Assignment-1: Multivariate Regression

Rezwan-Ul-Alam (ID: 2011659042)
Md. Nur Alam Jowel (ID: 2012355042)
Raian Ruku (ID: 2013409642)

Email addresses:
rezwan.alam1@northsouth.edu
alam.jowel@northsouth.edu
raian.ruku@northsouth.edu

I. Introduction

This assignment aims to develop a multi-variable linear regression model for predicting relative humidity (RH) based on the air quality factors obtained from a gas multi-sensor device deployed in an Italian city. In this context, our analysis will involve creating custom functions for computing the cost and gradient and implementing the gradient descent algorithm to optimize the model parameters. Finally, feature scaling will be applied to maximize the test performance.

TABLE I
Multiple Features

feature1	**feature2**	**feature3**	**feature4**	**target**
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

· Notation

$$n = \text{number of features}$$
$$m = \text{number of training examples}$$

Univariate Linear Regression: In univariate linear regression, only one target column (dependent variable) and one feature (independent variable). The hypothesis function is represented as:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Hence, both θ_0 and θ_1 are scalar values

Multivariate Linear Regression: In contrast, multivariate linear regression involves predicting a target variable based on multiple features. If there are n features, the hypothesis function is extended to:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n$$

Here, θ_0 is still a scalar representing the intercept, but $\theta_1, \theta_2, \ldots, \theta_n$ form a vector. This vector ($\boldsymbol{\theta}$) contains the coefficients associated with each feature.

To express it more formally:

$$h_\theta(x) = \boldsymbol{\theta}^T \mathbf{x}$$

Where:

- $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \ldots, \theta_n]$ is the parameter vector.
- $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$ is the feature vector, with $x_0 = 1$ to account for the intercept term.

In summary, while univariate linear regression deals with a single feature and scalar coefficients, multivariate linear regression extends this to multiple features, with the coefficients organized into a vector.

II. Method

A. Data Exploration and Preprocessing

- We have decided to work on **Absolute Humidity** (AH), so we will drop Temperature (T) and Relative Humidity (RH)
- **Data Exploration and Preprocessing:** Before diving into model development, exploring and preprocessing the dataset is crucial.
  - **Handling Missing Values:** In our dataset, missing values are identified as -200. So we searched on all columns for -200 and replaced the value with the mean of the column

```
import pandas as pd
import numpy as np
df = pd.read_excel('/content/gdrive/MyDrive/data/air.xlsx')
df.head()
for column in df.columns:
    total = df[column].isnull().sum()
    print(f"Column␣'{column}'␣has␣{total}␣null␣values")
    print("\n")
check = -200

#checking the presence of -200 in each of the columns
count = {}
for col in df.columns:
    count[col] = (df[col] == check).sum()

# Display the presence of -200 in each column
for column, count in count.items():
    print(f"Column␣'{column}'␣has␣{check},␣{count}␣times")

#replace the missing value with NaN and later replace with the mean
df.replace(-200, np.nan, inplace=True)
df.isnull().sum()

for col in df.columns:
    mean = df[col].astype(float).mean()
    df[col].replace(np.nan, mean, inplace=True)
```

  - **Dropping Unnecessary Columns:** The date and time columns do not contribute significantly to the analysis, and our target value is not temperature (T) or relative humidity (RH). While checking for missing values, the column NMHC(GT) was observed to have over 90% missing values. Therefore, we will drop these columns.

```
df = data.drop(['Date', 'Time','T','RH'], axis=1)

for column, count in count.items():
    print(f"Column␣'{column}'␣has␣{check},␣{count}␣times") #total data: 9357
#Output : Column 'NMHC(GT)' has -200, 8443 times

df = df.drop('NMHC(GT)', axis=1)
```

– **Scaling Features:** For feature scaling, we have used our own **min_max_scaler(column)** method. The code below introduces a min_max_scaling function to perform min-max scaling on numeric columns in our data frame. It calculates the scaled values, transforming X_features to a common scale between 0 and 1.

```
def min_max_scaling(column):
    minimum_value = column.min()
    maximum_value = column.max()
    scaled_column = (column - minimum_value) / (maximum_value - minimum_value)
    return scaled_column

#selecting featured columns from the dataframe
all_columns = df2.select_dtypes(include=[float, int]).columns
features_columns = all_columns[:-1]
print(features_columns)

#applying min_max_scaler() on all the features
df2[features_columns] = df2[features_columns].apply(min_max_scaling)
df2.head()
```

