ... | 44.0 | 25.0 | 1010.6 | |
| | 2008-12-03 | Albury | 12.9 | 25.7 | 0.0 | NaN | NaN | WSW | 46.0 | W | ... | 38.0 | 30.0 | 1007.6 | |
| | 2008-12-04 | Albury | 9.2 | 28.0 | 0.0 | NaN | NaN | NE | 24.0 | SE | ... | 45.0 | 16.0 | 1017.6 | |
| | 2008-12-05 | Albury | 17.5 | 32.3 | 1.0 | NaN | NaN | W | 41.0 | ENE | ... | 82.0 | 33.0 | 1010.8 | |

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Date           145460 non-null  object
 1   Location       145460 non-null  object
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object
 10  WindDir3pm     141232 non-null  object
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object
 22  RainTomorrow   142193 non-null  object
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
```

```
data.isnull().sum()
```

```
Date                 0
Location             0
MinTemp           1485
MaxTemp           1261
Rainfall          3261
Evaporation      62790
Sunshine         69835
WindGustDir      10326
WindGustSpeed    10263
WindDir9am       10566
WindDir3pm        4228
WindSpeed9am      1767
WindSpeed3pm      3062
Humidity9am       2654
Humidity3pm       4507
Pressure9am      15065
Pressure3pm      15028
Cloud9am         55888
Cloud3pm         59358
Temp9am           1767
Temp3pm           3609
RainToday         3261
RainTomorrow      3267
dtype: int64
```

Importing the dataset from the csv file downloaded from Kaggle. The dataset is of 145460 rows × 23 columns. Extracting the information from the downloaded file to analyse the data.Checking the number of empty values in each column by data.isnull().sum() and reporting the number of empty cells. Out of 85000 total entries, almost 65000 are empty in 'Evaporation', and out of 75,000 total entries, almost 70,000 are empty in 'Sunshine'. Since most of the data for this particular category is empty, it is best to avoid these columns. Hence, we are dropping these columns from the dataset.

```
2   data = data.drop(["Evaporation","Sunshine"],axis=1)
```

```
numerical_feature = [feature for feature in data.columns if data[feature].dtypes != "O"]
discrete_feature=[feature for feature in numerical_feature if len(data[feature].unique())<25]
continuous_feature = [feature for feature in numerical_feature if feature not in discrete_feature]
categorical_feature = [feature for feature in data.columns if feature not in numerical_feature]
print("Numerical Features Count {}".format(len(numerical_feature)))
print("Discrete feature Count {}".format(len(discrete_feature)))
print("Continuous feature Count {}".format(len(continuous_feature)))
print("Categorical feature count {}".format(len(categorical_feature)))
```

```
Numerical Features Count 14
Discrete feature Count 2
Continuous feature Count 12
Categorical feature count 7
```

We are segregating the data into categorical (classification) and numerical (numbers) features to analyze data properly and we further classified numerical data into discrete and continuous features. We segregated numerical values based upon their datatypes, i.e., if the datatype is object, then we considered it as categorical, if the datatype is not object, we considered it as numerical. If there are very less number of different entries in particular column then we consider it as discrete and if there are more we consider it as continuous so that it will be easy to plot a distribution while doing data analysis.

```

def randomsamplelimputation(data,variable):
    data[variable]=data[variable]
    random_sample=data[variable].dropna().sample(data[variable].isnull().sum(),random_state=0)
    random_sample.index=data[data[variable].isnull()].index
    data.loc[data[variable].isnull(),variable]=random_sample
```

