Experimental Findings from Multivariate Linear Regression on Housing Price Prediction

Md Farhan Ishmam[1]

[1]Department of Computer Engineering, Islamic University of Technology, Gazipur, Bangladesh
farhanishmam@iut-dhaka.edu
ID: 180041120
Section - 1A

Keywords

Multivariate Linear Regression, Data Preprocessing, Normalization, Gradient Descent, Learning Rate, Mean Square Error (MSE), Forward Propagation, Backward Propagation, Epoch, Hyperparameter, Cost, Linear Model, Training Set, Testing Set, Pearson Correlation Matrix, Feature Selection, Loss

1. Introduction

The objective of the machine learning model is to perform linear regression with multiple variables on a provided dataset of housing prices. We start off by loading the dataset and enriching our knowledge on the dataset using functions and necessary graphs. Then some fundamental data pre-processing steps are performed. Afterwards, we define necessary functions to perform gradient descent and also perform feature selection using the Pearson Correlation Matrix. Using the hand-picked features, we finalize our model by minimizing the error.

2. Dataset and Preprocessing

2.1 Summary Statistics and Data Visualization

After loading the provided dataset, we try to have a rough idea of the dataset using statistical summarization functions and visualizations. Some of the steps we took are:

I. Checking the number of samples in the dataset using len() on the indices.
II. Checking the names of the columns.
III. Calculating the number of null values in each column using isnull() and sum().
IV. Checking the datatype of each column.
V. Looking at the first 5 columns of the dataset using head().
VI. Looking at the summary statistics of the dataset using describe().

2.2 Key Findings

The key takeaway is that our dataset has no null values which makes data cleaning easier. Usually columns with a higher number of null values are omitted.

The second major finding is that there are some columns with string data type. These columns might be used as features if we can map the strings to some integer values. For example - the city New York is mapped to 1, Los Angeles is mapped to 2 and so on. The address does play an important role in conveying extra information from the dataset. For example - the house prices in New York are relatively higher than prices in Cleveland. If our model is able to learn and differentiate regions then it will make a better prediction of house prices in Cleveland or New York. A major drawback can be - if there's a new city, let's say San Jose, in the test set. The model won't be able to use that piece of data since it doesn't know if prices in San Jose are higher or lower. It is best to treat such values as null during test time.

2.3 Data Preprocessing

After getting a rough sketch of the data, we will now perform the necessary pre-processing steps:

I. Extract the input features and predicted column from the dataset.
II. Shuffle and split the dataset into training and test set at 80-20 splits

The date column was omitted as the entry date has no significance on predicting the housing price. The string data type columns were omitted due to the complications of mapping the column (the task to map the columns was also not mentioned in the given assignment)

2.4 Data Splitting & Shuffling

The splitting was done in 80-20 splits because we need a larger set of data for training our model, and a smaller set of data to test it. A question might arise, why do we even need to split the data in the first place and use the same dataset for both training and testing? The reason is that if we use the whole data to train the model then we can't tell how well the model will perform on data it hasn't seen before. Models are often seen to be performing extremely well on the training dataset and performing relatively poor on the testing dataset. This problem is known as overfitting, and can be identified if we have separate training and testing sets.

We also need to shuffle the data before splitting it or else the splits won't be homogenous. For example - let's assume our dataset is sorted by price. Then the first 80% of the data will have relatively lower prices and will go to the training set. And the last 20% of the data will have relatively higher prices and will go to the test set. As our model was trained on houses with lower prices, it will perform poorly on predicted higher priced houses. Such an outcome is undesirable and hence, splitting is done.

During splitting the dataset, we haven't considered the cross-validation dataset. We use this dataset to tune the hyper-parameters (parameters that aren't automatically learnt by our model) The cross-validation dataset can not be the same as test dataset or else the model will be fine-tuned to the test set and we won't be able to evaluate the model on new data. Usually, if we want to include the cross-validation set, we make 80-10-10 splits.