B. *Cost Function : compute_cost(X, y, w, b)*

- We have worked on the given solution of the univariate linear regression code. There are some changes in the code. For univariate regression, there was only one feature. As a result, θ_1 (in code w) was a scalar. However, in multivariate regression, θ_1 becomes an n-dimensional array, where n is the number of features. And X (ndarray(m, n)) becomes a matrix as multiple features are added. Therefore, we can perform **np.dot()** for calculating the value of $h_\theta(x)$. Rest of the code remains the same.

```
def compute_cost(X, y, w, b):

    m = X.shape[0]
    cost = 0.0

    for i in range(m):
        f_wb = np.dot(X[i], w) + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost

    return total_cost
```

C. *Gradient Descent : compute_gradient(X, y, w, b)*

- As with the cost function, the code has some changes. Unlike univariate, in multivariate regression,dj_dw represents the cost gradient for the parameters w. It is expected to be a 1-dimensional NumPy array of length n. Here,
  1) Initialize dj_dθ ($\theta > 0$) as a zero vector of shape $(n,)$ and dj_db (θ_0) as 0.
  2) Loop over each example in the training data (m times).
  3) Compute the predicted value (f_{wb}) for each example using the current weights (w) and bias (b).
  4) Calculate the common term, the difference between the predicted value and the actual target value (common $=$ $f_{wb} - y[i]$).
  5) Update the gradients (dj_dw and dj_db) using the common term and the input features.
  6) After looping through all examples, normalize the gradients by dividing them by the number of examples (m).
  7) Return the computed gradients.
  8) **Note:** dj_dw and dj_db are nothing but $\frac{\delta}{\delta\theta_j}J(\theta)$, where $J(\theta)$ is the cost function and θ_j represents the j-th parameter.

Repeat

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

(simultaneously update θ_j for $j = 0, 1, \ldots n$)

}

```
def compute_gradient(X, y, w, b):

    m = X.shape[0]
    n = X.shape[1]
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        f_wb = np.dot(X[i], w) + b
        common = f_wb - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + common * X[i, j]
        dj_db = dj_db + common
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw
```

D. *Gradient Function : gradient_descent()*

1) Initialize w and b with the provided initial values.
2) Loop over a specified number of iterations.
3) Compute the gradient of the cost function with respect to the parameters using the provided `gradient_function`.
4) Update the parameters (w and b) using the gradient and the learning rate.
5) Save the cost and parameter values at each iteration in the `J_history` and `p_history` lists, respectively.
6) Print the cost, gradients, and parameter values at regular intervals during training.
7) Return the final optimized parameters (w and b), the history of cost values, and the history of parameter values.

```
def gradient_descent(X,y,w_in,b_in,alpha,num_iters,
                          cost_function,gradient_function):
    J_history = []
    w = copy.deepcopy(w_in)
    p_history = []
    b = b_in

    for i in range(num_iters):
        dj_db, dj_dw = gradient_function(X, y, w , b)
        w = w - alpha * dj_dw
        b = b - alpha * dj_db
        if i<100000:
            J_history.append( cost_function(X, y, w , b))
            p_history.append([w,b])
        if i% math.ceil(num_iters/10) == 0:
          print(f"Iteration␣{i:4}:␣Cost␣{J_history[-1]:8.2f}␣",
                 f"dj_dw:␣{dj_dw},␣dj_db:␣{dj_db:␣0.3e}␣␣",
                 f"w:␣{w},␣b:{b:␣0.5e}")

    return w, b, J_history,p_history
```