```
randomsamplelimputation(data,"Cloud3pm")
randomsamplelimputation(data,'WindDir9am')
randomsamplelimputation(data,'RainToday')
randomsamplelimputation(data,'MinTemp')
randomsamplelimputation(data,'MaxTemp')
randomsamplelimputation(data,'Rainfall')
randomsamplelimputation(data,'WindGustDir')
randomsamplelimputation(data,'WindDir3pm')
randomsamplelimputation(data,'WindSpeed9am')
randomsamplelimputation(data,'WindSpeed3pm')
randomsamplelimputation(data,'WindGustSpeed')
randomsamplelimputation(data,'Humidity9am')
randomsamplelimputation(data,'Humidity3pm')
randomsamplelimputation(data,'Pressure9am')
randomsamplelimputation(data,'Pressure3pm')
randomsamplelimputation(data,'Cloud9am')
randomsamplelimputation(data,'Temp9am')
randomsamplelimputation(data,'Temp3pm')
```

We are creating a function to impute (add) the data in the missing places of our dataset. This function adds the data to the missing values of the dataset. Imputing the data in all columns except ‘RainTomorrow’ since we are predicting ‘RainTomorrow’.

.

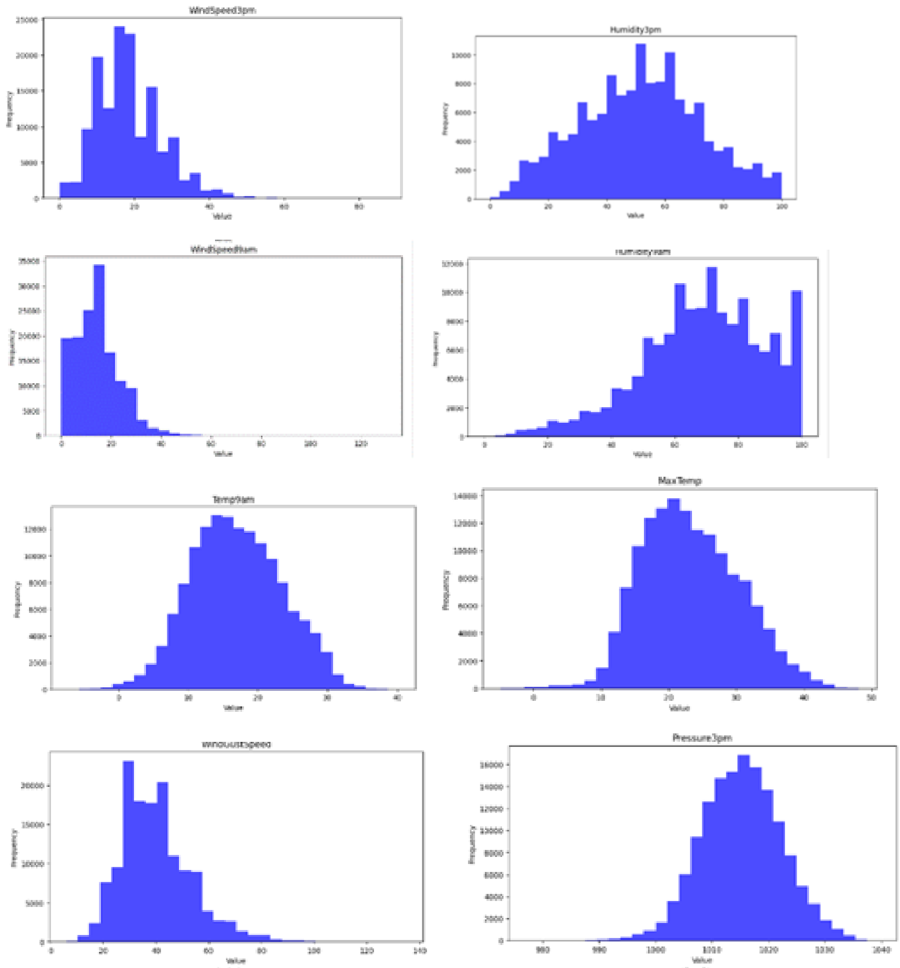
```
1  data.isnull().sum()
```

```
Date             0
Location         0
MinTemp          0
MaxTemp          0
Rainfall         0
WindGustDir      0
WindGustSpeed    0
WindDir9am       0
WindDir3pm       0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
RainToday        0
RainTomorrow  3267
dtype: int64
```

Now after imputing, the dataset is full without any missing values. Now we have to check whether the imputed data is proper or not, i.e., if it aligns with the original data. Here, only the column 'RainTomorrow' has missing entries, since we are predicting it, we are not going to add any entries in that column. ('RainTomorrow' is the predicted variable). So, we are dropping all the rows which have empty entries for 'RainTomorrow'.

```
1  data.dropna(inplace=True)
2  data.isnull().sum()
```

Now, there are no missing values in the dataset, and it's ready to use, but before using that, we have to check the validity of the data because we added the missing values. For that, we are plotting the distributions



Now from the distributions, we can see that the distribution of the whole data forms a normal distribution as shown in the above graphs hence, we can say that the data we imputed is proper and aligns with the data. The fact that the imputed data aligns with the overall data distribution is a positive indication of the appropriateness of the imputation methods that we have used. The goal is to fill in the missing values in a way that the imputed data maintains the statistical properties of the original data and is fulfilled.

**Plotting correlation matrix:**

```
[136] 1   corr=data.corr()

<ipython-input-136-c0bda979f113>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated.
  corr=data.corr()
```