Another factor to consider while making splits is the number of samples in our dataset. Since, the number of samples is relatively small (4600 samples), keeping 20% data for testing is enough. However, if we had a large number of samples, for example - a million samples, then taking 1% for testing should be enough. The rest of the data can be used for training and it is often seen that having more data for training improves the model's accuracy in predictions.

3. Model Training

The whole procedure of training our model was divided into several parts. Firstly, the necessary helper functions were created that will be used along with the gradient descent function to train the model. Afterwards the gradient descent function is defined along with slight modifications of early stopping. Finally, the necessary graphs are plotted.

3.1. Helper Functions

Three helper functions were defined to calculate mean square error, to concatenate a column of ones and to normalize the dataset. It is to be noted that all three helper functions are available in various python libraries but to perform these steps from scratch, we manually define these functions.

3.1.1 Mean Square Error

The mean square error calculates the loss of all the training examples in our dataset. The loss is the square difference of the predicted values from the model and the actual values. The individual losses of each training example is summed over and divided by 2*m, where m is the number of training examples. An extra 2 is used to divide the loss because the derivative of the square loss produces an extra 2 which cancels out this 2. We will see this while calculating the gradient.

3.1.2 Concatenate One

To perform a vectorized form of batch gradient descent, a row of ones is concatenated to the input matrix, increasing the number of feature dimensions by 1. The one is concatenated because the linear model has a parameter for each feature and an extra bias term. We can keep this bias term separate from our parameter matrix, theta and add it to the product afterwards. However, for simplicity, instead of separately keeping the bias term, we concatenated the bias term with the theta matrix and added a dummy feature (a row of ones) with the input matrix. During matrix multiplication the bias column of the theta matrix will be multiplied with the ones row of input matrix and hence, the product will essentially be the same as adding the bias term.

3.1.3 Normalize

The normalize function performs feature scaling on a given matrix. Feature scaling is important as it transforms a matrix with any value to a matrix with values within [0,1]. For example - the area of the house can be from 100 sq ft. to 10,000 sq ft. When these values are taken as inputs then this feature might have a higher impact on the model compared to other features such as number of rooms which is usually between 1 to 10. By performing feature scaling, all the input feature values are scaled within [0,1]. In this case, 100 sq ft. becomes 0 while 10,000 sq ft. becomes one. This type of feature scaling is known as feature normalization where the minimum value is subtracted from each value and is divided by the range of values. Another popular form of feature scaling is known as feature standardization which subtracts the mean from each element, and divides by the standard deviation. However, we prefer feature normalization for scenarios like these where values within [0,1] are more suitable.

One key experimental finding is that if feature scaling was not performed on this housing price dataset then the *gradients explode*. After 10 epochs, it was found that the gradients become absurdly large, and the cost tends to *increase* as well. After performing feature scaling, this issue was resolved.

3.2. Gradient Descent

3.2.1 Initialization

After defining the function, some slight preprocessing and initialization steps are taken:

I. Initialize the cost_list to an empty list which will store the M.S.E from each epoch.
II. Concatenate ones to the input vector x using concat_one function.
III. Retrieve the number of training examples, m and the number of input features, n from the matrix x. Here, n is the original number of input features plus 1 as a row of ones had been concatenated.
IV. Initialize the parameter matrix theta to zeros.

There is a certain drawback of initializing theta to zeros. While updating the parameters through gradient descent, there tends to be a symmetry between the theta values. This is more prominent in larger models such as neural networks, or deep neural networks. However, for a simpler model such as linear regression, this doesn't create any issue as we have a well-defined convex cost function. We tried using zero initialization, random initialization and He initialization, and found no significant difference in the cost. We decided to stick with zero initialization for its simplicity.