E. ***Train Test Split***

- As per the instruction, the first 75% of the data was used for training, and the last 25% of the data was used for testing. Training values are the features, and testing values are the target values. The code is given below:

```
first_75 = math.ceil(0.75*len(df1))
data_train = df1[:first_75, :] #first 75% of the data
data_test = df1[first_75:, :] #last 25% of the data

x_train = data_train[:, :-1]
y_train = data_train[:, -1]
x_test = data_test[:, :-1]  # First 9 columns
y_test = data_test[:, -1]   # Last column

print(x_train.shape)
print(x_test.shape)
```

III. Experiment results

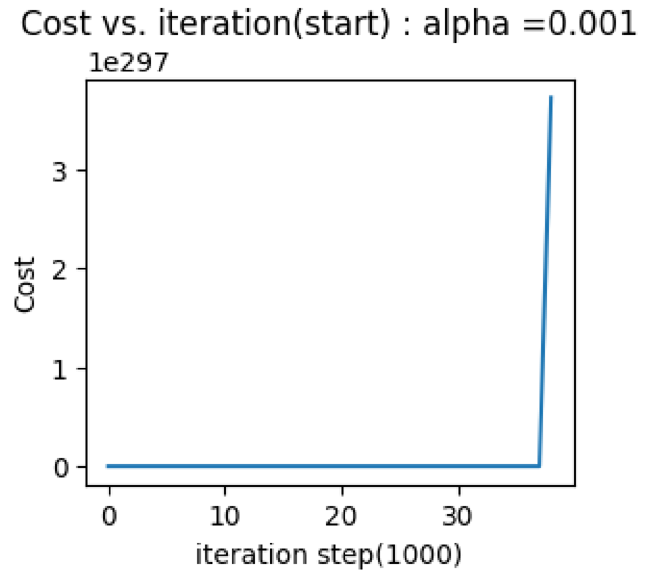
A. ***Before Scaling***

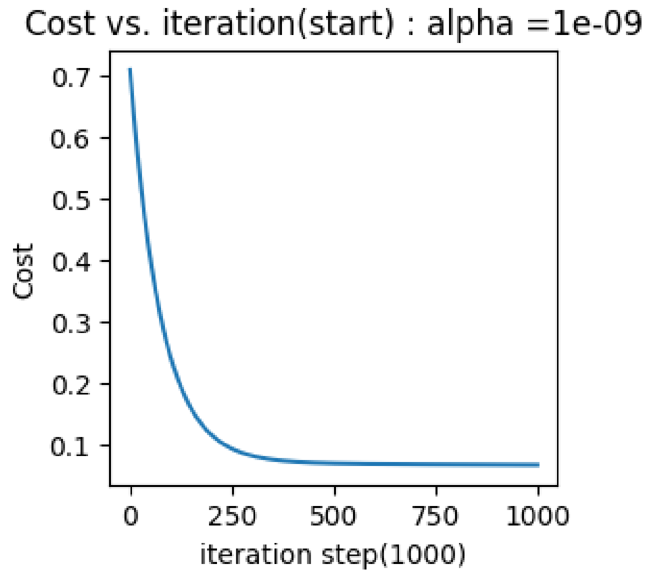
- Before scling we tried to check out output for different iteration value and learning rate.

Alpha	**Iteration**	**Cost**
1.0e-7	100	0.15
	200	0.05
	⋮	⋮
	800	0.04
	900	0.04
1.0e-8	100	0.64
	200	0.07
	⋮	⋮
	800	0.06
	900	0.06
1.0e-8 (3000 Iterations)	300	0.64
	600	0.07
	⋮	⋮
	2400	0.05
	2700	0.05
1.0e-9	100	0.71
	200	0.24
	⋮	⋮
	800	0.12
	900	0.07

