1. Introduction

Regression is a statistical technique employed to describe the relation between one or dependent variables and one or more independent variables. Under regression, linear regression specifically deals with a linear relationship between the dependent and independent variables. In light of algorithms pertaining to machine learning, regression as a comprehensive technique is instrumental in predicting a continuous response from a set of continuous input values/*predictors*. Multiple linear regression in turn is applied for problems involving two or more features that may be linearly correlated with a single final quantity to be predicted.

2. About the Dataset

The dataset assigned for the task of multiple linear regression is namely the *California Housing Prices Dataset*.The data pertains to the houses found in a given California district and some summary stats about them based on the 1990 census data. As per the given dataset the final output quantity to be predicted is the median house value/price based upon the following set of features-latitude,longitude,median age of the house, total number of rooms, total number of bedrooms,population in the area, households in the vicinity and proximity to the ocean.

3. Data Pre-processing Measures

3.1. Removing NA values if any

This has been implemented via the pandas library of Python. In consequence to reading the csv dataset file into a pandas dataframe, using **.dropna()** method of pandas, all na values are removed from the dataset.

3.2. Feature Engineering(Dimensionality Reduction)

The ocean proximity does not account for a continuous distribution of values but 5 discrete values- <1H Ocean, Inland, Near Ocean, Near Bay, Island. Besides, the latitude and longitude of the house effectively condense its exact location and nearby geographical features, thereby accounting for this feature. Therefore this feature has been dropped-reducing the number of features to only 7.

3.3. Z-Score Normalization

The values in different feature categories (e.g., latitude, number of rooms, population) are measured on vastly different scales. This causes imbalances during gradient descent — features with large magnitudes may dominate the loss function, while features with smaller scales may have minimal impact. As a result, the model may converge slowly or inefficiently.

Z-score normalization addresses this by standardizing each feature to have a mean of 0 and a standard deviation of 1. This brings all features to a comparable scale, ensuring that the model treats them equally during training.

4. Approaching the Problem

The task is to predict the median house price based on a pre-processed set of features. To achieve this, three different approaches were explored. The first approach is a naive yet foundational implementation of multiple linear regression from scratch, using batch gradient descent as the optimization method to minimize the mean squared error (MSE) loss function.

MSE is a convex loss function in linear regression, meaning it has a single global minimum, which ensures that batch gradient descent can reliably converge without getting stuck in local minima. This is a key advantage, as more complex cost functions with multiple valleys or saddle points may trap gradient descent algorithms.

While stochastic gradient descent (SGD) can sometimes escape such traps due to its inherent randomness, it can lead to noisy updates and may take longer to converge in convex settings. Therefore, for a convex loss function like MSE in linear regression, batch gradient descent is often preferred due to its stability and direct path to the global minimum.

4.1. Figures

Fig. 10 shows a multidimensional visualization of the MSE cost function.

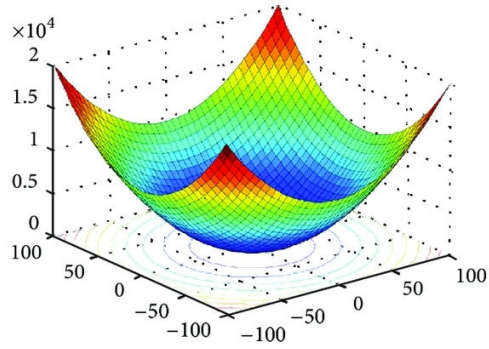

Figure 1. Mean Squared error cost function visualized in a multidimensional space

For multiple linear regression predicted value looks like:

$$f_{w,b}(X) = w_1X_1 + w_2X_2 + w_3X_3 + ... + w_nX_n + b \tag{1}$$

here, X_i represents the ith feature and b stands for bias, whereas w_i is the corresponding weight

$$\vec{w} = [w_1, w_2, w_3, ..., w_n] \tag{2}$$

$$\vec{X} = [X_1, X_2, X_3, ..., X_n] \tag{3}$$

therefore, the final predicted value can be represented with the help of vectors as,