3.2.2 Forward Propagation

During every epoch, a single step of forward propagation and a single step of back propagation is performed. During forward propagation, the inputs are multiplied by their corresponding weights and summed over to produce the predicted output. There will be a predicted output for each training example. After a single step of forward propagation, the cost is calculated using the mean square error function which takes this predicted value and the actual value. The cost is appended to the cost list.

3.2.3 Backward Propagation

Backward propagation has two key steps - calculating the gradient and updating the parameters. The gradient is calculated by differentiating the cost function with respect to the parameter matrix theta. The gradient matrix is composed of n values , where the i th value is the derivative with respect to the i th theta value. The gradients are stored and then used to simultaneously update the parameter matrix. If the values aren't stored then simultaneous update won't be possible.

During the update step, the gradients are updated by a scalar factor alpha which is called the learning rate. The learning rate helps gradient descent to converge and also prevents it from overshooting. The learning rate is a hyperparameter which will be tuned later.

To visualize gradient descent, it is common to use the analogy of a blind man going downhill holding a stick on his hand. Using the stick he finds out the slope of the hill. The hill is analogous to the cost function, and the slope of the hill is analogous to the gradient. If the man finds that the slope is too much, he can tell that the valley or the minimum point is further away from him and will take a larger step. If he finds the slope to be little, then he can tell the minimum point is close to him and will take a smaller step. The length of the step he takes is analogous to the learning rate alpha. If the learning rate is high, he will take a larger step and if it's low then he will take a smaller step. This analogy is particularly useful as our linear model is like a blind man and has no means to perceive the location of the real value except using his stick to find the slope of the hill.

3.2.4 Batch vs Mini-batch vs Stochastic Gradient Descent

As the forward and backward propagation is done over m training examples, it is known as *batch gradient descent* where m is the total number of training examples. However, there are other ways to perform gradient descent. Instead of using all the training examples, we can use a single training example for a single step of gradient descent. This will make each step quicker. However, each step won't be as accurate as the batch gradient descent step since a single training example is used and it might misguide the model to increase the cost. This type of gradient descent is known as *stochastic gradient descent.* There is a third option of using a batch of size b which is $1<b<m$. This type of training is somewhat in between batch and stochastic gradient descent and has the characteristics of both of them. It is called mini-batch gradient descent and is quite popular in training models with larger datasets.

In our machine learning task, the training set isn't excessively large. The number of features is also quite low. Hence, a single step of batch gradient descent didn't take much time. If we had a larger dataset, we might have to choose mini batch gradient descent by taking a small number of training examples, let's say 256 for each mini batch. If the number of input features was really high, and we were training a deeper or more complex model, then each step would take a lot of time for both batch and mini batch gradient descent. In that case, we would have selected stochastic gradient descent. However, since we aren't facing any of those problems, we will be using batch gradient descent in this task.

3.2.5 Vectorization

Vectorization allows us to perform loop-based operations faster by converting them to arrays or matrices and performing matrix operations. For example, the multiplication of weight with input features could have been done iteratively. But instead, we made an input matrix and a weight matrix and performed matrix multiplication. This saves time as python loops are slower, and numpy arrays are much faster as they are built on top of C++ arrays. Using vectorization can speed up gradient descent as much as 10 times.

3.2.6 Early Stopping

When there is no significant change in the cost after a step of gradient descent, then there is no need to proceed with gradient descent till it reaches the number of epochs. We can estimate that gradient descent is close to convergence and stop the algorithm. This is known as early stopping. In the plot, we can see that we have stopped gradient descent when the cost difference is less than a certain value, epsilon, which can be treated as a hyperparameter of our model. Early stopping saves training time.

3.2.7 Hyperparameter Tuning

Hyperparameters are the parameters which are not automatically learnt by the model. Parameters such as weights to the inputs stored in the theta matrix are automatically learnt while parameters such as learning rate, number of epochs, value of epsilon are not automatically learnt by our model. To get a proper value of these hyperparameters, we need to try and test different values.