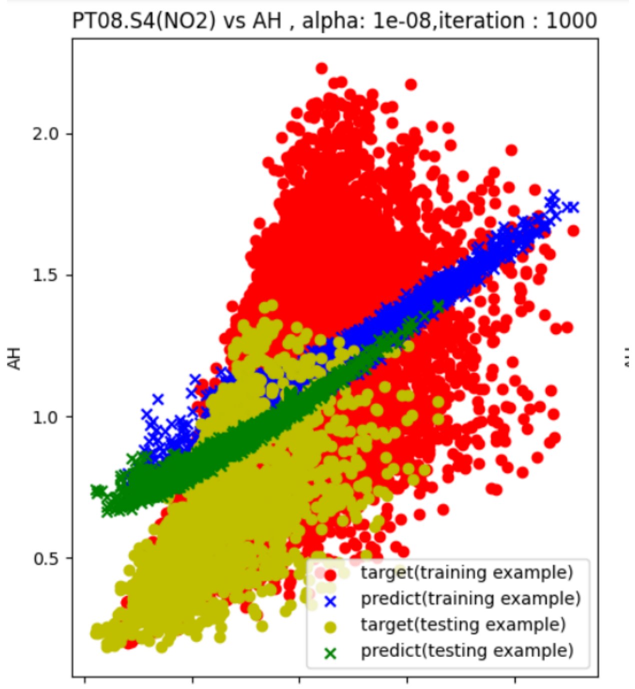
α	Iterations	Target Value	Predicted Value
1.0e-7	1000	0.84	0.77
	1000	0.80	0.77
	1000	0.81	0.77
1.0e-8	1000	0.84	0.86
	1000	0.80	0.86
	1000	0.81	0.86
1.0e-8	3000	0.84	0.80
	3000	0.80	0.80
	3000	0.81	0.81
1.0e-9	1000	0.84	0.80
	1000	0.80	0.80
	1000	0.81	0.84

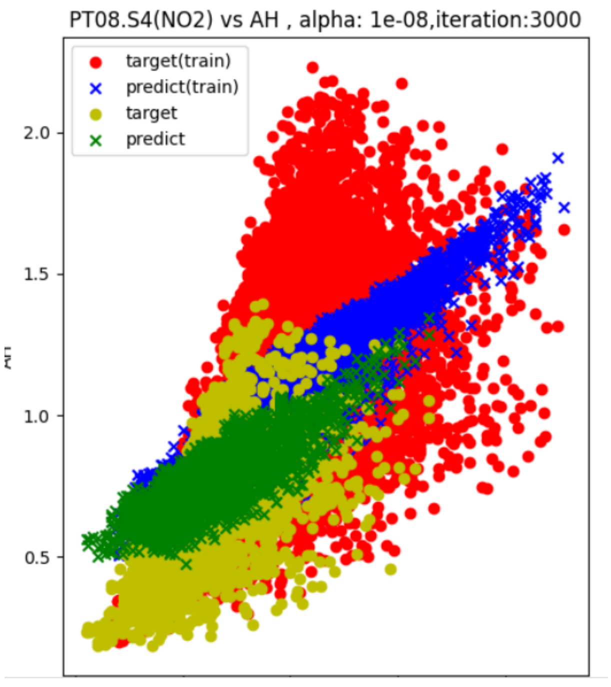
From the above tables, we can see that with different learning rates, the cost is reduced. However, when taking a relatively larger alpha value ($\alpha = 1.0e - 3$), the cost increases instead (Figure 1.1). Otherwise, in another case, the cost function decreases with iteration steps.





Again, with the same alpha value, the number of iterations is affected. If we look closely at the second table, the prediction is more accurate with a larger iteration(3000).





```
iterations = 1000 #3000
tmp_alpha = 1.0e-7 #1.0e-8,1.0e-9,1.0e-10

w_final, b_final, J_hist, p_hist
        = gradient_descent(x_train ,y_train, w_init, b_init,tmp_alpha,
            iterations,compute_cost, compute_gradient)

print(f"(w,b)␣found␣by␣gradient␣descent␣(alpha␣=␣␣␣␣␣␣␣␣{tmp_alpha:.1e}):
({w_final},
{b_final:8.4f})")

for i in range(5):
  print(f"alpha:␣{tmp_alpha:.1e},␣target␣value:␣{y_test[i]:0.2f},
␣␣predicted␣value:␣{np.dot(x_test[i],␣w_final)␣+␣b_final:0.2f}")

# plot cost versus iteration
fig, (ax1) = plt.subplots(1, 1, constrained_layout=True, figsize=(3,3))
ax1.plot(J_hist[:iterations])
ax1.set_title(f"Cost␣vs.␣iteration(start)␣:␣alpha␣={tmp_alpha}");
ax1.set_ylabel('Cost')            ;
ax1.set_xlabel(f'iteration␣step({iterations})')  ;
plt.show()


print(x_train.shape)
print(x_test.shape)
```