$$f_{w,b}(X) = \vec{w}.\vec{X} + b \tag{4}$$

Our cost function can be defined as,

$$J_{w,b} = \frac{1}{2n}\sum_{i}^{n}(y^i - f_{w,b}(X)^i)^2 \tag{5}$$

Applying gradient descent, we update the weights and the bias until convergence,

$$w_i = w_i - \alpha\frac{\partial}{\partial w_i}J_{w,b} \tag{6}$$

for all weights corresponding to every feature, and the bias is updated as,

$$b = b - \alpha\frac{\partial}{\partial b}J_{w,b} \tag{7}$$

For the mean squared cost function,

$$\frac{\partial}{\partial w_i}J_{w,b} = \frac{1}{n}\sum_{i}^{n}(f_{w,b}(X)^i - y^i)x^i \tag{8}$$

$$\frac{\partial}{\partial b}J_{w,b} = \frac{1}{n}\sum_{i}^{n}(f_{w,b}(X)^i - y^i) \tag{9}$$

Here α refers to the learning rate, basically the size of every stride towards the minimum. If this value is too large, it may lead to overshooting and a very high deviation from the minimum. On the contrary if this value is too small, then the time of convergence is unecessarily high. In such a case we may define a *tolerance value*, ϵ, if the value of the gradient is smaller or equal to this value, we simply break out of the iteration with a safe assumption of having reached very near to

the minima.

5. Approach-1: The naive approach

The first approach is to implement multiple linear regression via a pure python based implementation. The ***pandas*** library, a powerful open source python library to analyse, visualize and manipulate tabular and statistical data, has been utilized to read the csv file, access the rows and columns for predictor values and actual value of the housing prices. Besides this, no other python library has been leveraged in this implementation and the entire algorithm optimized via gradient descent has been implemented from scratch using the above mathematical formulae and basic python lists. The dataset has been divided as such, that after randomized shuffling, 55% of the training samples have been allocated to the training dataset, 25% to the validation set and the remaining 20% to the test dataset.

The purpose of the validation set is to avoid overfitting on the training data, and to see how the algorithm generalizes on unseen data. The same thing could be achieved from the test set solely but the drawback is that if only the test set is used, the generalization would again be specific to the test data. Cross validation is another method, wherein training data is split into multiple parts and the model is trained on them before testing to check for the generalization.

Hyperparameters are values set to a specific number that remain constant while the model is trained, they basically are training parameters and not model parameters unlike the weights, which are model parameters getting updated with every epoch.The number of epochs and learning rate are two such parameters which in this approach have been tuned to 2000 and 0.03 respectively. All the weights and the bias are initialized to 0.55555556, a value tuned by training.

Fig. 10 shows a multidimensional visualization of the MSE cost function.

```
epochs=2000
alpha_lr=0.03

weight_vector=[0.55555556 for i in range(train_df.shape[1])]

feature_matrix=[]
for i in range(train_df.shape[0]):
    h=[]
    for j in range(8):
        h.append(train_df.iat[i,j])
    feature_matrix.append(h)

bias_scaler=0.55555556

train_samples=train_df.shape[0]

start_time=time.time()

sum_avg=0

cost_func=[]

for i in range(epochs):
    y_pred=[]
    for j in range(train_df.shape[0]):
        sum=0
        for q in range(train_df.shape[1]):
            sum+=weight_vector[q]*feature_matrix[j][q]+bias_scaler
        y_pred.append(sum)
    mse_loss=0
    for j in range(train_df.shape[0]):
        mse_loss+=(y_pred[j]-y_obs_train[j])**2
    mse_loss/=train_df.shape[0]
    cost_func.append(mse_loss)
```

Figure 2. Code snippet

6. Approach-2 : Numpy Optimized

Approach 2 effectively incorporates the use of NumPy — a powerful and fundamental Python library for working with arrays, matrices, and a wide range of numerical operations in data science. A pandas DataFrame is essentially a collection of Series, and each Series is internally implemented as a NumPy array with additional indexing features. Therefore, each column of a DataFrame, being a Series, can be directly extracted as a NumPy array using the .values attribute.

One of the biggest advantages of using NumPy is vectorization, or in other words, parallel vector processing.

Supposing in multiple linear regression, we have a vector of weights and features as follows-

$$\vec{w} = [w_1, w_2, w_3] \tag{10}$$

$$\vec{x} = [x_1, x_2, x_3] \tag{11}$$

Following the naive approach, in order to calculate the predicted value,

$$f_{\vec{w},b}(\vec{x}) = \sum_{j=1}^{n} w_j x_j + b$$

We need to iterate through all the features ranging from 1 to n,

```
for j in range(0,n):
    f+=w[j]*x[j]
f+=b
```

Figure 3. Code Snippet(Approach 1)

This is time consuming since, if one operation takes unit time, n operations takes n units of time, supposing 1 operation takes 1 ms and n=1000, thus overall time consumed becomes 1s, thousand times the duration for each operation.

