

1 AI models used

1.1 Multivariate linear regression

1.1.1 Why ?

This Model is used when working with multiple variables, each independent. It is also used when working with probability. This algorithm has proven its effects in many world applications, such as Economics: To predict GDP growth based on investment, labor force, and technology and Engineering: to predict material strength based on composition and environmental factors.

1.1.2 How does it work?

The Model takes all variables given in the dataset labelled x_1, x_2, \dots, x_n and multiplies each variable with a confession $\theta_1, \theta_2, \dots, \theta_n$ and finally adds $x_0 = 1$ and θ_0 which is the initial offset. We have the formula for our hypothesis $h(\theta)$:

$$h_{\Theta}(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

changing to a vector notation will be helpful later

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \quad \Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix}$$

Finally after all the work

$$h_{\Theta}(x) = [\theta_0 \quad \theta_1 \quad \dots \quad \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} = \theta^{\top} x$$

This for one line of data if we consider all the data set our equations changes since X is a matrix the result of all of that is a Vector with all the Values of the hypothesis for all the dataset. The new formula is

$$H = \begin{bmatrix} x_0^1 & x_1^1 & \dots & x_n^1 \\ x_0^2 & x_1^2 & \dots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \dots & x_n^m \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix} = X \cdot \Theta$$

This hypothesis is not the result we are looking for we add the term ϵ to indicate the difference our final equation is

$$y = h_{\Theta}(x) + \epsilon$$

To try to minimize ϵ we calculate the cost function which determines if the values of the parameters are close to optimal or not this a formula

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^m \left(h_{\Theta}(x^{(i)}) - y^{(i)} \right)^2 = \frac{1}{2m} \sum_{i=1}^m (H - Y)^2 = \frac{1}{2m} \sum_{i=1}^m \begin{bmatrix} H_0 - y_0 \\ H_1 - y_1 \\ \dots \\ H_n - y_n \end{bmatrix}$$

where

m: is the length of the dataset

Y: a vector containing all the output for each line of the dataset

This value of $J(\Theta)$ can be used to update the parameter vector Θ by using the gradient descent which consists of subtracting from each element of Θ a learning rate named α (for most cases it should be that big so that the gradient will not shut the minimal point, $J(\Theta)$ and the partial derivative of the element and we repeat until it converges(if $J(\Theta) \leq 10^{-3}$) :

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\Theta)$$

(simultaneously update for every $j = 0, \dots, n$)

expanding and doing the derivation of the formula we have this:

$$\theta_j := \theta_j - \alpha \frac{2}{m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

expanding more :

$$\begin{aligned} \theta_j &:= \theta_j - \alpha \frac{2}{m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \\ \theta_0 &:= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right) x_0^{(i)} \\ \theta_1 &:= \theta_1 - \alpha \frac{2}{m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right) x_1^{(i)} \\ &\vdots \\ \theta_n &:= \theta_n - \alpha \frac{2}{m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right) x_n^{(i)} \end{aligned}$$

changing to vector notation and applying this formula to all data of the dataset and shortening all by having the vector

$$E = \sum_{i=1}^m (H - Y)$$

we have our last formula for gradient descent :

$$\Theta := \Theta - \alpha \cdot \frac{2}{m} \cdot X^T \cdot E$$

1.1.3 Optimization and Problems to check :

- Scaling (Optimization) : When working with Gradient Descent it is best to feature scaling the data that we are working with so that each element of the dataset will be removed from the mean of the column and then divided by the difference between the max and min values in the column and the formula is:

$$x_j^{(i)} := \frac{x_j^{(i)} - \mu_i}{s_i}$$

where

μ_i : The mean of the column

s_i : is the difference between the max and min values in the column after that

$$-0.5 \leq x_j^{(i)} \leq 0.5$$

implementation in Python will be something like this

```
1 import pandas as pd
2 import numpy as np
3 # Data Scaling (Standardization: mean=0, std=1)
4 k = 21 # the number of columns of X
5 mean = np.mean(X[:, 1:k], axis=0) # Compute mean for
   features (exclude Column_of_Ones)
6 std = np.std(X[:, 1:k], axis=0) # Compute std for
   features (exclude Column_of_Ones)
7 X[:, 1:k] = (X[:, 1:k] - mean) / std # Scale features
   (Column_of_Ones remains unchanged)
```

- seeing a correlation (Problems to check):
This is an essential step when working with multiple variables for that we check for any two pairs of variables vector are linear correlated using this formula:

$$r = \frac{\text{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y}$$

where :

$\text{Cov}(X, Y)$: Covariance between X and Y.