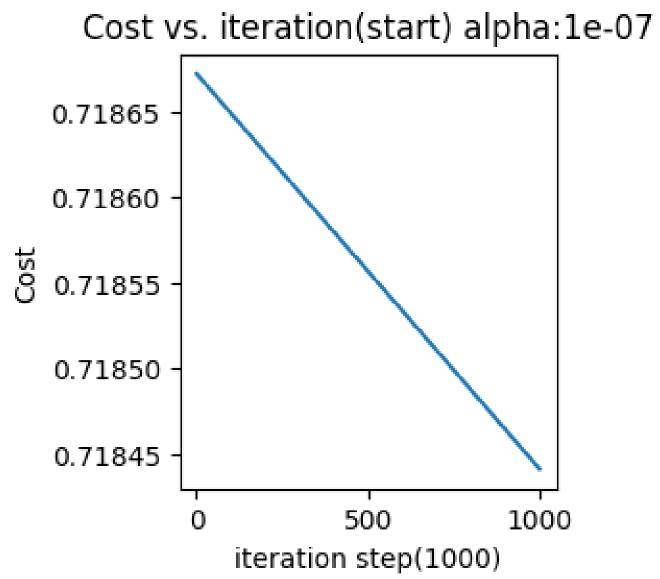
B. *After Scaling*

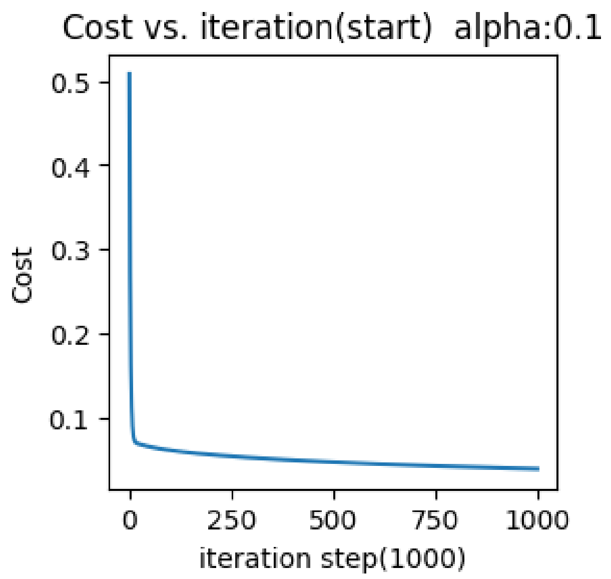
- After scaling, we tried to check out the output for different iteration values and learning rates. This time, we were able to find the cost function for a relatively larger value of alpha. This time, for some larger alpha values ($\alpha = 1.0e-1, 1.0e-3$), we got a better prediction than the smaller value ($\alpha = 1.0e - 10, 1.0e - 7$).

Alpha	Iteration	Cost
1.0e-3	100	0.72
	200	0.52
	⋮	⋮
	800	0.11
	900	0.10
1.0e-1	100	0.51
	200	0.06
	⋮	⋮
	800	0.04
	900	0.04
1.0e-7	100	0.72
	200	0.72
	⋮	⋮
	800	0.72
	900	0.72
1.0e-10	100	0.72
	200	0.72
	⋮	⋮
	800	0.72
	900	0.72

