Prerequisites

Sudarshan Kulkarni

1 Data Preprocessing

1. We fill the empty values of any columns appropriately. In this case, only *total_bedrooms* has empty values, and median was used to fill them.
2. One hot encode *ocean_proximity* as it is categorical data.
3. Use Sklearn's standard scalar to normalize mean to 0 and variance to 1 for all non-categorical features, and also for the labels.
4. Finally, split the data into train and test set in 7:3 ratio (no significance in this specific split. could've chosen 6:4 or 8:2 as well)

2 Assumptions

1. Formula used for Mean Absolute Error (where **m** is number of training examples):
$$\frac{1}{\mathbf{m}}\sum_{i=1}^{m}|\mathbf{y_i}-\widehat{\mathbf{y}}_\mathbf{i}|$$
2. Formula used for Mean Squared Error (Also the cost function used for gradient descent in both part 1 and 2):
$$\frac{1}{2\mathbf{m}}\sum_{i=1}^{m}(\mathbf{y_i}-\widehat{\mathbf{y}}_\mathbf{i})^2$$

3 Pure Python Implementation

The train, test datases were converted into python lists before making more calculations

3.1 Functions

1. **predict**: for predicting labels, given features and parameters. Handles both the cases for 2d and 1d array.
2. **mae_cost**: Calculation of Mean Absolute Error across full data

3. **mse_cost**: Calculation of Mean Square Error. We can just take square root to get RMSE as well.

4. **r_squared**: Calculates R^2 Score, according to the definition on wikipedia

5. **compute_gradient**: Calculate $\frac{\partial J}{\partial w_i}$ and $\frac{\partial J}{\partial b}$ according to the assumed cost function

6. **gradient_descent**: perform batch gradient descent for the number of epochs given. Also calculates and stores the time taken for each step, and the error metrics after each step for both training and test sets.

3.2 Results

1. Model was trained for epoch=1000, with learning rate 0.2, this learning rate was chosen after a lot of testing with different learning rates among, 10^{-5}, 10^{-3}, 0.1, 0.5, 1

2. The graphs plotting all 4 metrics v/s Epoch are present in the file *python_implementation.ipynb*

4 Numpy Implementation

1. All the functions in pure python implementation are also defined for numpy implementation. Although each one of them takes advantage of vectorization to maximum possible extent.

2. Same learning rate of $\alpha = 0.2$ and 1000 epochs as the 1st part was used for this.

3. The results for all metrics are almost the same as part 1 except the time taken. This is because it is implemented in a similar way to make appropriate comparisons on time only.

5 Scikit-Learn Implementation

Scikit-learn directly calculates the parameters using various techniques, which all directly result in parameters with lowest *L2 Norm*. But its interesting to know the details of how it works, so, lets see.

5.1 First Attempt

I tried implementing what sklearn does by using the first formula I got from googling,

$$\hat{\beta} = (\mathbf{X^T X})^{-1} \mathbf{X^T y}$$

where $\hat{\beta}$ are the parameters (weights + bias, together). $\mathbf{X}$ are all the independent variables, along with a column of *1*'s padded (for the bias), and $\mathbf{y}$ is the dependent variable.

But there is an obvious problem in this. What if $\mathbf{det}(\mathbf{X}) = \mathbf{0}$? It would mean the inverse in the equation would not exist! This will most likely not be a problem, unless we one hot encoded our categorical data, which is exactly what we did 🤦‍♂️. This is a problem because then the sum of the columns we just added is always 1. So it means they're linearly dependent, making the determinant zero.

As they are dependent, we can just remove one of the columns, and it would be fixed, by using *drop_first=True* in *pd.get_dummies* function. After this, the parameters obtained by sklearn and by using the above formula are exactly same.

5.2 What Sklearn internally does?

It uses various techniques, and but not the formula discussed above as it is not *stable*, that is, could result in problems discussed when determinant is zero. It calls scipy functions internally like *lstsq*, which in turn calculates Singular Value Decomposition (SVD) of matrix $\mathbf{X}$, which is used to compute the least-squares solution. I believe understanding much deeper requires learning more maths, which I current don't know. So I will be visiting this topic in future again

6 Final Results

RMSE, MAE and R^2-Score are taken after 1000 epochs for part 1 and 2. The Time taken was for 300 epochs. 300 was chosen after seeing that the graph becomes for all metrics from quite a bit before 300, so we can assume that it is near the minima.

Note: these are calculated on training data. RMSE is the root of the MSE formula stated before. Hence it differs by a factor of $\sqrt{2}$ from the actual formula.

	Pure Python	Numpy	Scikit Learn
RMSE	0.4216	0.4216	0.4214
R^2 Score	0.647	0.647	0.647
MAE	0.4311	0.4311	0.4309
Time Taken(s)	61.7	10.34	0.008

1. Time taken is quite long when using pure python. It can be concluded that using Numpy is always beneficial as it reduces time by a huge factor. In this case, only 1 reading is taken, but numpy vectorization can be even 500 times faster. source.

2. All metrics are same till 4 digits after decimal in 1st and 2nd case, as the functions are all almost same, and both run for 1000 epochs.

3. Time when using scikit-learn

4. For implementation of unregularized linear regression, scikit-learn is the way to go, as it finds the parameters almost instantly. Otherwise in general case, for other models when its not possible to directly find the parameters, gradient descent with vectorization in NumPy is the way to go

5. Initialization of parameters in parts 1 and 2 were all zero, as there isn't much of a difference. They will all eventually converge as the cost function is convex. For Sklearn, there is no initialization as parameters are directly calculated.