This is where NumPy offers a major performance gain. When using np.dot(w, x), NumPy internally vectorizes the computation: it multiplies corresponding entries in parallel and sums them in a single, optimized operation. Hence, the entire dot product operation takes approximately the same time as one scalar operation, dramatically reducing runtime and improving performance.

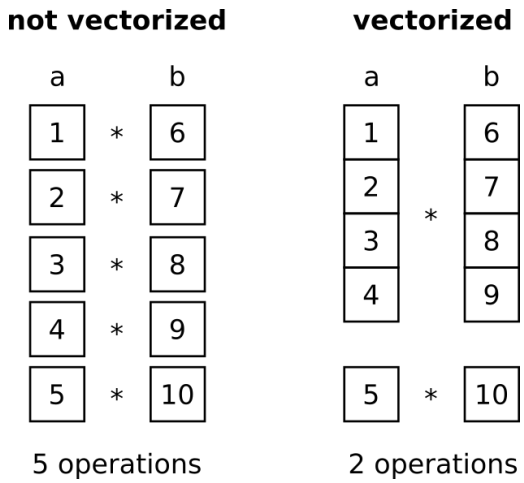

Figure 4. Vectorized vs Non Vectorized

The initialization of weights and bias along with the number of epochs and learning rate have been kept consistent with Approach 1 to draw a fair comparison. The only difference is computation happens in parallel, replacing many loops with single line expressions.

```
epochs=2000
alpha_lr=0.03
weight_vector=np.full((train_df.shape[1],),0.55555556)

feature_matrix=train_df.iloc[:,:8].values
bias_scaler=0.55555556

train_samples=train_df.shape[0]

start_time=time.time()

sum_avg=0

cost_func_2=[]

for i in range(epochs):
    beg_sec=time.time()
    y_pred_training=np.dot(feature_matrix,weight_vector)+bias_scaler
    cost_func_2.append(np.mean(np.square(y_pred_train-y_obs_train)))
    error=y_pred_train-y_obs_train
    grad_w=(np.dot(feature_matrix.T,error))/train_samples)*alpha_lr
    grad_b=(np.sum(error))/train_samples)*alpha_lr
    weight_vector=weight_vector-grad_w
    bias_scaler=bias_scaler-grad_b
    end_sec=time.time()
    sum_avg+=end_sec-beg_sec

end_time=time.time()

sum_avg/=epochs

print(f"Average time taken per epoch for calculating gradients and updating weights and bias using parallelization: {sum_avg:0.6f} seconds")
print(f"Converging time: {(end_time-start_time):0.2f} seconds")
```

Figure 5. Code Snippet(Approach 2)

7. Approach 1 v/s Approach 2

Approach 1	Approach 2
Converging time-151.2019 seconds	Converging time-0.81 seconds
MSE loss(test set): 0.36	MSE loss(test set): 0.36
RMSE loss(test set):0.61	RMSE loss(test set):0.60
R2 Score(test set): 0.64	R2 Score(test set): 0.64

Table 1. Comparison Table

Convergence of cost function v/s the number of iterations in both approaches 1 and 2

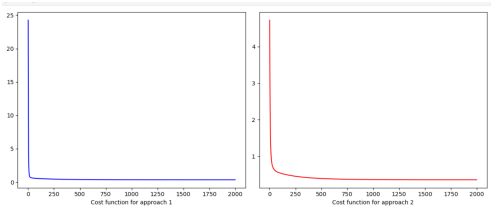

Figure 6. Comparison between the approaches 1 and 2

The differences in both approaches though applying the same algorithm and the same optimization technique of gradient descent can be elucidated upon as follows-

1. Vectorization with Numpy hugely reduces the time taken for the computations of the cost function, the predicted value and the gradient value in each iteration. While all these had to be implemented manually in the first approach, Numpy offers a parallel processing alternate thereby reducing the time of execution drastically from 151.209 seconds to 0.81 seconds

2.The code becomes concise since manual computation of cost function, predicted value and gradient value at every step is reduced to single expression assignments and computation, thereby decreasing the number of lines in the code.