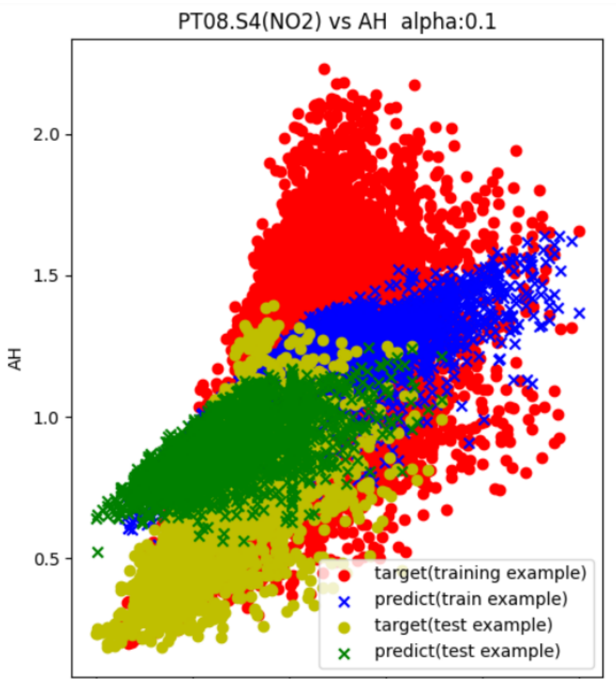
α	Iterations	Target Value	Predicted Value
1.0e-1	1000	0.84	0.89
	1000	0.80	0.80
	1000	0.81	0.85
1.0e-3	1000	0.84	0.80
	1000	0.80	0.76
	1000	0.81	0.78
1.0e-7	3000	0.84	0.00017
	3000	0.80	0.00017
	3000	0.81	0.00016
1.0e-10	1000	0.84	0.00000
	1000	0.80	0.00000
	1000	0.81	0.00000

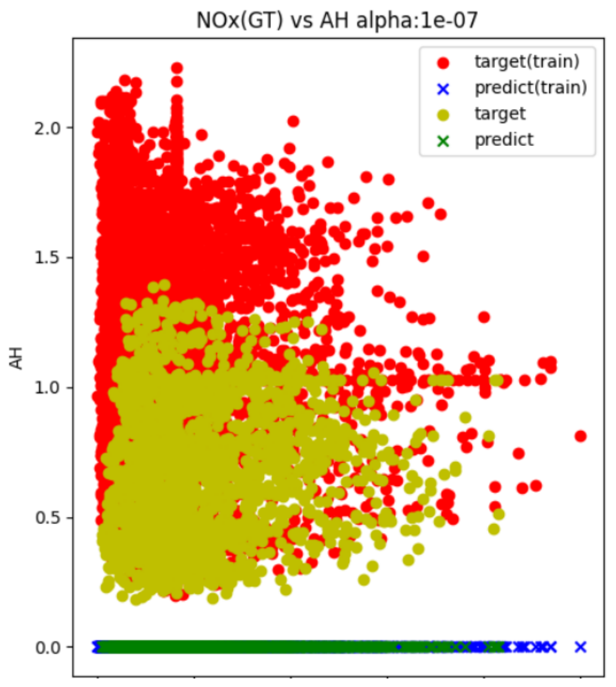
After scaling cost vs Iteration for ($\alpha = 1.0e-1, 1.0e-7$)





Prediction for training and testing values when $\alpha = 1.0e-1, 1.0e-7$





IV. Discussion

In this section, we discuss key aspects of our analysis and implementation.

A. Data Analysis and Pre-processing

We meticulously analyzed the dataset, employing data pre-processing techniques. This involved dropping unnecessary columns and handling missing values, crucial steps to ensure the quality and reliability of our analysis.

B. Cost Function and Gradient Descent Implementation

Creating and implementing the cost function and gradient descent algorithm were pivotal steps in our study. We delved into the transformation of univariate to vectorized implementation, gaining a deeper understanding of the underlying mathematical principles.

C. Effect of Learning Rate on Data

Our exploration included observing the impact of the learning rate on the dataset. Notably, we varied the learning rate and observed corresponding changes in the data. For instance, adopting a relatively low rate (e.g., 1.0×10^{-9}) without scaling yielded promising predictions on the test data. Furthermore, we discovered that iteration plays a crucial role in achieving better results with a larger iteration count (e.g., 3000) at the same learning rate.

D. Scaling and its Influence on Model Output

Our study delved into the significance of scaling in the context of model output. Without scaling, certain learning rates (e.g., 0.1, 0.01) increased the cost function instead of decreasing it. However, after implementing scaling, we observed improved outputs even for larger learning rates like 0.001.

These findings underscore the importance of thoughtful data preprocessing, algorithm implementation, and parameter tuning in the success of machine learning models.

References