σ_X, σ_Y : Standard deviations of X and Y

This poses is standard for each 2 pairs if $|r| \leq 0.8$ the two variables are not linear related. for the implementation, we visualize all possible pairs in a grid and the value of r in the heat map:

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 import numpy as np
5 df = pd.DataFrame(data)
6 # Calculate the correlation matrix
7 correlation_matrix = df.corr().round(3)
```

```

8 # Heatmap of correlation matrix
9 sns.heatmap(correlation_matrix, annot=True,
10             cmap="coolwarm")
11 plt.title("Correlation Matrix Heatmap")
12 plt.show()

```

- Visualizing data (optional):

We want to know the frequency of each variable's Values. For this, we will use bars and see the distribution. Using Python already built library :

```

1 import matplotlib.pyplot as plt
2 # Histogram with more bins (bars)
3 plt.figure(figsize=(8, 6))
4 plt.hist(np.max(data['X']))
5 plt.hist(data['X'], bins=30, color='blue',
6          edgecolor='black', alpha=0.6, label='X Values') #
7 # More bins
8 plt.xlabel('X')
9 plt.ylabel('Frequency')
10 plt.title('Frequency Distribution of X')
11 plt.legend()
12 plt.show()

```

- overfitting (Problems to check): Overfitting occurs when a machine learning model learns not only the underlying patterns in the training data but also the noise and random fluctuations. This makes the model overly complex and tailored to the training dataset, which harms its ability to generalize to unseen data. This could be harmful for our purposes for we use $J(\Theta)$ discussed 2.1.2 if $J(\Theta)$ is a fresh input that hasn't been using for training so we now the result is acceptable:

The implementation will be :

```

1 import numpy as np
2 k = 1000 # The number of data row
3 X = data.drop(columns=["Y"]).values
4 X = X[:-k]
5 y = data["Y"].values
6 y = y[:-k]
7 m = X.shape[0]
8 predictions = np.dot(X, theta)
9 J = (1 / m) * np.sum(errors ** 2) # Mean
10 # Absolute Error
11 print(f"Mean Squared Error: {round(J, 6)}")

```

1.1.4 implementation:

we will be using Python because it's the most reliable one and our implementation will be:

```

1 import pandas as pd
2 import numpy as np
3 import math
4 # Load the dataset
5 file_name = "data.csv"
6 data = pd.read_csv(file_name)
7 data.insert(0, 'Column_of_Ones', 1) # Add a column of ones
   for the intercept term
8
9 # Separate the features (X) and the target (y)
10 X = data.drop(columns=["FinalProbability"]).values
11 y = data["FinalProbability"].values
12
13 # Data Scaling (Standardization: mean=0, std=1)
14 k = 21 # the number of columns of X
15 mean = np.mean(X[:, 1:k], axis=0) # Compute mean for
   features (exclude Column_of_Ones)
16 std = np.std(X[:, 1:k], axis=0) # Compute std for
   features (exclude Column_of_Ones)
17 X[:, 1:k] = (X[:, 1:k] - mean) / std # Scale features
   (Column_of_Ones remains unchanged)
18
19 # Initialize parameters
20 theta = np.ones(X.shape[1]) # Initialize theta
21 alpha = 0.1 # Learning rate()
22 m = X.shape[0] # Number of training examples
23
24 # Gradient Descent with MSE
25 J = float('inf')
26 iteration = 0
27
28 while J > 1e-3:
29     predictions = np.dot(X, theta)
30     errors = predictions - y
31     J = (1 / m) * np.sum(errors ** 2)
32     # Update theta
33     theta -= alpha * (2/m) * np.dot(X.T, errors )
34     iteration += 1

```

2 Catastrophe tackled:

2.1 Flood

2.1.1 Introduction to the Problem

2.1.2 Solution

This paragraph will discuss the 3 solutions we have using AI models from satellite:

- the likelihood of a flood :
Given different parameters from the satellite the AI can predict what place is vulnerable to flood and the possibility that it will happen
- The presence of flood :
giving the rainfall in mm from the satellite The AI can determine if we have a flood or not
- Another layer of protection The presence of flood:
Given the images of the AI and the AI determines if we have a flood or not based on the image.

2.1.3 the likelihood of a flood :

- dataset :
<https://www.kaggle.com/datasets/naiyakhalid/flood-prediction-dataset>
- The Model of AI used:
Multivariate linear regression
- Independent variables:
Using the code from 2.1.3 (seeing a correlation) we check all the variables' correlation and we have this picture :

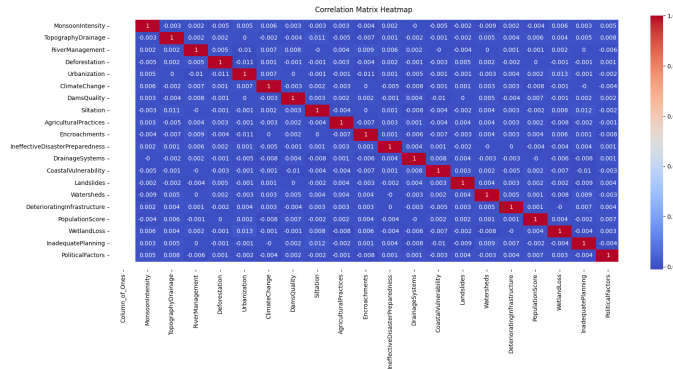
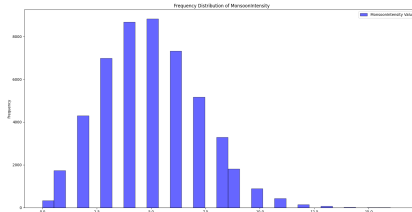


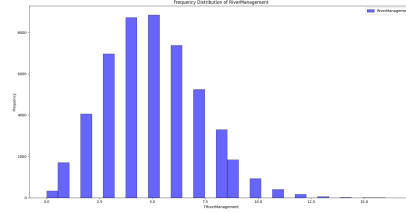
Figure 1: The Correlation Matrix

The correlation matrix showcases that the values are close to 0 and the diagonals are 1 (X is linearly related to itself)

- Visualizing data: Using the code from 2.1.3 (Visualizing Data) we check all the variables including the final probability and we have those pictures:



(a) MonsoonIntensity distribution



(b) RiverManagement distribution

Figure 2: distribution of variables to their frequency of appearing

All the other variables have the same trend the only difference is the probability distribution where the probabilities are mostly concentrated in the center

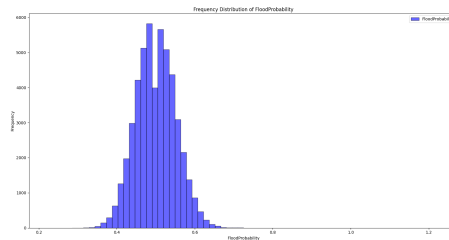


Figure 3: FloodProbability distribution

- Choosing the learning rate:
The learning rate α is important for the gradient descent to have α too big and the gradient will converge. have α too small and will take forever to converge for this we will be choosing different values of alpha and then will be timing all of them for the same data size ($m = 300$) and we will showcase the plot of $J(\Theta)$ in relation of the number of iteration:
We will be choosing $\alpha = 0.3$ since it gives the faster convergence without over shooting

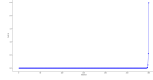
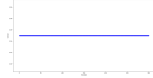

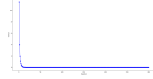
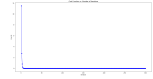
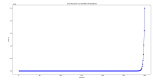
Alpha	Time	Graph
-0.5	0.09281	
0	0.06708	
0.1	0.06727	
0.3	0.07359	
0.7	0.08604	
1	0.08894	

Table 1: The Values Table

- testing accuracy : After training the module with 80 % of all the data we used the other 20 % of the rest to test the accuracy of the algorithm and to test if we have a rencounter overfitting problem 2.1.3 (overfitting). The testing algorithm has an error of 0.000117, which is below the value to be concerned about.