3.The MSE, RMSE and R2 Score are nearly the same, infact MSE and R2 score are exactly the same indicating that the foundational mathematical logic and skeleton remains consistent across both the methods.

4.The final values of the Cost function are also the same, the only difference is the time of convergence.

Note

In the techniques above, the value of R2 score over validation and test set are pretty close,(0.63 and 0.64-for both approaches 1 and 2),indicating that the model generalizes well without employing regularization measures like ridge regression or elastic net thereby sticking to the simple straightforward approach.

8. Approach 3-Using Scikit-learn Library

Scikit-learn is a powerful, open-source machine learning library for Python. It provides simple and efficient tools for data mining, data analysis, and machine learning modeling. In this approach, we use the **LinearRegression** class from the **linear_model** module of **Scikit-learn** to perform multiple linear regression. An instance of this class serves as our model, which is then fitted to the training dataset. To ensure robust evaluation, the **train_test_split** function from **model_selection** is employed to divide the data — allocating 20% to testing and the rest to training. Additionally, we utilize cross-validation using the **cross_val_score** method from the model_selection module. A 5-fold cross-validation is performed, and the R^2 score is used as the evaluation metric. Internally, the LinearRegression class uses the *closed-form solution*, also known as the *Normal Equation*:

$$w = (X^T X)^{-1} X^T y \tag{12}$$

Here X is the matrix of features, y is the vector of actual values and w is the vector of the weights to be determined to minimize the cost function. Thus, the class does not apply gradient descent at all and in turn applies OLS Algorithm(ordinary least squares algorithm) which is independent of the initialization of weights or the bias and reduces the convergence time in first case to an extremely efficient fitting time of 0.009999 seconds.

Which approach stands the best?

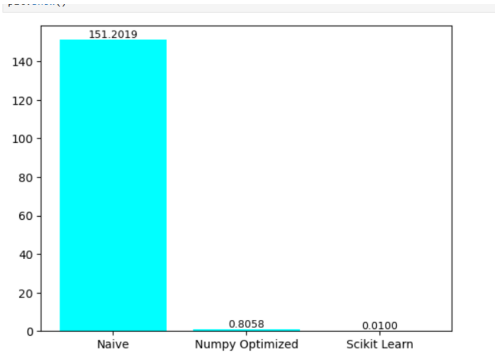

Figure 7. Convergence time across all three approaches

The naive approach employing pure python implementation and applying batch gradient descent for optimization takes the maximum time to converge-151.2019 seconds. Numpy optimization hugely reduces the time of convergence for the same initialization conditions and hyperparameter values(like number of epochs and learning rate).The duration of 151.2019 seconds is reduced to 0.8058 seconds, that is reduced by a factor of nearly 188. On the contrary, scikit learn library which utilizes the OLS algorithm for this task takes the minimum time 0.01 seconds, reducing the time of even the second approach by a factor of 8.