```
[137] 1   corr.style.background_gradient(cmap = 'coolwarm')
```

| | MinTemp | MaxTemp | Rainfall | WindGustSpeed | WindSpeed9am | WindSpeed3pm | Humidity9am | Humidity3pm | Pressure9am |
|---|---|---|---|---|---|---|---|---|---|
| ıp | 1.000000 | 0.731741 | 0.102630 | 0.168042 | 0.173798 | 0.172547 | -0.230463 | 0.006302 | -0.402136 |
| ıp | 0.731741 | 1.000000 | -0.073582 | 0.065019 | 0.014477 | 0.049975 | -0.496312 | -0.492909 | -0.293644 |
| ll | 0.102630 | -0.073582 | 1.000000 | 0.122192 | 0.085563 | 0.056583 | 0.218112 | 0.244761 | -0.151457 |
| ;peed | 0.168042 | 0.065019 | 0.122192 | 1.000000 | 0.555937 | 0.630503 | -0.201724 | -0.024534 | -0.393017 |
| d9am | 0.173798 | 0.014477 | 0.085563 | 0.555937 | 1.000000 | 0.506377 | -0.265954 | -0.030572 | -0.203866 |
| d3pm | 0.172547 | 0.049975 | 0.056583 | 0.630503 | 0.506377 | 1.000000 | -0.141832 | 0.015479 | -0.262959 |

From the above correlation matrix, we can infer that 'MaxTemp' and 'Temp3pm' have a strong positive correlation (0.959), which means that as 'MaxTemp' increases, 'Temp3pm' also tends to increase. 'Humidity9am' and 'Humidity3pm' have a strong negative correlation (-0.543601), suggesting that as 'Humidity9am' increases, 'Humidity3pm' tends to decrease. 'Rainfall' and 'WindSpeed9am' have a low correlation (0.085563), indicating a weak relationship between these two variables. From the above matrix, we can interpret that all the features contribute towards predicting the 'RainTomorrow' i.e, considering every feature is important.

Since our data has some categorical features, we have to change them in such a way that our system understands it, i.e., we have to change every entry in the categorical feature into dummy items such that our model can be trained with those features.

```
dummies = pd.get_dummies(data[['Date', 'Location', 'WindDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow']])

data[['Date', 'Location', 'WindDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow']]
```

We created dummies values for further items in categorical features with the help of
pd.get_dummies(). Now we added dummy values to the dataset and now we have to remove the previous categorical values and concatenate these dummy values to the original dataset.

```
# Concatenate the original dataset with the dummy variables
data = pd.concat([data, dummies], axis=1)
# Droping the original categorical columns
data = data.drop(['Date', 'Location', 'WindDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow'], axis=1)
```

Since the dataset is very big(it has 142193 rows and 3521 columns), so it is difficult to train(it will take lot of time, might be hours or days) the model. That is why we are considering some parts of the data i.e considering and taking the data in batches.

Creating a data batch of 2000 rows, training and testing the model which we have created.

```
new_data = data.head( 2000)
```

Creating predictor and predicated variables. To train and test the model.

```
X = new_data.drop(["RainTomorrow_No","RainTomorrow_Yes"],axis= 1)
Y = new_data["RainTomorrow_Yes"]


len(X)
```

```
2000
```

Now splitting the data into training and testing set.

```



# Splittinh the data into training and testing sets

X_train = X[:1600]
Y_train = Y[:1600]
X_test= X[1600:2000]
Y_test = Y[1600:2000]


X[1:2]
```

Afterwards we created 2 models **to take validation of our created "LogisticRegression()".**

# Results:

It took 10 minutes for the model-1 to get trained by the created batch. Our model secured an accuracy score of 0.2075

```
145] predictions=model.predict(X_test)

] predictions

array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1])

147] accuracy = np.sum(predictions == Y_test) / len(Y_test)
     accuracy

0.2075
```

```
53] predictions=model_2.predict(X_test)

/usr/local/lib/python3.10/dist-packages/pandas/core/arraylike.py:402: RuntimeWarning: overflow er
  result = getattr(ufunc, method)(*inputs, **kwargs)

predictions

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0])

55] accuracy = np.sum(predictions == Y_test) / len(Y_test)
    accuracy

0.7925
```

It took 20 minutes for the model-2 to get trained by the created batch. Our model secured an accuracy score of 0.7925

# Conclusion :

Gradient descent is an optimization algorithm used to find the minimum of a function. It works by iteratively taking steps in the opposite direction of the gradient of the function. The gradient of a function represents the direction of the greatest increase of the function. By taking steps in the opposite direction of the gradient, gradient descent can gradually move towards the minimum of the function.

Gradient descent algorithm:
1. Start with an initial guess for the parameters of the function.
2. Calculate the gradient of the function at the current parameters.
3. Take a step in the opposite direction of the gradient.
4. Repeat steps 2 and 3 until the function converges to a minimum.

This is one of the most efficient and simple way to get towards to result with better accuracy, this is the reason why we chose this method.

First, we built a LogisticsRegresson class and then we pre-processed the data and imputed the missing values with some numpy method. To check whether the imputed values make sense we plotted the distribution curves of those predictor variables and we saw that we were getting proper normal distribution cures by this we ensured that the data imputed aligned with the original data and we could rely on that. After this, we trained the model it took 10 mins initially to train the model.
We created 2 models , In model-1 we gave learning_rate=0.01,no_of_iterations =100 and in model_2 we gave learning_rate=0.001,no_of_iterations =10000 as hyperparameters . The accuracy score of model-1 is 0.2075 and the accuracy score of model_2 is 0.7925.
The learning_rate and no_of_iterations are hyperparameters that control how quickly gradient descent converges. A smaller learning rate will converge more slowly, but it is less likely to overshoot the minimum of the function. A larger learning rate will converge more quickly, but it is more likely to overshoot the minimum of the function. We can see that in the results too.