For our learning rate alpha, we plotted the cost at different values of alpha. The values are taken at multiples of 3 i.e. 0.001 to 0.003 to 0.01 to 0.03 and so on. For the number of epochs, we trained till 1000 epochs as it is a significantly smaller model and relied on early stopping to save training time. For the value of epsilon, we minimized the value by a factor of 10 until the algorithm stops right after gradient descent tends to converge. As given in the task, a value of 0.5 had to be selected for epsilon but experimental findings showed 0.00005 to be more useful.

4. Feature Selection

4.1.Pearson Correlation Matrix

The Pearson Correlation Matrix tells us how correlated two features are within a matrix. It gives a matrix of all such correlation values. From the matrix, we can tell how correlated an input feature is to the predicting output value. The inputs with higher correlation contribute more to the predicting value. For example - the area of the house has a strong correlation with the predicting column price while year renovated has a low correlation. We can tell that the price is more dependent on the area and less dependent on the year renovated and can hence remove the year renovated column from our input feature vector of the model.

Another use case of the correlation matrix is to see how much two features are correlated. For example - area above and area living have a really high correlation with each other. So, they are essentially behaving as the same input feature. Hence, keeping both of them won't be useful and we can keep only the one which has higher correlation with the price.

4.2.Rationale behind Feature Selection

A number of features were omitted during feature selection. The steps performed are explained below -

I. Year built, year renovated, sqft_lot, and condition are removed as they have low correlation with the price of the house.
II. Sqft_living and sqft_above have high correlation with each other. Sqft_living is taken as it has a higher correlation value with price. Similarly, sqft_living has a high correlation with bedrooms, bathrooms, and sqft_basement. These columns were omitted.

Here, any correlation value higher than 0.4 was considered as high correlation while a value lesser than 0.1 was considered as insignificant or low correlation.

5. Justification of Obtained Results

After performing feature selection, the cost decreased a bit. That means, the gradient descent algorithm performed better on the dataset with selected features. Some of the omitted features didn't help the model learn new things about the data. Instead, considering those features made gradient descent perform worse on predicting outputs from the test set.

An example of such a redundant feature is the date of an entry. The date of the entry is completely random and has no correlation with the price of a house. However, if we allow gradient descent to consider this column as a feature, then gradient descent will try to put some weights to these date values. The weights are usually low but in most of the cases, they will move the predicted value away from the actual value due to the randomness of the date feature.

Another example of a redundant feature is the sqft_above which is similar to the sqft_living. Both of these features contribute similarly to predicting the price as they have high correlation with each other. Hence, treating them as separate features makes gradient descent learn extra parameters where it is not necessary. Instead if we treat them as a single feature, i.e. omit one of them then gradient descent can easily assign the same weight and will perform better on this reduced feature set. Due to excluding these redundant features, the cost decreased when we predicted using the feature selected model.

6. Conclusion

The overall procedure of predicting house prices using multivariate gradient descent has several key steps and several design choices that were made to achieve higher accuracy. While fundamentally, the algorithm might behave the same, the design choices allow us to achieve that higher accuracy saving time and computation resources. Through experience, we learn to make better decisions, especially during model selection, feature selection and hyperparameter tuning. With a good amount of data available, continuously improving the algorithm through slight modifications can allow us to make near to perfect predictions.

References:

[1] Machine Learning - Andrew Ng

https://www.coursera.org/learn/machine-learning

[2] Applied Data Science with Python - Christopher Brooks

https://www.coursera.org/specializations/data-science-python

[3] House Price EDA - Kaggle

https://www.kaggle.com/negarev/beginners-start-here-house-prices-eda

[4] Beginner's Guide to Pearson's Correlation Coefficient - Analytics Vidhya

https://www.analyticsvidhya.com/blog/2021/01/beginners-guide-to-pearsons-correlation-coefficient/

[5] NumPy, SciPy, and Pandas: Correlation With Python - Real Python

https://realpython.com/numpy-scipy-pandas-correlation-python/