MSE,RMSE and R2 score evaluated on the test dataset

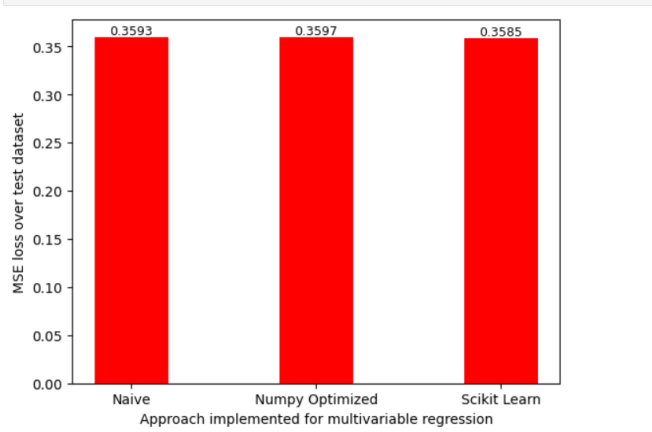

Figure 8. MSE over test dataset

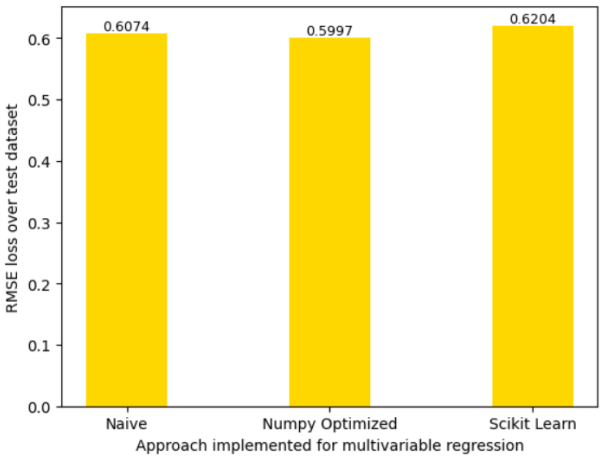

Figure 9. RMSE over test dataset

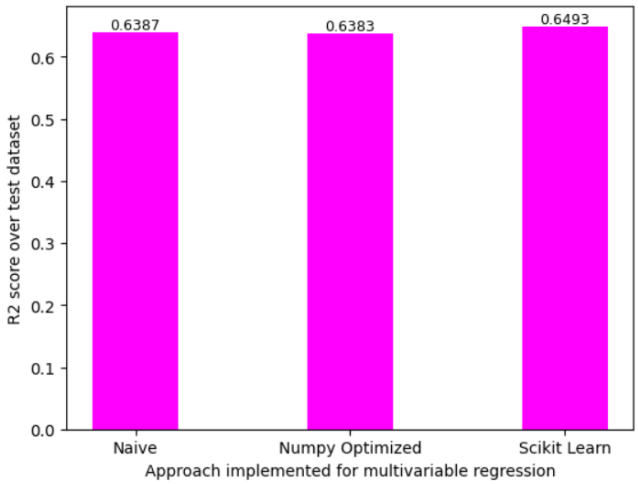

Figure 10. R2 Score over test dataset

Performance metrics as scrutinized in parallel across all three procedures yet again shows mostly consistent results, with all values being almost the same or differing by atmost 0.01. The entire procedure specifically across the approaches 1 and 2 remains exactly the same with exception to Numpy optimization gives identical values approximately. The value for R2 score is the highest compared on the small scale of 0.01 in the last approach, this may pertain to the tuning of hyperparameters in the first two approaches(Tolerance value ϵ has not been used pertaining to the less R2 score over the test dataset and lesser accuracy of final results). Since the last approach simply uses OLS algorithm or closed form solution, hence it is independent of any initialization values and this could be a potential cause behind a slightly higher R2 score. The convergence time for gradient descent is clearly higher, even the Numpy optimized approach is 8 times slower than a simple closed form solution. This happens as approaching the minima over a course of epochs definitely takes more time than the application of a direct mathemtical formula derived for our specific cause of Linear Regression(the closed form solution).

However gradient descent is a better approach in terms of its ability to generalize across algorithms as a universal optimization technique, which is not the case with closed form solution that specifically fits to our cause of Linear Regression.

Also the learning rate- too high a learning rate causes overshooting and deviation from minima for a given number of epochs and too small a learning rate may lead to too slow a convergence time

Scalability and Efficiency tradeoffs

1. Pure python implementation is simple, educational providing wholesome foundational understanding of multiple linear regression and gradient descent, however it isnt a practical approach in terms of convergence time and utilization of resources.It has poor scalibility-very large datasets with thousands of features require lot of space and time complexity($O(n^3)$). Computation is slow and lacks vectorization

2. As for the second approach, convergence time is less, vectorization implemented,better speed and resource use-Numpy arrays require lesser space as compared to ordinary python lists. This method also, though fails without distributed computing because large datasets yet again require very high CPU/GPU capacity along with higher efficiency RAMs. Thus this method is moderately scalable.

3.The third method is fastest of all, direct, with same or better accuracy(depends on hyperparamter tuning). It is moderately scalable, provided linear relationships between features and dependent variable yields acceptable results. The only drawback is the specificty of the closed form solution, gradient descent is more of a universal optimization technique, yielding generalized results. Otherwise, in terms of efficiency and resource usage catering to our given domain of linear regression, this method stands the best.

9. Resources Utilized

1.GitHub - IG repository The "Hands-On Machine Learning" resources available on the official IG GitHub repository were extensively referred to for understanding the theoretical concepts and practical implementations of various machine learning algorithms.

2.Andrew Ng's Course and Kaggle Blogs

3. ChatGPT (by OpenAI)-

Basic grammatical corrections in the report. Clarification of theoretical doubts. Removing bugs in source code.